Tampereen yliopisto

Miska Melkinen

RAJAPINTADOKUMENTAATION MUODOSTAMINEN OSITTAIN AUTOMATISOIDUSTI LÄHDEKOODISTA

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Huhtikuu 2019

TIIVISTELMÄ

Miska Melkinen: Rajapintadokumentaation muodostaminen osittain automatisoidusti lähdekoodista
Diplomityö
Tampereen yliopisto
diplomi-insinöörin tutkinto-ohjelma
Huhtikuu 2019

Ohjelmointirajapintojen dokumentaatio on tärkeässä osassa rajapintojen käytön oppimisessa ja oikeanlaisessa käytössä. Dokumentaation virheettömyys ja ajantasaisuus ovat tärkeitä elementtejä dokumentaation laadun kannalta. Rajapinnan dokumentaatio täytyy yleensä aina päivittää, kun rajapinnan totetutus päivittyy. Dokumentaation uudistaminen voi olla työlästä ja manuaalisesti dokumentoidessa voi syntyä helposti virheitä tai rajapintaan lisättyjen uusien ominaisuuksien dokumentaatiota ei välttämättä muisteta lisätä.

Muodostettaessa dokumentaatiopohja automaattisesti lähdekoodista voidaan varmistua siitä, että dokumentaatiosta löytyvien rakenteiden nimet, tyypit ja ominaisuudet vastaavat senhetkistä toteutusta. Dokumentaatiopohjan muodostaminen ei kuitenkaan poista ohjelmistokehittäjän vastuuta rakenteiden kuvauksien kirjoittamisesta dokumentaatioon, mutta vähentää manuaalisesti suoritettavan työn määrää dokumentoidessa.

Työssä suunniteltiin ja toteutettiin työkalu ohjelmointirajapintojen dokumentointiin. Toteutetun työkalun avulla pyrittiin helpottamaan dokumentointiprosessia sekä varmistamaan rajapinnan toteutuksen ja rajapinnan dokumentaation vastaavuus toisiinsa nähden. Rajapintadokumentointityökalu lukee rajapinnan lähdekoodia ja muodostaa rajapintaa kuvaavan tietomallin hyödyntäen lähdekoodia. Dokumentoitavan rajapinnan lähdekoodiin lisättiin attribuuttiluokkia, joilla voidaan kirjata lisätietoja dokumentaatiotyökalun jäsentäjälle. Muodostettua rajapinnan tietomallia voidaan muokata sekä tallentaa siihen kuvaustekstejä työkaluun toteutetun graafisen käyttöliittymän kautta.

Rajapintaa kuvaavan tietomallin muodostamisessa käytettiin .NET-kehyksen kääntäjään perustuvaa Roslyn-työkalua, jonka avulla voitiin tarkastella C#-ohjelmointikielellä kirjoitetusta lähdekoodista käätämisen jäsennysvaiheessa muodostettua syntaksipuuta. Dokumentoitavien rakenteiden tunnistamiseen syntaksipuusta käytettiin työssä suunniteltuja attribuuttiluokkia, joita oli lisätty dokumentoitavan eRA-järjestelmän koodiin. Attribuuttiluokkien avulla kirjattiin lähdekoodiin dokumentointia helpottavia lisätietoja, kuten rajapintaversionumeroita, metodien poikkeuksia ja metodien kutsumiseen vaadittavia esiehtoja. Tietomallista voidaan muodostaa toteutetulla dokumentointityökalulla LaTeX-muotoinen dokumentaatio.

Toteutetulla dokumentointityökalulla tuotettiin dokumentoitavaksi valitun Atostek eRA -järjestelmän integraatiorajapinnan uuden version dokumentaatio. Työkalun käyttäminen nopeutti dokumentaation muodostamista verrattuna edellisten dokumentaatioversioiden kirjoittamiseen. Toteutetun työkalun toimintaa testattiin tuottamalla jo olemassa oleva rajapintadokumentaatio uudestaan ja vertaamalla sitä manuaalisesti kirjoitettuun dokumentaation. Työkalun ansiosta havaittiin virheitä edellisissä dokumentaatioversiossa, joten työkalun käyttäminen paransi myös dokumentaation laatua.

Avainsanat: dokumentointi, ohjelmointirajapinta, rajapintadokumentaatio

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

ABSTRACT

Miska Melkinen: Interface documentation generation partially automatically from source code
Master of Science Thesis
Tampere University
master of science programme
April 2019

Documentation of application programming interfaces is in major role when learning to use interfaces properly. Correctness and up-to-date state of documentation are important aspects concerning the quality of documentation. Interface documentation needs to be updated when implementation of interface is updated. Updating interface documentation can be laborious and when documentation is generated manually there is a risk to forget to add documentation for newly added features or to make other mistakes easily.

When documentation template is generated automatically from source code correct names, types and properties of documented structures can be assured. Generating documentation template programmatically does not remove software developer's responsibility to write descriptions of interface structures to documentation but amount of manually executed work is reduced.

This work consists of design and implementation of a software tool to document application programming interfaces. Implemented tool aims to ease documentation process and ensure equivalence of interface implementation and interface documentation. Application interface documentation tool reads interface's source code and creates information model to describe analyzed interface based on it's source code. Attribute classes were added to documented interface source code to provide additional information to documentation tool's source code parser. Generated information model can be modified and documentation texts can be written to it by using tool's graphical user interface.

A tool called Roslyn was used to generate information model describing the application programming interface. Roslyn is based on .NET framework compiler and it provides functionalities to analyze syntax trees cteated from C# source code in parsing stage of compilation. Designed attribute classes were added to documented software's interface and used to detect structures that need to be added to documentation. By using attribute classes it was possible to ease documentation process by adding heplful additional information such as interface version numbers, method exeptions, and method preconditions to source code. Documentation tool can produce a text file containing documentation with LaTeX syntax.

Implemented tool was used to produce a new version of Atostek eRA software's integration interface documentation. Generating new version of interface documentation with implemented tool was faster compared to writing previous versions manually without using any documentation tool. Tools correct operation was tested by regenerating existing interface documentation and comparing it to previously manually generated documentation. Due to using documentation tool some mistakes in existing documentaion were found, therefore by using implemented documentation tool the quality of documentation was improved.

Keywords: documentation, application programming interface, interface documentation

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

ALKUSANAT

Työn tarkastajana sekä ohjaajana toimi professori Hannu-Matti Järvinen jota haluan kiittää työn tekoa ohjaavista kommenteista sekä palautteesta. Tahdon kiittää Atostekia työn toteuttamisen mahdollistamisesta. Kiitän myös Jeri Haapavuota, joka toimi työni ohjaajana Atostekilla, sekä työn toisena tarkastajana. Häneltä sain ohjeita dokumentointityökalun toteuttamiseen liittyvissä asioissa, sekä apua eRA-järjestelmän toiminnan ymmärtämisessä.

Tampereella, 10. huhtikuuta 2019

Miska Melkinen

SISÄLLYSLUETTELO

Lyhenteet ja merkinnät vi
1 Johdanto 1
2 Teoriatausta 3
2.1 Ohjelmointirajapinta 3
2.1.1 XML-rajapinta 4
2.1.2 .Net-rajapinta 5
2.1.3 ActiveX-rajapinta 6
2.2 Rajapinnan dokumentointi 6
2.3 Rajapinnan kuvauskielet 7
2.4 Lähdekoodin jäsentäminen ja syntaksipuu 8
2.5 Rajapinnan tarkastelu käännetystä .NET-kirjastosta 12
3 Toteutettavan työkalun esittely 14
3.1 Työkalulle asetetut vaatimukset 14
3.2 Dokumentoitavien rajapintojen esittely 15
3.3 LaTeX-dokumentaatio 16
4 Työkalun toteutus 18
4.1 Jäsentäjän toteutustapojen vertailu 19
4.1.1 Ensimmäinen vaihtoehto: oman jäsentäjän toteuttaminen 19
4.1.2 Toinen vaihtoehto: käännetyn kirjaston tutkiminen reflektiolla 20
4.1.3 Kolmas vaihtoehto: valmiin työkalun soveltaminen 20
4.1.4 Valittu toteutustapa: Roslyn-työkalun käyttäminen 21
4.2 Dokumentoitavien rakenteiden tunnistaminen koodista 22
4.3 Tietomallimoduuli 27
4.3.1 Attribuutin tietojen tallentaminen 27
4.3.2 eRA-järjestelmän rajapintoihin liittyvän tiedon tallentaminen 28
4.3.3 Dokumentaation tallentaminen 30
4.3.4 Luettelotyyppien tallentaminen 31
4.3.5 Metodien tallentaminen 32
4.3.6 Luokkien tallentaminen 33
4.3.7 Rajapinnan tietomalli 34
4.4 C#-moduuli 34
4.5 Tietomallien vertaaminen 36
4.5.1 Vertailutekniikan toteutustavan valinta 37
4.5.2 Vertailijan toiminta 38
4.6 LaTeX-moduuli 41
4.6.1 Dokumentaation jäsentäminen 41

4.6.2 Dokumentaation generointi 42
4.6.3 LaTeX-dokumentin sisäisten hyperlinkkien kirjoittaminen 43
4.7 Käyttöliittymä 44
5 eRA-järjestelmän integraatiojapinnan dokumentointi työkalun avulla 48
5.1 Attribuuttiluokkien lisääminen eRA-järjestelmän rajapintaan 48
5.2 Dokumentointityökalun toiminnan testaaminen tuottamalla vanha dokumentaatio uudelleen 49
5.3 Havainnot dokumentointityökalun käytöstä dokumentoinnissa 49
6 Jatkokehitys 50
6.1 Uusien attribuuttiluokkien muodostaminen 50
6.2 Erilaisten ohjelmointikielten jäsentäminen toteutettuun tietomalliin 51
6.3 Ohjelmiston rakenteen visualisointi 52
6.4 Ohjelmakoodin tuottaminen tietomallin avulla 53
6.5 Rajapintakomentojen testirunkojen automaattinen tuottaminen 56
6.6 Tietomallin sarjallistaminen OpenApi Documentation -kuvauskieleksi 57
6.7 Rajapintadokumentaation automaattinen toteuttaminen jatkuvassa integraatiossa 58
7 Yhteenveto 59
Lähteet 61

LYHENTEET JA MERKINNÄT

API	Application programming interface
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
CI	Continous Integration
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
COM	Component Object Model
CSV	Component Object Model
GAC	Global Assembly Cache
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JIT	Just In Time
JSON	JavaScript Object Notation
LaTeX	Ladontajärjestelmä tieteelliseen kirjoittamiseen
MVVM	Model-view-viewmodel
OAS	OpenAPI Spesification
OpenDDL	Open Data Description Language
PDF	Portable Document Format
RAML	RESTful API Modeling Language
REST	Representational State Transfer
TCP	Transfer Control Protocol
UML	Universal Modelling Language
URL	Uniform Resource Locator
WPF	Windows Presentation Foundation
XML	Extensible Markup Language
YAML	YAML Ain’t Markup Language

1 JOHDANTO

Tässä diplomityössä suunnitellaan ja toteutetaan rajapintadokumentointityökalu, jonka tarkoituksena on helpottaa ohjelmistokehittäjän työtaakkaa ohjelmointirajapintojen dokumentoinnissa. Työkalu automatisoi dokumentointiprosessia, joka ennen suoritettiin manuaalisesti. Työssä toteutettua dokumentointityökalua käytettiin Atostek OY:n eRA-järjestelmän [64] rajapinnan dokumentointiin. Työkalun tuottama integraatiorajapintaohje auttaa sekä eRA-järjestelmän kehittäjiä että eRA-järjestelmän asiakasorganisaatioita rajapinnan käyttämisessä. eRA-järjestelmä mahdollistaa potilastietojärjestelmän integroinnin kansallisen sosiaali- ja terveydenhuollon Kanta-palveluihin [28] ja Kanta-palvelujen, kuten sähköisen lääkemääräyksen, potilastiedon arkiston ja lääkärintodistusten käyttämisen. Vaikka työkalu toteutetaan ensisijaisesti eRA-järjestelmää varten, toteutuksesta pyritään tekemään laajennettava ja yleiskäyttöinen.

eRA-järjestelmä toimii potilastietojärjestelmien ja kansallisen sosiaali- ja terveydenhuollon Kanta-palvelujen välissä. eRA-järjestelmä toimii liityntäpisteenä Kansaneläkelaitoksen Kanta-palveluihin ja eRA-järjestelmän kautta voidaan tallentaa tietoa Kanta-palveluihin, sekä hakea ja esittää Kanta-palveluista noudettua tietoa. eRA-järjestelmä sisältää toiminnallisuuden Kela Kanta -palvelujen rajapintojen käyttämiseen, web-käyttöliittymän sekä web-rajapinnan kautta. Tämä web-rajapinta on integraatiorajapinta, jota dokumentoidaan osana tätä työtä.

Valmiit rajapinnan kuvauskielet (esim. Swagger [70], RESTful API Modeling Language (RAML) [60], Doxygen [75]) eivät sellaisenaan täytä kaikkia vaatimuksia, joita työkalulta vaaditaan. Niitä ovat mm. dokumentaation kansainvälistäminen, eriteltävät rajapintaosat ja -versiot, määritelty LaTeX-dokumentaation muoto sekä vaatimus siitä, että dokumentaatiotekstejä ei kirjoiteta suoraa lähdekoodiin. Näiden vaatimusten takia toteutetaan itse uusi dokumentointityökalu.

Suunniteltavan ja toteutettavan työkalun avulla voidaan helposti kirjoittaa kuvaustekstejä rajapinnan rakenteille, ylläpitää dokumentaation ja lähdekoodin vastaavuutta toisiinsa nähden sekä havainnollistaa niiden eroavaisuuksia käyttäjälle graafisessa käyttöliittymässä. Muodostettua rajapintadokumentaatiota voidaan validoida vertaamalla sitä attribuuttiluokilla merkittyyn rajapinnan lähdekoodiin. Työkalu toteuttaa oman tietomallinsa, joka sisältää dokumentoitavan eRA-järjestelmän integraatiorajapinnan metodit, luokat ja näiden sanalliset kuvaukset eri kielillä. eRA-järjestelmän viranomaisvaatimuksiin kuuluvasta lähdekoodin auditoinnista johtuen, rajapinnan rakenteiden kuvauksia ei voida kirjoittaa lähdekoodiin. Sen sijaan toteutettava työkalu kirjoittaa kuvaukset tietomalliin, jo-

ka voidaan sarjallistaa rakenteiseksi tekstiksi ja tallentaa tiedostoon. Työkalu automatisoi rajapinnan lähdekoodin lukemisen ja näyttää graafisessa käyttöliittymässä kehittäjälle listauksia ja näkymiä rajapinnan metodeista, luokkarakenteista ja pyyntöjen vastauksista. Työkalun graafisen käyttöliittymän kautta voidaan dokumentoida luokkien, metodien ja luettelotyyppien (engl. enumeration) ominaisuuksia. Työkalu tukee dokumentaation ja lähdekoodin vertailua muuntamalla nämä tietomalleiksi, joita voidaan vertailla työkalun avulla. Vertailun tulokset näytetään käyttäjälle graafisessa käyttöliittymässä. Tämän havainnollistamisen perusteella ohjelmistokehittäjän tulee päättää, muokkaako dokumentaatiota vastaamaan lähdekoodia vai lähdekoodia vastaamaan dokumentaatiota. Työkalun tuottamasta tietomallista voidaan muodostaa LaTeX-dokumentaatiota.

Työkalun käyttäjiä ovat eRA-järjestelmän ohjelmistokehittäjät. Työkalulla tuotettua dokumentaatiota eRA-integraatiorajapinnoista hyödyntävät sekä eRA-järjestelmän ohjelmistokehittäjät että eRA-järjestelmän asiakasorganisaatiot ja asiakasyritykset.

Työ sisältää rajapintaa kuvaavan työkalun arkkitehtuurin, tietomallin, graafisen käyttöliittymän, lähdekoodin jäsentäjän sekä dokumentaation jäsentäjän suunnittelun ja toteuttamisen. Lähdekirjallisuutena käytetään vertaisarvioituja tieteellisiä julkaisuja, eRA-järjestelmän ohje- ja integraatiodokumentteja, työkalun toteutuksessa käytettyjen C#-kirjastojen dokumentaatiota ja lähdekoodia sekä muiden valmiiden rajapintakuvauskielien dokumentaatiota.

Toisessa luvussa kerrotaan teoriataustaa työkalulla dokumentoitavista ohjelmointirajapinnoista sekä ohjelmointirajapintojen kuvauskielistä, tutustutaan hieman C#-kielen käännösprosessissa tapahtuvaan jäsentämiseen, abstraktiin syntaksipuuhun ja lähdekoodin tutkimiseen reflektion avulla. Kolmannessa luvussa esitellään työkalulle asetetut vaatimukset ja työkalun toimintaympäristö. Neljännessä luvussa esitellään työn tuloksena toteutettu työkalu. Viidennessä luvussa kerrotaan, kuinka työkalua testattiin eRA-järjestelmän integrointirajapintoja dokumentointoidessa. Kuudennessa luvussa pohditaan mahdollisia jatkokehityskohteita dokumentointityökalulle.

2 TEORIATAUSTA

Tässä luvussa käydään ensin läpi ohjelmointirajapinnan määritelmä sekä peruskäsitteistöä, jota liitetään ohjelmointirajapintoihin. Tämän jälkeen käydään läpi lyhyesti kolme rajapintatyyppiä, joiden dokumentointi on mahdollista toteutetulla työkalulla. Seuraavissa alaluvuissa tarkastellaan, mitä tietoja rajapinnasta tulisi dokumentoida ja minkälaisia ratkaisuja rajapintojen kuvauskielet tarjoavat dokumentointiin. Teoriaosuuden kahdessa viimeisessä alaluvussa käsitellään tekniikoita, joilla rajapintojen ohjelmallinen tutkiminen on mahdollistettu.

2.1 Ohjelmointirajapinta

Ohjelmointirajapinta (API, Application Programming Interface) on kokoelma funktioita, rakenteita ja toiminnallisuutta, joita kutsumalla asiakasohjelmisto voi suorittaa rajapinnan toteuttavan ohjelmiston toiminnallisuuksia [10]. API on siis menetelmä, jolla eri ohjelmistot voivat kommunikoida keskenään. Ohjelmointirajapintaa toteuttavaa ohjelmistoa käyttämällä ohjelmistokehittäjän ei itse tarvitse toteuttaa haluttua toiminnallisuutta, vaan kehittäjä saa käyttöönsä valmiiksi toteutetun komponentin. Usein tämä komponentti voi olla esimerkiksi ohjelmistokirjasto. Ohjelmointirajapintojen avulla ohjelmakoodista voidaan saada uudelleen käytettävää ja ohjelmistoja voidaan jakaa modulaarisiin osiin, jotka toteuttavat ohjelman toiminnan kannalta yhden loogisen kokonaisuuden.

Ohjelmointirajapintoja on toteutettu useille eri ohjelmointikielille ja tekniikoille. Rajapintoja voidaan hyödyntää monella eri tasolla, kuten laitteistotason, käyttöjärjestelmän, ohjelmointikirjaston tai web-rajapinnan tasolla. Ohjelmointirajapinnan toteutustavasta riippuen toiminnallisuutta tarjoava ohjelmisto voi olla paljastamatta toteutusta asiakasohjelmistolle, jonka tulee olla tietoinen vain julkisten rajapintafunktioiden, asetusten, luokkien ja takaisinkutsufunktioiden asettamisesta ja kutsumisesta. Ohjelmointirajapinta abstrahoi toteutettavan toiminnallisuuden niin, että käyttäjän ei tarvitse ymmärtää, miten toiminnallisuus on toteutettu rajapinnan takana. Toiminnallisuuden toteutus voi olla hyvinkin monimutkainen useiden luokkien ja funktioiden muodostama kokonaisuus, joka voi silti näyttää käyttäjälle yksinkertaiselta, yhden luokan metodilta. Rajapinnan toteuttajan perspektiivistä abstrahoinnin hyöty on siinä, että toteutus voi olla täysin erillinen siitä riippuvasta asiakasohjelmistosta, joten toteutusta voidaan muuttaa ja optimoida ilman, että rajapinnan käyttäjän tarvitsee tehdä minkäänlaisia toimenpiteitä. Monissa olio-ohjelmointiin perustuvissa kielissä kuten C#:ssa on mahdollista ohjelmoida Interface-tyyppisiä rakenteita [59].

Näiden tarkoituksena on esitellä joukko metodeja, jotka periytettävien luokkien on pakko toteuttaa. Myös rajapinnan takana toimivaa teknologiaa voidaan vaihtaa, kunhan toiminallisuus ja toiminnon mahdolliset sivuvaikutukset pysyvät samoina.

Ohjelmoitaessa staattisesti tyypitetyllä ohjelmointikielellä metodeista esitellään yleensä niiden paluuarvon tyyppi, parametreina tarvittavien muuttujien nimet, tyypin ja määreet sekä metodin määreet. Nämä kertovat kaiken tarvittavan informaation, jotta metodia voidaan kutsua. Kuitenkaan pelkästään metodin esittelyn perusteella ei voida tietää kaikkea metodin käyttäytymisestä. Esimerkiksi metodilla voi olla vaatimuksia siitä, missä tilassa järjestelmän tai olion tulee olla ennen kuin metodia voidaan kutsua onnistuneesti. Myöskään sitä, mihin tilaan järjestelmä, kirjasto tai olio jää, ei yleensä voida päätellä metodin esittelystä. Poikkeuksia, joita metodi voi laukaista joko odottamattoman toiminnan, vääränlaisen parametrin tai täyttämättömien esiehtojen takia ei myöskään useissa ohjelmointikielissä voida havaita metodin esittelystä.

Jo edellä mainittujen asioiden perusteella voidaan huomata, että ilman lähdekoodia rajapinnan toiminnallisuutta ei voida päätellä pelkistä funktioiden esittelyistä. Tämän takia rajapinta tulee dokumentoida, jotta sitä osataan käyttää oikein. On mahdollista, että web-rajapintoja käytettäessä käyttäjälle voi näkyä rajapinnasta vain joukko URL-osoitteita (Universal Resource Locator) [6], joihin on mahdollista lähettää pyyntöjä.

Seuraavissa kolmessa alaluvussa käsitellään kolmea erityyppistä rajapintatekniikkaa. Esitellyt rajapintatekniikat on valittu, koska työssä dokumentoitavaksi valitusta järjestelmästä löytyy näillä tekniikoilla toteutettuja rajapintoja.

2.1.1 XML-rajapinta

Extensible Markup Language (XML) on rakenteinen kuvauskieli, jonka avulla voidaan tallentaa tai välittää monimutkaista tietoa tekstimuodossa, joka on jäsennelty parillisten alku- ja loppumerkkien (engl. tag) avulla sisäkkäisiksi elementeiksi ja elementtien attribuuteiksi [7]. XML-rajapinnoissa komponenttien välillä kulkeva data on muunnettu XML-muotoon. XML-muotoinen data on hyödyllistä, sillä useissa ohjelmointikielissä tietorakenteita ja olioita voidaan sarjallistaa XML-muotoon ja XML-merkkijonoista voidaan rakentaa ohjelmointikielen luokkien mukaisia oliota. XML ei ole riippuvainen ohjelmointikielestä ja XML-syntaksilla kirjoitettuhin dokumentteihin perustuva tiedonsiirto mahdollistaa sen, että asiakasohjelmisto sekä palvelinohjelmisto voivat olla toteutettu eri ohjelmointikielillä ja tekniikoilla.

Osa web-rajapinnoista on asiakas-palvelin-arkkitehtuurin mukaisia palvelimen toteuttamia rajapintoja, joihin asiakasohjelmiston on mahdollista yhdistyä. Useissa web-rajapinnoissa käytetään Transfer Control Protocol (TCP) -yhteyden päällä toimivaa Hypertext Transfer Protocol (HTTP) -protokollaa, jossa asiakasohjelmisto avaa TCP-yhteyden palvelimelle ja lähettää HTTP-protokollan mukaisia pyyntöjä, joita palvelin suorittaa [16]. HTTP-protokollan pyyntöjä ovat esimerkiksi GET-, POST-, PUT- sekä DELETE-pyynnöt.

HTTP-protokollaan kuuluu, että pyynnöt erotetaan toisistaan URL-osoitteilla ja usein palvelimen toiminnot on järjestetty URL-osoitteiden mukaan niin, että yhtä URL-osoitetta ja HTTP-metodia kohden on yksi ennalta määritelty suoritettava palveluprosessi. Näin ei kuitenkaan aina ole, ja suoritettava prosessi voi riippua pyynnön rungossa kulkevasta hyötykuormasta. Usein pyynnön rungossa mukana kulkeva data on rakenteista ja datassa voidaan välittää mukana tieto siitä, missä muodossa rakenteellinen data on. Muita rakenteisia kuvauskieliä XML-kielen lisäksi ovat esimerkiksi JSON [14], YAML [5], CSV sekä OpenDDL [63].

Dokumentoitavassa järjestelmässä yksi rajapinnoista on Representional State Transfer (REST) -mallia mukaileva rajapinta, joka käyttää rakenteisena datana XML-dokumentteja. REST-mallin mukaisesti palvelupyynnöt erotellaan toisistaan URL-osoitteiden avulla, jotka kuvaavat dataresursseja palvelimella. Riippuen pyynnöstä asiakas voi pyytää itselleen resurssin, joka palautetaan pelvelimen vastauksessa tai lähettää palvelimelle uuden resurssin tallennettavaksi. Kaikki tarvittava data pyynnön tilasta, kuten istuntoja yksilöivät tunnisteet tai numerot, tulee lähettää pyynnön mukana. [17]

Tässä työssä dokumentoitavan järjestelmän rajapinta ei kuitenkaan noudata puhtaasti REST-mallia, sillä esimerkiksi resursseja haetaan HTTP POST -pyynnöillä, kun REST-mallissa suositellaan käyttämään tällöin GET-pyyntöjä. Tämän takia tästä rajapinnasta käytetään jatkossakin nimitystä XML-rajapinta.

2.1.2 .Net-rajapinta

.NET-rajapinnalla tarkoitetaan tässä diplomityössä .NET-kehyksellä ohjelmoitua ohjelmistokirjaston rajapintaa. .NET on Microsoftin kehittämä komponenttikirjasto, joka tukee useita eri ohjelmointikieliä, kuten C#:a ja Visual Basicia [35]. .NET-ympäristössä Common Language Infrastructure (CLI) -mallin ansioista eri .NET-kehitysympäristöön kuuluvat ohjelmointikielet käännetään yhteiseksi välikielikoodiksi (CIL, Common Intermediate Language). CIL on symbolinen konekieli, joka on riippumaton prosessorityypistä ja laitteistosta. Ajettaessa .NET-kehyksen CIL-koodia, sitä suoritetaan Common Language Runtime (CLR) -virtuaalikoneen avulla. CLR-virtuaalikoneeseen kuuluva ajonaikainen Just In Time (JIT) -kääntäjä kääntää ohjelman suorituksen aikana CIL-koodin laitekohtaiseksi konekieleksi (engl. native code), jota laitteiston prosessori osaa suorittaa. Kaikista .NET-kehyksen tukemista ohjelmointikielistä voidaan muodostaa CIL-pohjaisia dynaamisesti linkitettyjä kirjastoja, joita voidaan suorittaa kaikissa .NET-ympäristön ohjelmointikielillä ohjelmoiduissa sovelluksissa. [12]

Dynaamisesti linkitettävät kirjastot ovat jaettavia kirjastoja, joita useat ohjelmat voivat käyttää samaan aikaan. .NET-ympäristössä muodostetaan käännöksiä (engl. assembly), jotka sisältävät kaiken kirjaston käyttämiseen tarvittavan tiedon yhdessä tai useammassa tiedostossa, jotka voidaan tallentaa kovalevylle. Kirjaston käännös voi sisältää neljää erilaista tietotyyppiä: käännöksen CIL-koodia, tyyppien metadataa, käännöksen manifestin ja mahdolliset muut resurssit, kuten kuvat [58]. Koska kirjasto sisältää käännöksen mani-

festin sekä tyyppien metadatainformaation, ajoympäristölle on selvää sidonnan jälkeen, kuinka kirjastoa voidaan käyttää ja mitä tyyppejä kirjasto sisältää. Dynaamisia kirjastoja voidaan kutsua dynaamisen sidonnan jälkeen näiden tarjoaman rajapinnan kautta aivan kuin ne olisivat määritelty paikallisesti niitä kutsuvasta koodista. .NET-ympäristössä dynaamisia kirjastoja tallennetaan yleensä .dll-päätteisisiin tiedostoihin. [56]

Käännöksen manifesti on XML-tiedosto, joka sisältää muun muassa tiedon tiedostoista, jotka muodostavat käännöksen, listauksen tyyppiviitteistä ja tiedon siitä, mistä näiden tyyppien esittely ja määrittely löytyy. Manifestissa on myös lista muista käännöksistä, joista se on itse riippuvainen [33]. .Net-ympäristössä käännöksiä ei tunnisteta niiden tiedostonimien perusteella, vaan niiden käännösnimen perusteella. Tämä tunniste koostuu käännöksen näytettävästä nimestä, versiosta ja julkisesta avaimesta. .Net-ympäristössä voidaan yleensä asettaa tietty tiedostopolku, josta käännöksiä ladataan ajon aikana. CLR-ympäristössä Global Assembly Cache (GAC) on välimuisti, mistä käännöksiä etsitään ajonaikana ensimmäisenä [9].

2.1.3 ActiveX-rajapinta

ActiveX on Microsoftin kehittämä Windows-ympäristöön suunniteltu viestinvälitysprotokolla [30]. Tämän diplomityön kannalta tarkasteltava ActiveX-tekniikalla muodostettu rajapinta on toteutettu käärekirjastoksi .NET-ympäristössä toteutetulle rajapinnalle. Työssä mainittu ActiveX-rajapinnan toteutus on itsessään .NET-luokkakirjasto, joka on muodostettu niin, että se toteuttaa ActiveX-tekniikan vaatimat ominaisuudet ja rajoitteet. Kaikki rajapinnassa toteutetut rakenteet tulee rekisteröidä Component Object Model (COM) -standardin [34] mukaan tunnisteilla, joiden avulla näitä voidaan kutsua toisesta ajoympäristöstä, kuin missä ne ovat luotu. .NET-ympäristössä voidaan luoda esimerkiksi C#-kielellä kirjoitetusta ohjelmointikirjastosta System.EnterpriseServices-nimiavaruudesta löytyvillä toiminnallisuuksilla käärekirjasto, joka toteuttaa ActiveX-tekniikan vaatimukset [29]. Koska dokumentoitava ActiveX-rajapinta on toteutettu .NET-kirjastona, on sen käsittely tämän diplomityön kannalta hyvin samanlaista kuin edellisessä kappaleessa kuvatun .NET-rajapinnan käsittely.

2.2 Rajapinnan dokumentointi

Rajapinnan oikeanlaiseen käyttämiseen tarvitaan lähes aina dokumentaatiota. Usein lähdekoodia ei ole saatavilla käyttäjälle ja silloinkin kun lähdekoodi on saatavilla, sen tulkinta voi olla aikaa vievää ja tarpeetonta. Usein rajapinnan käyttäjä ei ole kiinnostunut miten toteutus on tehty, vaan siitä, miten rajapintaa tulee käyttää.

Uuden rajapinnan ymmärtämisessä tärkeimpiä tekijöitä on rajapinnan dokumentaation laatu [66]. Kuitenkin vuonna 2014 tehdyssä tutkimuksessa [15] on havaittu, että myös ohjelmointikielen tyyppijärjestelmällä on vaikutusta siihen, kuinka helppoa rajapinta on ottaa käyttöön. Staattisesti tyypitetyllä ohjelmointikielellä muodostetun rajapinnan avulla tutki-

mushenkilöt saivat toteutettu nopeammin tutkimuksessa vaaditut tehtävät kuin dynaamisesti tyypitetyllä rajapinnalla. [15] Työssä dokumentoitava rajapinta on toteutettu staattisesti tyypitetyllä ohjelmointikielellä, joten lähdekoodissa on valmiina määritelty muuttujien tyypit, mikä helpottaa dokumentaation muodostamista.

Dokumentaatiotapoja on monenlaisia. Perinteisesti rajapinnan dokumentaatiota on kirjoitettu tekstidokumenttiin, jossa on listattu kaikki metodit ja rakenteet, sekä kirjoitettu sanalliset kuvaukset. Toinen dokumentaatiotapa on esimerkkimuotoinen dokumentaatio, jossa esitetään konkreettisesti, kuinka jokin pieni toiminnallisuus toteutetaan käyttämällä yhtä tai useammista rajapinnan ominaisuuksista. Joskus tällä tavoin dokumentoiduissa kirjastoissa voidaan julkaista esimerkkiprojekteja, joiden lähdekoodin kirjaston käyttäjä voi ladata tietokoneellensa. Kolmantena tapana dokumentoida rajapintaa ovat keskustelupalstat. Suosittujen avoimien rajapintojen käyttäjät saattavat hakea tietoa rajapinnan käytöstä internetin keskustelupalstoilta, joissa käyttäjät ovat kyselleet rajapintojen toiminnasta ja toiset käyttäjät ovat vastanneet. Tämä dokumentaatiotapa ei välttämättä ole aina suunniteltua ja koska vastaajat voivat olla eri henkilöitä kuin rajapinnan kehittäjät, myös virheellisten vastausten mahdollisuus on suuri. Neljäntenä myös interaktiivisia dokumentaatioita on olemassa, joissa käyttäjä voi kutsua rajapinnan metodeja web-palvelun kautta ja nähdä miten rajapinnan metodit toimivat käytännössä. Yksi tämänkaltainen palvelu on SwaggerHub [68].

Dokumentaation muodostaminen ilman erityisiä dokumentointityökaluja on yleensä rajapinnan kehittäjän manuaalisesti tekemää työtä, jossa kirjoitetaan ulkoiselle dokumentille kuvauksia rajapinnan rakenteista ja metodeista. On olemassa dokumentointityökaluja, joiden avulla pyritään vähentämään manuaalisen työn määrää tai parantamaan dokumentaation laatua. Seuraavassa aliluvussa tutustutaan lyhyesti muutamaan olemassa olevaan rajapinnan kuvauskieleen, joiden käyttöä harkittiin työn toteutusvaiheessa.

2.3 Rajapinnan kuvauskielet

Ohjelmistojen dokumentointiin on kehitetty useita dokumentointityökaluja. Työkaluja on kehitetty yleskäyttöisiksi koko ohjelmiston rakenteen dokumentointia varten, sekä esimerkiksi erityisesti REST-tyyppisten ohjelmointirajapintojen dokumentointiin.

Dokumentointityökalujen toiminta voi perustua rajapinnan lähdekoodin hyödyntämiseen dokumentaation muodostamisessa. Tällöin dokumentointityökalun ohjelmointikielen tai dokumentointiin käytetyn dokumentaatiosyntaksin jäsentäjä lukee lähdekooditiedoston rakenteet ja muodostaa siitä rajapinnan tietomallin rajapinnan kuvauskielellä. Dokumentaatiotekstit tulee tällöin kirjoittaa suoraan lähdekoodiin, dokumentointiin tarkoitetulla syntaksilla.

Doxygen on yleiskäyttöinen dokumentointityökalu, jota käytettäessä lähdekoodiin kirjoitetaan kommentteja erityisellä syntaksilla, jonka työkalu lukee ja tuottaa HTML- (Hypertext Markup Language) tai LaTeX-muotoista dokumentaatiota. Sen avulla voidaan dokumentoida useita eri ohjelmointikieliä kuten C++:aa, Javaa, Pythonia, C#:a ja Fortrania. [75]

Suosittu dokumentointityökalu REST-tyyppisten rajapintojen dokumentointiin on Swagger [70]. Swaggerin avulla voidaan muodostaa dokumentaatiota sen rajapinnan kuvauskielen avulla. OpenAPI Spesification (OAS) -kuvauskieli [72] on Swagger-dokumentointityökalun käyttämää JSON- tai YAML-muotoon kirjoitettua merkintäkieltä, jolla kuvataan rajapinnan rakenne. OAS:sta on pyritty muodostamaan standardia REST-tyyppisten rajapintojen kuvaamiseen. Kuvauskielestä voidaan muodostaa dokumentaatiota tai rajapinnan toteutus tuottamalla ohjelmallisesti kuvauskielestä haluttua ohjelmointikieltä. OAS-kuvauskieli tukee useita eri ohjelmointikieliä avoimen lähdekoodin yhteisön kirjoittamien työkalujen ansiosta. Tuettuja kieliä ovat esimerkiksi C#, Python, Java, Clojure sekä JavaScript. [70]

Swagger Inspector -työkalulla on mahdollista muodostaa dokumentaatiota kutsumalla rajapinnan URL-osoitteita. Tällöin dokumentoinnissa ei käytetä apuna lähdekoodia, vaan rajapinnan toteutuksen toimintaa. Swagger Inspector tallentaa HTTP-kutsut ja HTTP-vastaukset, joita Swagger Inspector -työkalun kautta on kulkenut. Kutsuista muodostetaan OpenAPI Spesification -malli ja sen avulla voidaan näyttää rajapinnan interaktiivinen dokumentaatio Swaggerhub-palvelussa. Työkalu lisää dokumentaatioon kaikki kutsutut URL-osoitteet ja dokumentaation kautta voidaan kutsua uudestaan näitä pyyntöjä ja nähdä vastaukset. [67]

RESTful API Modeling Language (RAML) on toinen web-rajapintojen kuvauskieli. RAML toimii käyttäjän näkökulmasta hyvin samankaltaisesti kuin Swagger. Kuitenkin RAML on keskittynyt enemmän rajapinnan suunnitteluun ja toteuttamiseen mallin pohjalta kuin valmiin rajapinnan dokumentointiin, sillä RAML:n avulla ei voida muuntaa lähdekoodia kuvauskieleksi. Tätä kuvauskieltä kannattaakin käyttää rajapinnan suunnitteluvaiheessa. [60]

Rajapintojen toteutustapojen sekä rajapinnan dokumentaatiolle asetettujen vaatimusten takia dokumentointia ei suoriteta tässä työssä käyttäen mitään valmista rajapinnan dokumentointityökalua.

2.4 Lähdekoodin jäsentäminen ja syntaksipuu

Dokumentointityökalu toteutettiin Microsoftin .Net-ympäristön tukemalla C#-kielellä, sillä dokumentoitava rajapinta oli kirjoitettu C#-ohjelmointikielellä. Lähdekoodia voidaan tarkastella eri tasoilla, joko syntaktisella tasolla tai semanttisella tasolla, joista kumpaakin käytetään työkalussa toteutetussa jäsentäjässä. Tässä aliluvussa tarkastellaan C#-kääntäjän toimintaa, koska dokumentointityökalun toteutuksessa on käytetty C#-kääntäjän toimintaan perustuvaa ohjelmistokirjastoa, jonka avulla on analysoitu lähdekoodia. Muiden käännettävien ohjelmointikielten kääntäjät toimivat pääpiirteittäin samalla tavalla.

C#-kielen kääntäminen tapahtuu peräkkäisissä vaiheissa [13]. Ensin lähdekooditiedostot muutetaan Unicode-merkkijonoiksi. Tämän jälkeen Unicode-merkkijonoille suoritetaan jäsentäminen (engl. parsing). Viimeisenä vaiheena jäsennykseen kuuluvan leksikaalisen analyysivaiheen tuloksille toteutetaan syntaktinen analyysi, jonka tuloksena saadaan CIL-

koodia. [31]

C#-kääntäjän jäsentäjä on tyypiltään yleinen vasemmalta oikealle jäsentäjä (engl. Generalized Left to Right Parser) [13]. Jäsentäjän toiminnan tavoitteena on muodostaa lähdekoodin perusteella abstrakti syntaksipuu, joka on hierarkkinen rakenne koodin suorittamisen kannalta oleellista tietoa [11].

Selaaja (engl. scanner) pilkkoo ensin lähdekoodin lekseemeihin (engl. lexeme) leksikaalikieliopin avulla. Selaajan avulla muodostetaan lekseemejä vastaavia leksikaalielementtejä (engl. token). Lekseemit ovat merkkijonoja lähdekoodissa, joista jäsentäjän tulee päätellä niiden merkitys. Leksikaalielementit kuvaavat lekseemien merkitystä, esimerkiksi lekseemeistä ”muuttujaABC” ja ”foo” jäsentäjän tulee päätellä, että nämä ovat molemmat tunnisteita (engl. identifier). [2]

C#-kielelle on viisi erilaista lekseemityyppiä: rivinvaihto, tyhjämerkki (engl. white space), kommentti, leksikaalielementti ja esikääntäjän ohje (engl. preprocessing directive)[31]. Nämä edellä mainitut joukot muodostavat C#-kielen äärelliset lekseemijoukot. Kieli ei tunne muita merkkejä näiden merkkien ulkopuolelta. Joukot ovat myös keskenään erillisiä joukkoja, eli joukoista ei voi löytyä yhteisiä alkioita. C#-ohjelmointikielen leksikaalielementit ovat joko tunnisteita (engl. identifier), avainsanoja (engl. keyword), literaaleja (engl. literal), operaattoreita (engl. operator) tai välimerkkejä (engl. punctuator) [31].

Tunnisteilla merkitään ohjelmointikielen rakenteiden nimiä kuten, muuttujien, luokkien ja metodien nimet. Avainsanat ovat ohjelmointikielten rakenteiden muodostamiseen varattuja merkkijonoja, C#-ohjelmointikielessä avainsanoja ovat esimerkiksi ”continue”, ”class”, ”new”, ”return” ja ”public” [23]. Literaalit ovat merkkijonoja, joilla on kirjoitettu muuttujan arvo lähdekoodiin. Literaaleja ovat totuusarvot, kokonaisluvut, liukuluvut, merkit (engl. characters), merkkijonot sekä ”null” [31]. Operaattoreilla merkitään kielen rakenteisiin kohdistuvaa toimintaa tai arvojen muokkausta. Näitä ovat esimerkiksi ”+”, ”-”, ”=”, ja ”[]” [24].

Lekseemityypeistä vain leksikaalielementit ovat hyödyllisiä jäsennyksessä ohjelmiston toiminnan kannalta, joten vain leksikaalielementit lisätään abstraktiin syntaksipuuhun. Kommentit, rivinvaihdot ja tyhjämerkit ohitetaan abstraktin syntaksipuun kokoamisvaiheessa. Esikääntäjän ohjemerkit käsitellään C#-kääntäjässä samassa vaiheessa kuin selaus suoritetaan, eikä varsinaista esikääntäjävaihetta ole. C#-esikääntäjän merkit alkavat aina #-merkillä. Näillä voidaan esimerkiksi määritellä ehdollisia osia, joita kääntäjä voi huomioida tai olla huomioimatta. #Pragma-määrittelyillä voidaan kertoa kääntäjälle lisätietoa, kuitenkaan muuttamatta ohjelman semanttista rakennetta. Nämä esikääntäjän merkit tulee tulkita, ennen kuin abstraktia syntaksipuuta voidaan muodostaa. [31]

Leksikaalielementeistä muodostetaan abstrakti syntaksipuu (AST, engl. Abstract Syntax Tree). AST on puumainen tietorakenne, joka sisältää hierarkkisia solmurakenteita, jotka ovat linkitetty yhteen. AST kuvaa koodin semanttista rakennetta. Jäsennyspuun ja syntaksipuun erona on yksinkertaistetusti se, että abstraktista syntaksipuusta on siivottu pois solmut, jotka eivät merkitse ohjelman suorituksen kannalta mitään.

Ohjelmointikielten kieliopeissa on kahdenlaisia symboleita, muuttujasymboleita (engl.

nonterminal symbol) ja päätesymboleita (engl. terminal symbol). Kieliopin avulla esitetään sääntöjä, joilla voidaan muuntaa päätesymboleita muuttujasymboleiksi sekä edelleen korkeamman tason muuttujasymboleiksi. Kääntäjiä on kahta eri päätyyppiä, kokoavia kääntäjiä sekä rekursiivisia kääntäjiä. C#-ohjelmointikielen kääntäjä on kokoava kääntäjä [13]. Kokoavat kääntäjät rakentavat syntaksipuuta alhaalta ylöspäin. Kokoava kääntäjä kerää selaajalta saatuja päätesymboleita jäsennyspinoon ja muuntaa symbolijonoja kielen sääntöjen mukaan muuttujasymboleiksi, jolla korvataan jäsennyspinoon laitetut alemman tason symboleja. Syntaksipuun kokoamista jatketaan niin kauan kun syöte on tyhjä ja on muodostettu aloitussymboli, jolloin syntaksipuu on rakennettu onnistuneesti. Kokoava jäsentäjä tunnistaa ensin kielen matalan tason rakenteet ja kokoaa näistä korkeamman tason rakenteita. Kokoavan jäsentäjä tuottaa pieniä alipuita aina muuntaessaan symboleja korkeamman tason symboleiksi. Alipuista kootaan näin koko syötettä kuvaava syntaksipuu. Valmiin abstraktin syntaksipuun lehtisolmut ovat päätesymboleita ja muut solmut muuttujasymboleita. [2]

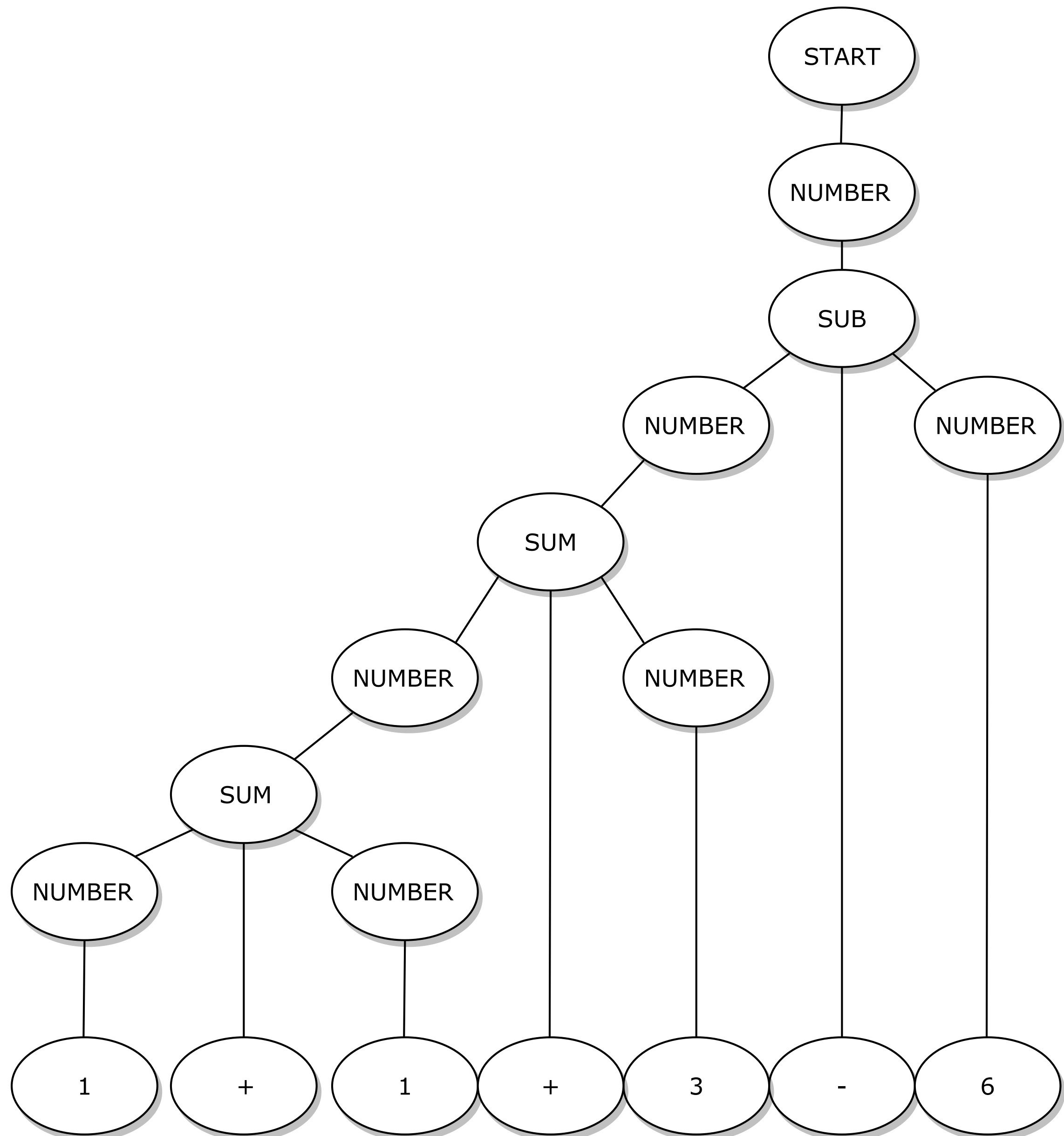
Seuraavaksi esimerkki hyvin yksinkertaisesta Backus-Naur Form (BNF) [25] -muotoon kirjoitetusta kieliopista, jolla voidaan muodostaa vain yhteenlaskuja sekä vähennyslaskuja kokonaisluvuilla. Kielen kielioppiin kuuluu neljä sääntöä, jotka on listattu alla:

1. <START> ::= < NUMBER>
2. <NUMBER> ::= <SUM> | < SUB >|0|1|2|3|4|5|6|7|8|9
3. <SUM> ::= <NUMBER> + <NUMBER>
4. <SUB> ::= <NUMBER> – <NUMBER>

Esimerkiksi lause 1+1+3-6 on mahdollinen kirjoittaa kieliopin mukaan ja lause voidaan jäsentää syntaksipuuksi. Mikäli lause ei noudata kieliopin sääntöjä, ei jäsentäminen ole mahdollista. Mikäli kielioppi tukisi esimerkiksi tyhjämerkkejä tai kommentteja, tulisivat nekin näkyville omina solmuinaan syntaksipuun lehdiksi. Kuvassa 2.1 esitetty jäsennyspuu ei ole ainoa mahdollinen jäsennyspuu, jolla voitaisiin kuvata laskulausekkeen arvoa, vaan jäsentäjän algoritmin tyypistä riippuen voidaan muodostaa myös muunlaisia puita, joilla voidaan kuvata lausekkeen arvo.

Operaattorien assosiatiivisuus vaikuttaa siihen miten edellä mainittu lause voidaan jäsentää syntaksipuuksi. Yhteenlaskujen laskujärjestyksellä ei ole väliä, mutta ne voidaan silti laskea eri järjestyksessä, esimerkiksi (1+1)+3-6 tai 1+(1+3)-6. Näistä molemmat tuottavat saman lopputuloksen, mutta laskujärjestys on erilainen. Vähennyslaskujen laskujärjestyksellä taas on väliä ja muuttamalla laskujärjestystä voidaan päästä eri lopputulokseen. Esimerkiksi lauseke 1-1-3 voidaan laskea (1-1)-3 = -3 tai 1-(1-3) = 3. Yhteen- ja vähennyslaskujen sanotaan olevan vasemmalle päin assosiatiivisia, sillä jos symbolin kummallakin puolella on yhteenlaskuoperaattori, symboli tulkitaan kuuluvaksi vasemmalla puolella olevalle operaattorille. Esitellyn kieliopin säännöillä voidaan laskea vain yhteen- ja vähennyslaskuja. Näiden operaatioiden prioriteetti on sama. Prioriteetilla ratkaistaan samalla tasolla olevien operaatioden laskujärjestys. Jos esimerkin kielioppiin lisättäisiin säännöt esimerkiksi kertolaskujen laskemiselle, tulisi kertolaskun prioriteetti olla

korkeampi kuin yhteenlaskun prioriteetti, jotta tuloksesta saataisiin oikea. [2]



Kuva 2.1. *Jäsennyspuu yksinkertaisesta laskulausekkeesta.*

Abstraktin syntaksipuun muodostamisen jälkeen kääntäjä käyttää vierailijaluokkia (engl. visitor), jotka käyvät läpi syntaksipuun solmut ja suorittavat operaatioita riippuen vierailijan tyypistä. Yksi tiedosto voidaan kääntää lcsc-kääntäjän (Local C Sharp Compiler) avulla kutsuen komentorivillä kääntäjää seuraavin asetuksin:

```
lcsc -visitor.bind -visitor:typecheck
-visitor:rewrite -visitor:ilgen <tiedoston nimi>
```

Kääntäjä jäsentää tiedoston ja bind-vierailija kerää solmuista symbolit ja lisää ne symbolitauluun. Symbolitauluun kerätään kaikki lähdekoodissa määritellyt muuttujien nimet, tietotyypit, funktioiden nimet, luokat jne. Symbolitauluun tallennetuista tiedoista käy ilmi

muuttujan nimi, tyyppi, sille annetut määreet sekä muuttujan näkyvyysalue. Typecheck-vierailija suorittaa staattisen tyyppitarkastuksen muuttujille käyttäen edellisessä vaiheessa muodostettua symbolitaulua. Muuttujan nimellä haetaan symbolitaulusta muuttujan tyyppi ja päätellään tyyppisääntöjen perusteella onko tyyppiä käytetty oikein. Typecheck-vierailija tarkastaa esimerkiksi annetaanko funktiokutsulle oikea määrä parametreja ja ovatko annetut parametrit oikean tyyppisiä. Rewrite-vierailija voi muuttaa joitain syntaksipuun solmujen rakenteita välikielikoodin muodostamisen helpottamiseksi. Viimeinen Ilgen-vierailija tuottaa välikielikoodia abstraktin syntaksipuun pohjalta. [13]

2.5 Rajapinnan tarkastelu käännetystä .NET-kirjastosta

.NET-kehyksen CLI-ympäristössä ohjelman suorittama ohjelmakoodi sijaitsee omassa sovellustoimialueessaan (engl. application domain), joka on eristetty muista mahdollisesti samanaikaisesti suoritettavista sovelluksista turvallisuussyistä. Sovellustoimialue toimii eri tasolla kuin samankaltaiset käyttöjärjestelmän hallinnoimat prosessitoimialueet, joilla varataan omat muistialueensa kullekin prosessille. Jotta ohjelma voisi käyttää ulkoista kirjastoa, täytyy kirjaston käännös ladata ohjelman omaan toimialueeseen. Kaikki koodi, jota ohjelma voi käyttää, sijaitsee sen ajonaikaisessa toimialueessa. [27]

Reflektio on menetelmä, jolla voidaan tutkia käännöksen tyyppien ominaisuuksia. Reflektiota voidaan käyttää käännöksen tutkimiseen tai itse ajossa olevan ohjelmiston oman käännöksen tyyppien tutkimiseen, sillä .NET-ympäristössä kaikki ajettava koodi kääntyy käännöksiksi, jotka sisältävät metatietoa tyypeistä. Kaikki .NET-ympäristössä tuotettu ja käännetty koodi on hallittua koodia (engl. managed code), joka kääntyy virtuaalikoneen päällä ajettavaksi CLI-koodiksi. [4]

Mscorlib.dll-käännöksen System.Reflection-nimiavaruudesta [52] löytyvillä toiminnallisuuksilla voidaan löytää ja tutkia käännöksestä löytyvien tyyppien tietoa ajonaikaisesti. Reflektiometodit hakevat tiedon halutusta tyypistä käännöksen metadatasta, joka sijaitsee jokaisessa .NET-käännöksessä. Ensin reflektiolla haetaan tieto tyypistä, josta ollaan kiinnostuneita. Tässä tapauksessa halutaan kutsua metodeja, jotka palauttavat System.Type-luokan [54]. System.Type-luokan instanssin voi saada käsiteltäväksi esimerkiksi kutsumalla jo rakennetun olion GetType-metodia. Jos oliota ei ole luotu ajonaikaisessa koodissa, voidaan System.Type-luokan instanssi saada luotua myös System.Type.GetType-metodilla, jota kutsuakseen täytyy antaa metodille parametrina käännöksen nimi, jossa haluttu tyyppi sijaitsee sekä halutun tyypin nimi. Kun System.Type-luokan instanssi on luotu, voidaan siltä hakea tietoa luokan tarjoamin metodein. System.Type-luokalta voidaan hakea tietoa esimerkiksi sen rakentajista, kentistä, rajapinnoista, jäsenistä sekä ominaisuuksista. Kaikki tieto, mitä tyypiltä voidaan kysyä reflektion avulla, tulee löytyä käännöksen metadatasta. [73]

.NET-ympäristössä voidaan lisätä käännöksen metadataan lisää tietoa käyttämällä attribuuttiluokkia. Attribuuttiluokkia voidaan periyttää System.Attribute-luokasta [51] ja muodostaa näin mukautettuja attribuutteja, joille voidaan antaa parametreja niiden rakentajan

kautta. Parametrina annettavien arvojen tulee olla yksinkertaisia tyyppejä, luettelotyyppejä tai taulukoita, jotta niiden arvot voidaan tallentaa metadataan. Attribuutteja voidaan antaa esimerkiksi käännöksille, luokille, luettelotyypeille, kentille, rajapinnoille, metodeille, ominaisuuksille tai paluuarvoille. Reflektiolla voidaan hakea tietoa myös attribuuteista käyttämällä System.Type-luokan getCustomAttributes-pyyntöä. Tutkimalla attribuutteihin tallennettua tietoa reflektion avulla voi luokkaa tutkiva koodi lukea lisätietoa rakenteesta ja toimia sen mukaan. [73]

3 TOTEUTETTAVAN TYÖKALUN ESITTELY

Luvussa esitellään työkalulle asetetut vaatimukset sekä kuvataan ympäristöä, jossa työkalun tulee toimia. Työssä dokumentoitiin eRA-järjestelmän integraatiorajapinta toteutetulla työkalulla, joten aliluvussa 3.1 esiteltävät dokumentointityökalun toiminnan vaatimukset perustuvat eRA-järjestelmän dokumentointiin.

3.1 Työkalulle asetetut vaatimukset

Toteutettava työkalu on tarkoitettu rajapintojen dokumentoimiseen. Työkalun tulee nopeuttaa ja helpottaa dokumentointiprosessia ja osittain automatisoida dokumentaation luominen. Tavoitteena on varmistaa toteutuksen ja dokumentaation vastaavuus toisiinsa nähden. Vastaavuus tulee varmistaa vertailemalla lähdekoodia ja kirjoitettua dokumentaatiota, joka on toteutettu ennalta määritellyllä LaTeX-syntaksillaan. Vertailtavia ominaisuuksia ovat rajapinnan tarjoamat luokkarakenteet, luettelotyypit ja funktiot. Vertailun tuloksia näytetään dokumentointityökaluun toteutettavassa käyttöliittymässä.

Työkalun tulee osata tunnistaa tarpeelliset rakenteet ja metodit lähdekoodista. Rakenteiden lisäämistä tietomalliin tulee voida rajoittaa esimerkiksi projektin tai lähdekooditiedoston perusteella. Tunnistetuille rakenteille on mahdollista kirjoittaa sanallisia kuvauksia työkalun käyttöliittymässä usealla eri luonnollisella kielellä, mutta kuvauksia ei saa tallentaa lähdekoodiin, sillä rajapinnan muuttuessa sille on tehtävä viranomaisen suorittama auditointi. Rajapinnasta on voitava tehdä tietomalli, johon rajapinnan rakenteet ja dokumentaatio voidaan tallentaa tarvittaessa levylle. Työkalun tulee tukea ja osata eritellä käyttöliittymässä eri tekniikoilla toteutetut rajapintaosat, joita on tähän mennessä toteutettu kolme: Web-rajapintana XML-rakenteisia dokumentteja käsittelevä rajapinta ja rajapinnan asiakasjärjestelmän puolelle integroitavat käärerajapinnat, .NET-rajapinta sekä ActiveX-tekniikkaa käyttävä rajapinta. Eri rajapintatekniikoiden lisäksi työkalun tulee tukea ja tunnistaa lähdekoodista eri rajapintaversioita, jotka voivat sisältää erilaisia rakenteita ja metodeja. Työkalun on myös tunnistettava eri rajapintaosuuksiin liittyviä rakenteita lähdekoodista.

Näiden kategorioiden tunnistamisen lisäksi työkalun tulee voida generoida määritellyn muotoista LaTeX-dokumentaatiota työkalussa valituille kategorioille eli rajapintaversiolle, rajapintaosille ja rajapintatekniikalle. LaTeX on valittu dokumentaatiotavaksi, sillä edellisten eRA-järjestelmän integraatiorajapinnan versioiden dokumentaatio on kirjoitettu LaTeXilla ja LaTeXia on yksinkertaista muokata ohjelmallisesti. Valmis dokumentaatio

käännetään LaTeX-kuvauskielisesta dokumentista Portable Document Format (PDF) -tiedostoksi [1], johon kuvauskieleen kirjoitetut tekstit muutetaan niille merkittyjen tyyliasetusten mukaiseen esitysmuotoon.

Työkalu tulee suunnitella siten, että se on laajennettavissa tukemaan myös eri ohjelmointikielellä toteutettua rajapintaa. Mahdollisia tulevia tutkimuskohteita ovat myös lähdekoodin generointi työkalun käyttöliittymässä niin, että työkalua voitaisiin käyttää rajapinnan suunnitteluun kirjoittamalla dokumentaatio ensin.

3.2 Dokumentoitavien rajapintojen esittely

Dokumentoitavat rajapinnat kuuluvat kaikki samaan eRA-järjestelmään. Järjestelmää käytettäessä asiakasorganisaatioiden järjestelmät integroituvat rajapinnan kautta eRA-järjestelmään. Kunkin asiakkaan tulee valita käyttöönsä joko XML-muotoista dataa välittävä REST-tyyppinen rajapinta, .NET-rajapinta tai ActiveX-rajapinta, joista kaikilla voidaan saavuttaa sama toiminnallisuus. XML-rajapinta toimii .NET- ja ActiveX-rajapintojen taustalla ja nämä käärerajapinnat kutsuvat operaatioissaan XML-rajapintaa. Rajapintoja on tehty useita, sillä Windows-ympäristössä toimivien asiakasohjelmistojen integroituminen on helpompaa .NET- tai ActiveX-rajapintoihin, jos asiakasohjelmistot tukevat valmiiksi näitä tekniikoita. Integroiduttaessa XML-rajapintaan asiakasohjelmiston tulee itse toteuttaa palvelua kutsuvat metodit, jotka osaavat sarjallistaa ja lukea XML-muotoiset luokkarakenteet, joita REST-rajapinnan kautta lähetetään. .NET- ja ActiveX-rajapintojen tapauksissa asiakkaat lataavat käärekirjaston, joka helpottaa rajapinnan kutsumista esimerkiksi ketjuttamalla XML-rajapintaan lähetettäviä pyyntöjä sekä toteuttamalla valmiiksi tarvittavat tietorakenteet, jotka rakennetaan vastauksen perusteella.

Rajapinta on jaettu dokumentaatiossa eri osiin toiminnallisuuden perusteella. Asiakkaan ei välttämättä tarvitse ottaa käyttöön kaikkia osia tarpeistaan riippuen, joten asiakkaan ei tässä tapauksessa tarvitse kirjoittaa toteutusta kaikille metodeille. Metodeille on ilmoitettu dokumentaatiossa niiden toteuttamisen pakollisuus ja muut mahdolliset ehdolliset pakollisuudet, kuten se tarvitseeko pyyntö avoimen istunnon. Rajapinnasta julkaistaan uusia versioita sitä mukaa, kun eRA-järjestelmään toteutetaan uusia toiminnallisuuksia. Uusissa versioissa yleensä kutsuttavien pyyntöjen määrä lisääntyy, mutta myös vanhojen pyyntöjen toiminnallisuus voi muuttua tai ominaisuuksia voi poistua uusissa versioissa. Usein myös jo määriteltyihin luokkarakenteisiin tulee uusia ominaisuuksia ja luettelotyyppeihin lisätään uusia jäseniä. Tämän takia aina julkaistaessa uusi rajapintaversio julkaistaan myös uusi dokumentaatio, johon kirjataan kaikki uuteen rajapintaan toteutetut ominaisuudet. Rajapintaan integroituneilla asiakkailla on käytössään eri rajapintaversioita, joiden pyyntöjen URL-osoitteet ovat samat, mutta pyynnöissä tulee lähettää tieto siitä, mitä rajapintaversiota halutaan kutsua.

Valmiit dokumentointityökalut eivät sovellu eRA-järjestelmän dokumentointiin, sillä niiden avulla ei suoraan voida tuottaa haluttua LaTeX-dokumentaatiota. Niissä ei myöskään ole suoraan mahdollisuutta merkitä lähdekoodiin mihin rajapintaosaan rakenteet kuuluvat.

Swagger ja RAML ovat tarkoitettu REST-tyyppisten rajapintojen dokumentointiin, joten eRA-järjestelmän käärerajapintojen dokumentointi ei ole mahdollista käyttäen näitä työkaluja, sillä nämä eivät ole REST-tyyppisiä rajapintoja. Muodostettaessa Swagger-dokumentaatiota valmiista lähdekoodista tulee kuvauskommentit kirjoittaa lähdekoodiin. Tämä ei ole tässä työssä dokumentoitavan järjestelmän tapauksessa mahdollista, koska rajapinta täytyy hyväksyttää viranomaistaholla, kun sen lähdekoodiin on tehty muutoksia. Ei myöskään haluta kirjoittaa rajapinnan lähdekoodiin dokumentaatiota samalle rakenteelle usealla eri luonnollisella kielellä eikä useita kuvaustekstejä eri versioita varten. Dokumentoitavan eRA-järjestelmän tapauksessa lähdekoodin katselmointi on pakollinen tehtävä, joten myöskään tämän takia lähdekoodiin ei tahdota dokumentaatiota. Edellä mainituissa dokumentointityökaluissa ei myöskään ole mahdollisuutta vertailla dokumentaatiota ja rajapinnan toteutusta toisiinsa.

3.3 LaTeX-dokumentaatio

Työkalun edellytettiin muodostavan ja lukevan ennalta määriteltyä LaTeX-muotoista dokumentaatiota, jolla oli kuvattu jo aikaisemmin julkaistuja ja dokumentoituja rajapintaversioita. Dokumentaatio tulee olla eriytettynä niin, että yhteen LaTeX-dokumenttiin kirjataan valitun rajapintateknologian ja rajapintaversion metodit ja rakenteet valitulla luonnollisella kielellä. Dokumentti on jaettu lukuihin, joista kukin kuvaa yhtä rajapinnan osaa. Luku on jaettu alalukuihin, jotka ovat metodeja, luokkia ja luettelotyyppejä ja asetuksia kuvaavat alaluvut. Jokaisessa alaluvussa kyseisen alaluvun käsittelemät rakenteet on kuvattu aakkosjärjestyksessä tai lähdekoodiin merkityn attribuuttiluokan järjestysnumeron järjestyksessä.

Esimerkiksi XML-rajapintaan kuuluvan metodin dokumentaatiossa kuvataan ensin metodin nimi, kutsumiseen tarkoitettu URL-osoite, yleinen sanallinen kuvaus, parametrit XML-rakenteena, jossa esitetään luokkarakenteen ominaisuudet yhden iteraatiotason syvyyteen sekä ominaisuuksien pakollisuudet ja sanalliset kuvaukset. Pakollisuus tarkoittaa tässä tuleeko ominaisuuden arvo olla XML-elementin sisällä lähetetyssä pyynnössä. Samoin kuvataan vastausrakenne. Tämän lisäksi luetellaan mahdolliset virhekoodit. Ominaisuuksien ja virhekoodien kuvaukset voivat olla erilaisia eri rajapintaversioiden välillä.

LaTeX-dokumentti sisältää dokumentin sisäisiä linkkejä, tekstissä mainittuihin metodeihin, websivuihin, tyyppeihin ja vakioarvoihin. Esimerkiksi luokkaan viitatessa voidaan viittaus tehdä seuraavasti: \eRAWebClassLink<Luokan Nimi>.

```
1 \subsection{<HTTPMetodi> <URL>}\label{subsec:<MetodinNimi>}
2 \eRACommandSummary{<pakollisuuden kuvaus>}
3 {<pakollisuus>}{<pakollisuus>}{<pakollisuus>}
4 \subsubsection{Kuvaus}
5 \eRAParagraph{<Sanallinen kuvaus metodista>}
6 \subsubsection{Parametrit}
```

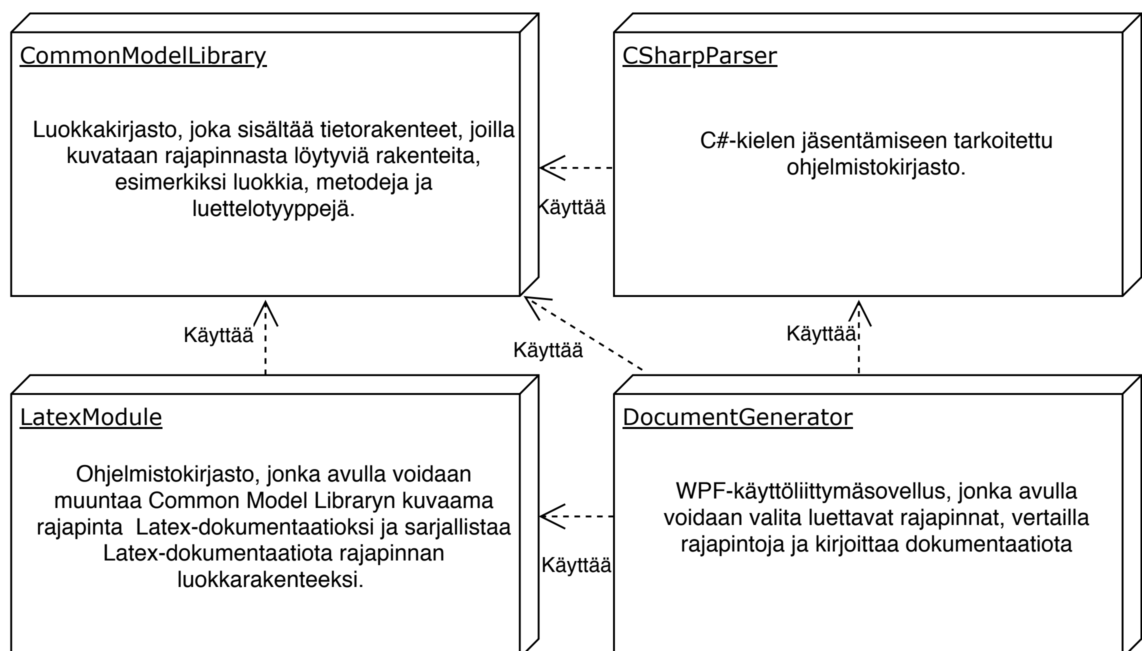
```
7  {\RaggedRight{}
8  \eRAXmlElementBegin{<XMLNimi1>}{<pakollisuus>}{0}
9  \eRAXmlElementBegin{>XMLNimi2>}{<pakollisuus>}{1}
10 \eRAXmlElement{<XMLNimi3>}{<pakollisuus>}{2}
11 \eRAXmlElementEnd{<XMLNimi2>}{<pakollisuus>}{1}
12 \eRAXmlElementEnd{<XMLNimi1>}{<pakollisuus>}{0}
13 }
14 \eRAParamTableBegin
15 \eRAParamTableItem{<XMLNimi2>}{<pakollisuus>}{<kuvaus>}
16 \eRAParamTableItem{<XMLNimi3>}{<pakollisuus>}{<kuvaus>}
17 \eRAParamTableEnd
```

Kuva 3.1. *Esimerkki LaTeX-dokumentaatioon kirjoitettavasta XML-metodin kuvauksesta.*

Kuvassa 3.1 on yksinkertaistettu esimerkki yhden XML-rajapinnan funktion kuvauksesta LaTeX-dokumentaationtyylissä. Kuvassa esitetyn metodin tapauksessa dokumentaatiossa voidaan metodille esittää myös vastausrakenne sekä virhekoodit samankaltaisella syntaksilla. Muut dokumentoitavat rakenteet kuvataan LaTeX-dokumentaatiossa samankaltaisesti.

4 TYÖKALUN TOTEUTUS

Suurimmat haasteet työn toteuttamisessa olivat rajapintaa kuvaavan tietomallin suunnittelu sekä jäsentäjän toiminnan suunnittelu ja toteutus. Jotta työkalu olisi laajennettavissa tukemaan useita ohjelmointikieliä ja dokumentaatiotekniikoita, toteutettiin arkkitehtuuri modulaariseksi niin, että jokaiselle tuetulle ohjelmointikielelle voidaan toteuttaa oma jäsentäjämoduulinsa. Myös dokumentaation muodostamiseen ja dokumentaation tietomalliksi jäsentämiselle toteutettiin oma moduulinsa. Vaatimuksena oli toteuttaa moduulit LaTeX-dokumentaatiolle ja C#-lähdekoodille. Samaa tietomalli-moduulia sekä käyttöliittymä-moduulia voidaan käyttää riippumatta siitä, millä ohjelmointikielellä toteutettua rajapintaa tahdotaan kuvata.



Kuva 4.1. *Työkalun korkean tason arkkitehtuuri.*

Kuvassa 4.1 on toteutetun työkalun korkean tason arkkitehtuuri. Tulevaisuudessa jäsentäjämoduuleita voi siis olla useampia, jos ilmenee tarvetta dokumentoida rajapintoja, jotka ovat toteutettu muilla ohjelmointikielillä kuin C#. Myös dokumentaatiomoduuleita voidaan kirjoittaa lisää, jos halutaan tuottaa erilaista dokumentaatiota, esimerkiksi HTML-muodossa. Kaikki jäsentäjä- ja dokumentaatiomoduulit käyttävät samaa tietomallia. Jäsentäjän tehtävä on tunnistaa ja rakentaa tietomalliin tarpeelliset rakenteet.

4.1 Jäsentäjän toteutustapojen vertailu

Jäsentäjän tarkoitus on lukea lähdekoodi ja muodostaa siitä rajapintaa kuvaava tietomalli. Jäsentäjän toteuttamisessa harkittiin kolmea eri toteutustapaa. Ensimmäinen tapa oli staattisen lähdekoodin syntaksin lukemiseen perustuvan jäsentäjän toteuttaminen itse, käyttäen esimerkiksi säännöllisiä lausekkeita. Toisena vaihtoehtona jäsentäjän toteutustavaksi oli käännetyn kirjaston tutkiminen reflektion avulla tyyppi kerrallaan. Kolmas vaihtoehto oli löytää jokin valmiiksi tehty vapaasti käytettävä työkalu lähdekoodin tutkimiseen ja soveltaa sitä.

4.1.1 Ensimmäinen vaihtoehto: oman jäsentäjän toteuttaminen

Ensimmäinen vaihtoehto jäsentäjän toteuttamiselle oli kirjoittaa itse kokonainen jäsentäjä alusta alkaen. Jäsentäjän tulisi siis toimia samalla tavoin kuin ohjelmointikielen kääntäjän jäsentäjä, mutta tuottaa tuloksena halutun mukainen tietomalli. Välttämättä näin teknistä ja kattavaa ratkaisua ei tarvita, koska tietomallin ei ole tarkoitus esittää koko lähdekoodia, vaan vain rajapintaa ja rajapinnan kannalta oleellisia rakenteita. Esimerkiksi funktioiden toimintaa ei ole tarpeellista tutkia, vaan funktion määreiden, parametrien ja paluuarvon tunnistaminen riittää, jotta tietomalliin voidaan saada kuvaus funktiosta.

Lähdekoodista olisi mahdollista etsiä kielelle määriteltyjä varattuja avainsanoja ja peräkkäisiä sanapareja, esimerkisi ”public” ja ”class” tai ”private” ja ”enum” ja niiden löytymisen jälkeen etsiä näillä avainsanoilla tunnistettua lohkoa paikantamalla parilliset aaltosulkeet. Tämän jälkeen voitaisiin etsiä esimerkiksi luokkaa kuvaavan lohkon sisältä luokan julkisia ominaisuuksia samalla tavoin. Toteutuksen positiivisena puolena on se, että lähdekoodia ei tarvitse kääntää, kun sitä luetaan. Tällöin mahdollisista semanttisista virheistä ei tarvitsisi välittää, sillä jäsentäjä tutkisi vain syntaksia. Avainsanaparien ja lohkojen tarkasteluun perustuvasta jäsentämisestä voidaan saada nopea, sillä lohkoja, jotka eivät ala sanapareilla, joista ei olla kiinnostuneita, ei tarvitse tarkastella tarkemmin, vaan niiden yli voidaan hypätä tarkastelussa.

Rajapinnan lukeminen olisi mahdollista toteuttaa nopeammin syntaksiin perustuvalla tarkastelulla, kuin tutkimalla käännettyä koodia, sillä kääntämisessä kuluvaa aikaa ei tarvitsisi odottaa prosessissa. Toteutettava jäsentäjä voisi olla toiminnaltaan nopea myös siksi, että jäsentäjän ei välttämättä tarvitsisi jäsentää kaikkia lähdekoodin rivejä, sillä funktioiden toteutuksesta ei olla kiinnostuneita rajapintakuvauksen tapauksessa. Kuitenkin esimerkiksi muuttujien tietotyyppien ratkaiseminen voisi olla vaikeaa, sillä jos tietotyyppiä ei ole määritelty samassa tiedostossa ja tiedostossa käytetään using-avainsanalla määriteltyjä viittauksia muiden lähdekooditiedostojen nimiavaruuksiin, ei lähdekoodin syntaksin avulla voida päätellä, mistä nimiavaruudesta tietotyyppi on kotoisin. Ei välttämättä riitä, että vain tietotyypin nimi saadaan tietää, koska projektissa voi olla useita nimiavaruuksia, joissa on määritelty samannimisiä luokkia.

Ainut vaihtoehto saada selville tyypit, joiden esittelyssä ei ole käytetty syntaksia <nimiavaruus>.<tyyppi>, on käydä kaikki projektin tiedostot läpi ja tutkia kaikki tiedostossa viitattujen nimiavaruuksien määritellyt tietotyypit. Tällä tavoin rakennetusta jäsentäjästä on työlästä ja vaikeaa tehdä luotettavaa kaikissa tapauksissa, niin että lukuvirheitä ei sattuisi minkään muotoisen lähdekoodin osalta.

4.1.2 Toinen vaihtoehto: käännetyn kirjaston tutkiminen reflektiolla

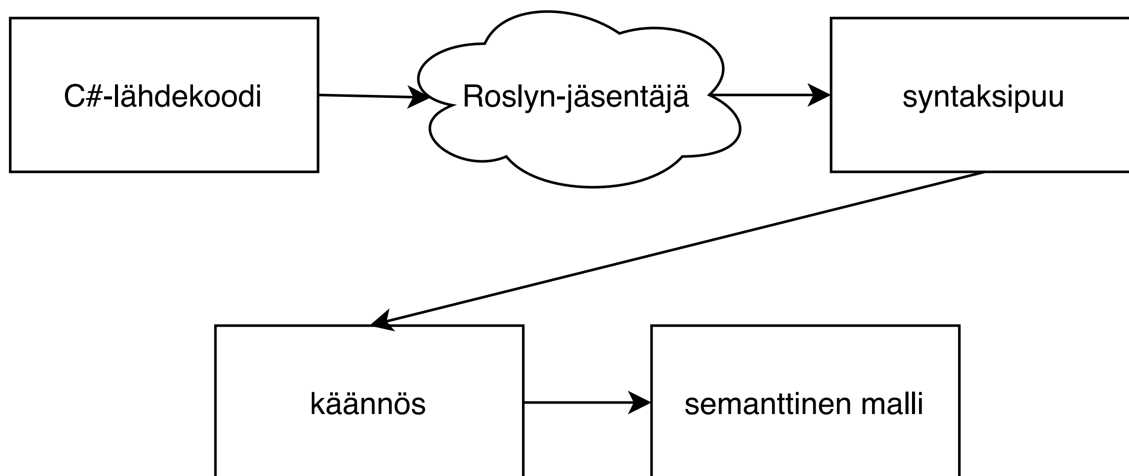
Toinen vaihtoehto jäsentäjän toteuttamistavaksi oli käännetyn kirjaston läpikäymiseen reflektion avulla. Reflektion avulla voidaan tutkia dynaamisesti ohjelman tilaa .NET-kehykseen rakennetun luokkien tyypin palauttavan GetType-metodin avulla [46]. Reflektiolla voidaan ensin hakea GetAssemblies-metodilla System.Reflection.Assebly-olioihin tiedot kaikista kirjastossa käytetyistä käännöksistä [36]. Sitten voidaan hakea kultakin System.Reflection.Assembly-oliolta käännöksen kaikki tyypit GetTypes-komennolla [37]. Paluuarvona GetTypes-komennolla saadaan taulukko System.Type-oliota. Kuitenkin tällä tavalla menetellessä täytyy käydä läpi tutkitun käännöksen kaikki tyypit, ennen kuin voidaan tietää, mistä tyypeistä ollaan kiinnostuneita. Esimerkiksi tyyppien lähdetiedostosijainnin perusteella tyyppejä ei voida hakea, sillä tämä tieto ei tallennu käännöksen metatietoihin. Käännöksen metatietoihin eivät tallennu myöskään esimerkiksi mahdolliset kommentit, joita lähdekoodiin on kirjoitettu, joten niitä ei voida tutkia reflektion avulla. Reflektioon perustuva jäsentäjä voi olla hidas silloin, kun käännös sisältää hyvin paljon tyyppejä.

4.1.3 Kolmas vaihtoehto: valmiin työkalun soveltaminen

Kolmas vaihtoehto jäsentäjän toteuttamiselle oli käyttää apuna jotain valmiiksi toteutettua työkalua. C#-kielen kääntäjään perustuva Roslyn [32] oli sopiva työkalu, jolla jäsentäjän toteutus voitiin tehdä C#-moduuliin. Roslyn tarjoaa rajapintoja kääntämisprosessissa tapahtuvaan koodin analysointiin ja kääntämisprosessissa syntyviin tietoihin. Roslynin kääntäjärajapinnan metodien avulla voidaan tarkastella koodia sekä syntaktisella että semanttisella tasolla. Roslynia käyttäen voidaan kääntää lähdekoodia, mutta myös tutkia sen syntaksia ja jäsentää lähdekoodi syntaksipuuksi. Roslynin ohjelmointirajapinnasta voidaan saada lähdekooditiedoston syntaksipuu ja tarkastella kaikkia yksittäisiä solmuja puussa. Syntaksipuu sisältää kaiken lähdekoodin sisältämän tiedon mukaan lukien kommentit, tyhjämerkit ja rivinvaihdot, joten syntaksipuusta kannattaa poimia tietoa vain koodin toiminnan kannalta tärkeistä solmuista.

Syntaksipuun tarkastelun lisäksi Roslynilla on mahdollista kääntää lähdekoodi ja tutkia sen semanttista mallia. Semanttinen malli käyttää hyväkseen käännettyä koodia, joka tallennetaan kääntämisen tuloksena Microsoft.CodeAnalysis.CSharp.Compilation-luokkaan [39]. Käännetystä koodista voidaan hakea syntaksipuun avulla syntaksipuuta vastaavan

osan semanttinen malli. Semanttisesta mallista voidaan hakea symboleja mallissa olevan tiedon avulla. Symbolit ovat nimiavaruuksia, tyyppejä, metodeja, kenttiä, tapahtumia, parametreja tai paikallisia muuttujia. Symbolit ovat samankaltaisia oliota kuin reflektion avulla tarkasteltavat tyypit, mutta eivät täysin samoja kuin System.Reflection-rajapinnan avulla käsiteltävät tyypit. Semanttisen mallin hyötyjä verrattuna syntaksipuuhun on esimerkiksi se, että semanttisen mallin kautta voidaan suoraan hakea oikeat tyyppiviittaukset muuttujiin. Semanttista mallia ei luonnollisesti voida käyttää, jos lähdekoodi ei käänny. [32]



Kuva 4.2. *Semanttisen mallin tuottaminen Roslynin avulla.*

Kuvassa 4.2 esitetään kuinka lähdekoodista voidaan muodostaa semanttinen malli. Kuvassa näkyvät syntaksipuu käännös sekä semanttinen malli ovat kaikki Roslyn työkalun tuottamia rakenteita.

4.1.4 Valittu toteutustapa: Roslyn-työkalun käyttäminen

C#-jäsentäjän toteutustavaksi valittiin Roslynia hyödyntävä ratkaisu, sillä sen avulla saatiin pienennettyä itse toteutettavaa työmäärää. Valmiin työkalun etuina ovat jäsentäjän testattu toiminta ja toimiva kokonaisuus. Roslynin valitsemista tuki myös se, että jäsentäjän toteutuksessa voidaan hyödyntää sekä syntaktista että semanttista mallia. Esimerkiksi jos olisi tarve lukea lähdekoodiin kirjoitettuja kommentteja, ei niiden lukeminen onnistuisi reflektioon pohjautuvalla toteutuksella, sillä käännettyyn koodiin eivät siirry ne syntaktiset yksityiskohdat, jotka eivät vaikuta toimintaan kuten rivinvaihdot, tyhjämerkit ja kommentit. Semanttiseen malliin pohjautuvan työkalun metodien avulla saadaan helposti selville nimiavaruudet ja viitteet.

Koska dokumentoitava järjestelmä on toteutettu Microsoftin Visual Studio -kehitysympäristöllä, voidaan jäsentäjän toteutuksessa hyödyntää myös Roslyn Workspace APIa [57] Visual Studion ratkaisun (engl. solution) ja projektien löytämiseen sekä kääntämiseen. Visual Studion ratkaisu on kokoelma projekteja sekä asetuksia muun muassa projek-

tien kääntämiseen ja niiden tarvitsemiin muihin tiedostoihin [8]. Roslyn Workspace APIn avulla kääntämisessä ei itse tarvitse tutkia projektien välisiä mahdollisia riippuvuuksia tai kääntämisjärjestystä, vaan Roslyn Workspace APIn avulla voidaan kääntää yksittäinen projekti tai ratkaisu helposti. Toteutetun C#-jäsentäjän toimintaa esitellään kattavammin aliluvussa 4.4.

4.2 Dokumentoitavien rakenteiden tunnistaminen koodista

Kaikkia lähdekoodissa sijaitsevia rakenteita ei lisätä tietomalliin, vaan lähdekoodista tulee tunnistaa rajapinnan kannalta oleelliset rakenteet. Haluttujen rakenteiden tunnistamiseen toteutettiin attribuuttiluokat ja nämä lisättiin lähdekoodiin kaikille dokumentoitaville rakenteille. Luettelotyypit, metodit, luokat ja niiden ominaisuudet merkittiin uusilla attribuuttiluokilla. Kaikille attribuuteille tulee antaa parametreja, joilla kerrotaan tietoa rakenteen käytöstä. Toteutettuja attribuutteja ovat:

- DocumentationBaseAttribute
- DocumentationClassAttribute
- DocumentationPropertyAttribute
- DocumentationXmlRequestParameterAttribute
- DocumentationXmlResponseParameterAttribute
- DocumentationXmlInterfaceMethodAttribute
- DocumentationXMLPageAttribute
- eRAToolkitInterfaceMethodDocumentationAttribute
- eRAToolkitInterfaceCallbackMethodDocumentationAttribute
- eRAToolkitInterfaceSettingDocumentationAttribute.

Attribuutteja toteutettiin useita, sillä dokumentoitavat rakenteet tarvitsevat erilaisia tietoja. Esimerkiksi metodien kuvaamiseen tarvitaan tieto pyynnön kutsumiseen vaadittavista esiehdoista, mahdollisista poikkeuksista sekä tieto rajapintaversioista. Luokan ominaisuuden kuvaamiseen tarvitaan erilaisia tietoja kuten versiot, joihin ominaisuus kuuluu sekä pakollisuus sijaita sarjallistetussa pyyntö- tai vastausrakenteessa. Myös ylläpidettävyyden kannalta on hyödyllistä, että on olemassa useita attribuuttiluokkia eri tarkoituksiin. Kaikkea dokumentaatiossa tarvittavaa tietoa ei pystytä päättelemään helposti lähdekoodista ohjelmallisesti, kuten mahdollisia poikkeuksia, loogista rajapintaosaa, johon rajapintakutsut tai rakenteet kuuluvat, tai metodin kutsumiseen tarvittavaa ohjelman tilaa. Tämän vuoksi tiedot lisättiin lähdekoodiin attribuutteihin.

Attribuutit on suunniteltu versiokohtaisiksi niin, että attribuutit periytyvät DocumentationBaseAttribute-luokasta, joka sisältää vain tiedon attribuutin saavan rakenteen rajapintaversiosta. Luokka on esitelty kuvassa 4.1. Versionumerointi on toteutettu niin, että jokaisessa attribuutissa kerrotaan rakenteen ensimmäinen ja viimeinen rajapintaversio. Rajapintaversion merkitseminen on toteutettu lisäämällä lähdekoodiin luette-

```
1  [AttributeUsage(AttributeTargets.All)]
2  public class DocumentationBaseAttribute : Attribute
3  {
4      public InterfaceDocumentationVersion FirstVersion { get; set; }
5      public InterfaceDocumentationVersion LastVersion { get; set; }
6
7      public DocumentationBaseAttribute(
8      InterfaceDocumentationVersion firstVersion,
9      InterfaceDocumentationVersion lastVersion)
10     {
11         FirstVersion = firstVersion;
12         LastVersion = lastVersion;
13     }
14 }
```

***Ohjelma 4.1.** Attribuuttiluokka DocumentationBaseAttribute, josta muut attribuuttiluokat periytetään.*

```
1  [AttributeUsage(AttributeTargets.Class | AttributeTargets.Enum,
2  AllowMultiple = true)]
3  public class DocumentationClassAttribute : DocumentationBaseAttribute
4  {
5      public InterfaceDocumentationSection Section { get; set; }
6
7      public DocumentationClassAttribute(
8          InterfaceDocumentationVersion firstVersion,
9          InterfaceDocumentationVersion lastVersion,
10         InterfaceDocumentationSection section)
11         : base(firstVersion, lastVersion)
12     {
13         Section = section;
14     }
15 }
```

***Ohjelma 4.2.** Luokkien ja luettelotyyppien merkitsemiseen käytetty attribuuttiluokka.*

lotyyppi rajapintaversiolle, jossa arvo 0 kuvaa tuntematonta versiota. Kun rakenteelle merkitään alkuversioksi esimerkiksi 1.0 ja loppuversioksi 2.1, kuuluu rakenne kaikkiin näiden välillä oleviin versioihin eli 1.0, 1.1, 1.2, . . . , 2.1. Jos rakenteen loppuversioksi merkitään tuntematon versio, kuuluu rakenne kaikkiin alkuversiota uudempiin versioihin. Jos alkuversioksi merkitään tuntematon versio, ei rakenne kuulu rajapintaan. Rajapintaan kuuluu eri osia ja myös ne merkitään attribuutteihin. Osat merkitään omalla InterfaceDocumentationSection-luettelotyypin arvollaan.

Rajapintaluokat merkitään DocumentationClassAttribute-luokalla, joka on esitelty ohjelmassa 4.2. DocumentationClassAttribute-luokalle annetaan vain versionumerot ja rajapintaosa, johon luokka kuuluu.

Luokan ominaisuuksille (engl. property) voidaan lisätä metatietoja käyttäen Documenta-

tionProperty-attribuuttia. Tällä luokalla merkitään ominaisuudet, jotka eivät sisälly luokkaan kaikilla DocumentationClass-attribuutilla merkityissä rajapintaversioissa. Esimerkiksi luokalle voidaan antaa DocumentationClassAttribute-luokka, jolla merkitään se kuuluvaksi kaikkiin rajapintaversioihin versiosta 2.0 eteenpäin rajapinnan yleisessä osassa. Kuitenkin osa luokan ominaisuuksista voidaan merkitä kuuluvaksi vain uudempiin versioihin, esimerkiksi versiosta 2.1 eteenpäin. Jos ominaisuuksille ei ole annettu omaa attribuuttia, tulkitaan ne kuuluviksi samaan rajapintaversioon tai rajapintaversioihin kuin isäntäluokka. Ominaisuuksille voidaan antaa myös kahdenlaisia muita attribuutteja, joilla kerrotaan XML-metodissa paluuarvona tai parametrina käytettävän luokan pakollisesti annettavista arvoista. Jokaiselle ominaisuudelle voidaan antaa DocumentationXml-RequestParameter-attribuutti, jolle voidaan antaa versionumeroiden lisäksi DocumentationObjectCompulsion-luettelotyypin arvo, jolla kuvataan täytyykö ominaisuuden arvon löytyä sarjallistetun luokan XML-muodosta, kun luokka annetaan XML-rajapinnan pyynnössä parametrina metodille. Pakollisuutta kuvaavan DocumentationObjectCompulsion-luettelotyypin mahdolliset arvot ovat: ei-vaadittu, vaadittu ja ehdollisesti vaadittu. Jos attribuuttia ei anneta luokan ominaisuudelle, tulkitaan ominaisuus ei-vaadituksi kentäksi parametrina annettavassa XML-rakenteessa. DocumentationXmlResponseParameterAttribute-luokan rakenne ja toiminta on samankaltainen kuin DocumentationXml-RequestParameterAttribute, mutta tällä merkitään vastauksena saadun XML-rakenteen ominaisuuksia.

XML-rajapintateknologialla toteutetuille metodeille annetaan DocumentationXmlInterfaceMethod-attribuutti, joka sisältää tietoa versioista, osiosta, vaadituista parametreista, mahdollisista poikkeuksista ja ohjelman sallitusta tilasta ennen metodin kutsumista. Myös metodeja varten luotu attribuutti on rajapintaversiokohtainen, sillä eri versioissa vaadittavat parametrit tai heitetyt poikkeukset voivat vaihdella versiosta toiseen. DocumentationXmlInterfaceMethodAttribute-luokka on näkyvissä ohjelmassa 4.3.

Ohjelmassa 4.4 näytetään esimerkki, jossa DocumentationXmlInterfaceMethodAttribute-luokkaa käytetään yhden rajapintakomennon merkitsemiseen lähdekoodissa. Metodi kuuluu kaikkien rajapintaversioiden DentalArchive-osaan versiosta 2.1 eteenpäin. Metodia ei ole pakko toteuttaa, mutta se vaatii avatun istunnon, potilaan tietojen avaamisen, sekä avatun palvelutapahtuman. Kaikki XML-rajapinnan pyynnöt ottavat parametrina InterfaceRequest-luokan. Esimerkissä InterfaceRequest-luokan sisältämistä kentistä vaaditaan täytettäväksi SessionId, ContextId sekä ResetToothStatusRequest. Lisäksi pyyntö voi laukaista useita virhekoodeja. ResetToothStatusRequest-luokan vaatimat täytettävät arvot tulkitaan luokan esittelystä, sekä luokan ominaisuuksille annetuista DocumentationXmlRequestParameter-attribuuteista.

Jos luokan X ominaisuudelle on annettu DocumentationXmlRequestParameter-attribuutti, jonka mukaan ominaisuus on pakollinen, mutta luokan Y ominaisuudelle, joka on tyyppiä X, on annettu attribuutti, joka kertoo ei-pakollisuudesta, luokan X ominaisuuksia ei tarvitse löytyä sarjallistetusta XML-rakenteesta.

DocumentationXMLPageAttribute-luokalla merkitään XML-rajapinnassa sijaitsevia HTTP

```
1  [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
2  public class DocumentationXmlInterfaceMethodAttribute
3          : DocumentationBaseAttribute
4  {
5      public int Ordinal { get; set; }
6      public InterfaceDocumentationSection Section { get; set; }
7      public DocumentationObjectCompulsion Implementation { get; set; }
8      public DocumentationObjectCompulsion Session { get; set; }
9      public DocumentationObjectCompulsion Patient { get; set; }
10     public DocumentationObjectCompulsion ServiceEvent { get; set; }
11     public RequiredInterfaceXmlParameters Parameters { get; set; }
12     public Type RequiredParameterType { get; set; }
13     public Type ResponseParameterType { get; set; }
14     public ErrorType[] Exceptions { get; set; }
15
16     public DocumentationXmlInterfaceMethodAttribute(int ordinal,
17         InterfaceDocumentationVersion firstVersion,
18         InterfaceDocumentationVersion lastVersion,
19         InterfaceDocumentationSection section,
20         DocumentationObjectCompulsion implementation,
21         DocumentationObjectCompulsion session,
22         DocumentationObjectCompulsion patient,
23         DocumentationObjectCompulsion serviceEvent,
24     RequiredInterfaceXmlParameters parameters,
25         Type requiredParameterType,
26         Type responseParameterType, params ErrorType[] exceptions)
27         : base(firstVersion, lastVersion)
28     {
29         Ordinal = ordinal;
30         Section = section;
31         Implementation = implementation;
32         Session = session;
33         Patient = patient;
34         ServiceEvent = serviceEvent;
35         Parameters = parameters;
36         RequiredParameterType = requiredParameterType;
37         ResponseParameterType = responseParameterType;
38         Exceptions = exceptions;
39     }
40 }
```

Ohjelma 4.3. *XML-metodien merkitsemiseen käytetty attribuuttiluokka.*

```
1  [DocumentationXmlInterfaceMethod(7,
2  InterfaceDocumentationVersion.V2_1,
3  InterfaceDocumentationVersion.Unknown,
4  InterfaceDocumentationSection.DentalArchive,
5  DocumentationObjectCompulsion.NotRequired,
6  DocumentationObjectCompulsion.Required,
7  DocumentationObjectCompulsion.Required,
8  DocumentationObjectCompulsion.Required,
9  RequiredInterfaceXmlParameters.SessionID |
10 RequiredInterfaceXmlParameters.ContextID,
11 typeof(ResetToothStatusRequest), typeof(ResetToothStatusResponse),
12 ErrorType.InvalidXml, ErrorType.InvalidContextID,
13 ErrorType.UserNotLoggedIn,ErrorType.PatientNotOpen,
14 ErrorType.SessionLocked, ErrorType.ServiceEventNotOpen,
15 ErrorType.DentalStatusEditingNotStarted,
16 ErrorType.InvalidSenderIP, ErrorType.InvalidDentalStatusID,
17 ErrorType.InvalidToothSupernumeraryValue,
18 ErrorType.InvalidToothNumber)]
19 public InterfaceResult ResetToothStatus(InterfaceRequest request)
20 {
21 ...
22 }
```

***Ohjelma 4.4.** Esimerkki XML-metodin merkitsemisestä attribuuttiluokalla.*

GET -metodilla kutsuttavia metodeja. Tällä luokalla voidaan kertoa pyynnön sallimien GET-parametrien nimet, pakollisuudet ja esimerkkiarvot.

Rajapinnan käärekirjastoissa dokumentoitavien rakenteiden tunnistamiseen käytetään attribuuttiluokkia, joiden nimessä mainitaan eRAToolkit. eRAToolkitInterfaceMethod-Documentation-attribuuttiluokalla merkitään rajapintaan kuuluvia metodeja. eRAToolkitInterfaceMethodDocumentationAttribute on hyvin samankaltainen XML-rajapinnassa käytetyn DocumentationXmlInterfaceMethod-attribuutin kanssa. Eroavaisuutena näillä attribuuteilla on se, että poikkeustaulukossa poikkeuksia listataan eri luettelotyypin arvoilla. eRAToolkitInterfaceMethodDocumentationAttribute-luokalla voidaan antaa myös tieto siitä, mitä URL-osoitteita metodi kutsuu XML-rajapinnasta. URL-osoitteiden avulla voidaan hakea XML-rajapinnan metodeilta tieto, mitä poikkeuksia voidaan heittää palvelimen päästä.

eRAToolkitInterfaceCallbackMethodDocumentationAttribute-luokalla merkitään käärekirjastoissa takaisinkutsufunktioita, joita integroitavaan asiakasjärjestelmään tulee toteuttaa. Tätä attribuuttia kirjataan delegaattifunktioille (engl. delegate) ja attribuutin rakentajalle annetaan tieto parametrien nimistä ja siitä onko takaisinkutsufunktion toteutus pakollista toteuttaa asiakasjärjestelmään.

Attribuuttiluokkia käyttämällä jäsentäjän toimintaa saadaan nopeutettua, kun syntaksipuusta tarkastellaan vain sellaisten solmujen jälkeläisenä olevia solmuja, joille on annettu jokin edellä esitetyistä attribuuttiluokista. Esimerkiksi metodien tapauksessa kaikki meto-

dit, joille ei ole annettu attribuuttiluokkaa, jätetään tarkastelematta. Roslynin semanttisen mallin avulla voidaan tutkia, onko tarkasteltavalla rakenteella haluttuja attribuuttiluokkia ja tutkia niiden parametreja.

Lähdekoodista voidaan tunnistaa ja lisätä metodeja tietomalliin myös halutun tyyppisten paluuarvojen avulla. Lisäksi ei-julkiset luokat, metodit ja luettelotyypit voidaan suodattaa pois. Tällöin tietomallin muodostaminen lähdekoodista on mahdollista myös ilman siihen lisättäviä attribuuttiluokkia, mutta saatava informaatio ei ole niin hyödyllistä, sekä jäsentämisessä kuluva aika kasvaa.

4.3 Tietomallimoduuli

Tietomallimoduulin tarkoitus on kuvata rajapinnan rakenteita ja säilyttää rajapintaa koskevaa dokumentaatiota. Tietomallimoduuli ei sisällä logiikkaa tietorakenteen muodostamiseen vastaamaan lähdekoodia, vaan tietomallimoduuli on passiivinen luokkakirjasto, jonka luokkarakenteita jäsentäjämoduuli käyttää ja rakentaa lähdekoodin jäsennyksen perusteella rajapintaa kuvaavat luokat.

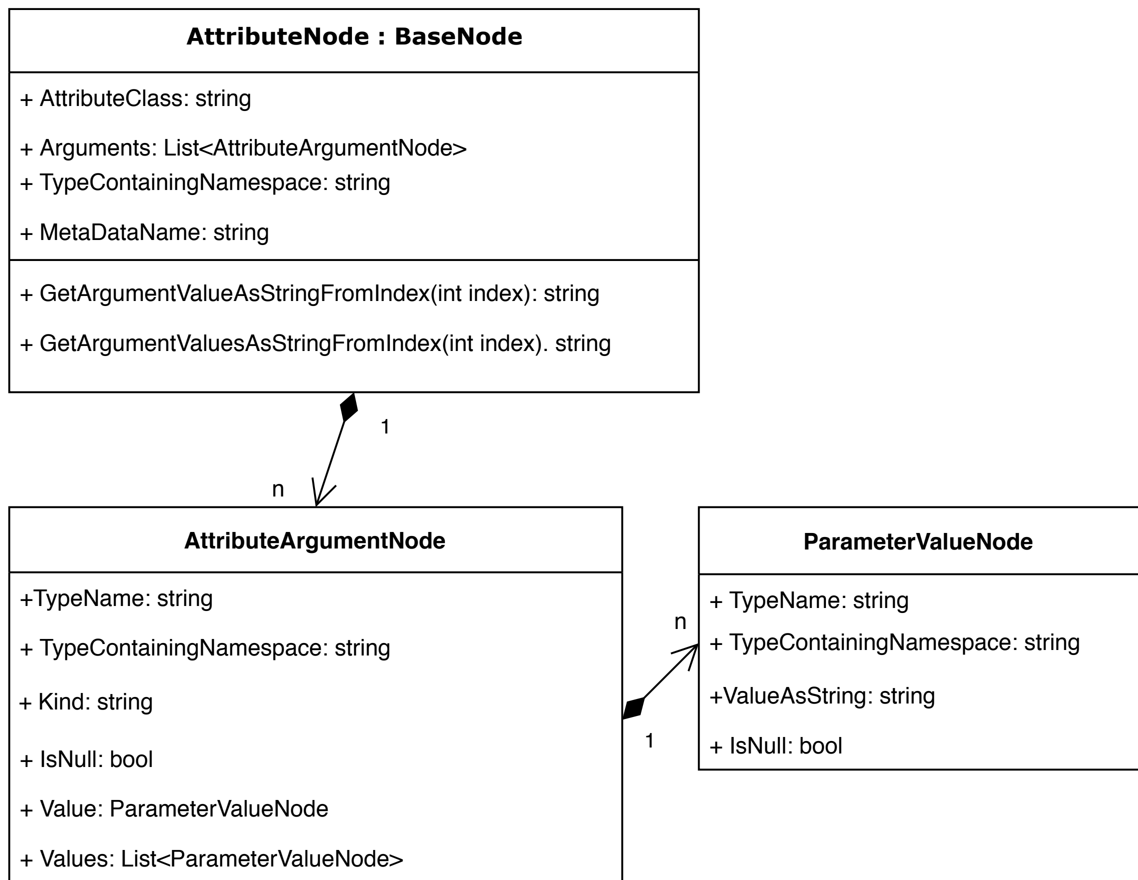
Tietomallin suunnittelussa lähdettiin liikkeelle siitä, että koodia kuvaavan luokkarakenteen kannattaa muistuttaa abstraktia syntaksipuuta, jotta luokat säilyisivät pieninä toisistaan eroavina kokonaisuuksina. Koska tietomalli kootaan jäsentäjän muodostamasta syntaksipuusta, on syntaksipuun kaltainen rakenne helposti koostettava. Kaikkea tietoa ei kutenkaan kannata sisällyttää malliin, sillä rajapinnan kannalta kaikki ratkaisusta löytyvä koodi ei ole oleellista. Tietomalli kuvaa vain rajapintaan kuuluvat metodit, luokat ja luettelotyypit. Metodien toimintaa ei tallenneta tietomalliin. Metodien toimintaa ei tallenneta siksi, että tietomalli on ensisijaisesti tarkoitettu rajapintojen dokumentointiin ja rajapintaa dokumentoidessa ei ole oleellista esittää tietoa rajapintafunktioiden sisäisestä toiminnasta. Jos tietomalliin tallennettaisiin kaikki syntaksipuusta löytyvä tieto, tulisi tallennettavasta tietomallista kohtuuttoman iso ja monimutkainen toteuttaa. Tietomalli voidaan sarjallistaa XML-muotoon ja tallentaa tiedostoon. Sarjallistaminen on toteutettu C#-kielen System.Xml.Serialization-kirjaston avulla. [55]

Seuraavaksi esitellään lyhyesti, millaisista luokista koko rajapinnan rakennetta kuvaava InterfaceModel-luokka koostuu. Luokkia esitellään seuraavissa alaluvuissa, järjestyksessä, jossa lähdetään ensin tarkastelemaan pienempiä kokonaisuuksia ja edetään sitten tarkastelemaan jo esiteltyjen kokonaisuuksien avulla suurempia rakenteita. Luokkia kuvaavissa Universal Modelling Language (UML) -luokkakaavioissa [74] ei aina ole esitelty kaikkia luokan rakenteita ja toiminnallisuuksia, vaan on pyritty selkeyttämään jättämällä osa ei niin oleellisista tiedoista pois.

4.3.1 Attribuutin tietojen tallentaminen

Attribuuteilla merkitään dokumentoitavassa ohjelmistossa rajapintaan kuuluvia rakenteita ja metodeja. Siksi on tärkeää, että tarvittavat tiedot attribuuteista saadaan tallennettua

tietomalliin. Attribuuttia ja sille parametreina annettavia arvoja kuvataan AttributeNode-luokalla. AttributeNode-luokka kuvaa yhtä rakenteelle annettua attribuuttia. Luokkaan tallennetaan attribuuttiluokan tietotyyppi, nimiavaruus, symbolinen nimi ja lista attribuutin rakentajalle annettavista argumenteista. AttributeArgumentNode kuvaa yhtä attribuutin rakentajalle annettavaa parametria. Jos parametri on taulukko, sisältää Values-muuttuja listan annetuista taulukon arvoista ja tyypeistä. Kuva 4.3 esittää attribuutin tietojen tallentamisessa käytetyt luokkarakenteet.



Kuva 4.3. *Attribuuttiluokan tietojen tallentaminen tietomalliin.*

Tietomallin luokat, jotka tallentavat attribuutteja periytetään abstraktista NodeWith-Attributes-luokasta, jossa on apufunktioita attribuuttilistojen käsittelyyn.

4.3.2 eRA-järjestelmän rajapintoihin liittyvän tiedon tallentaminen

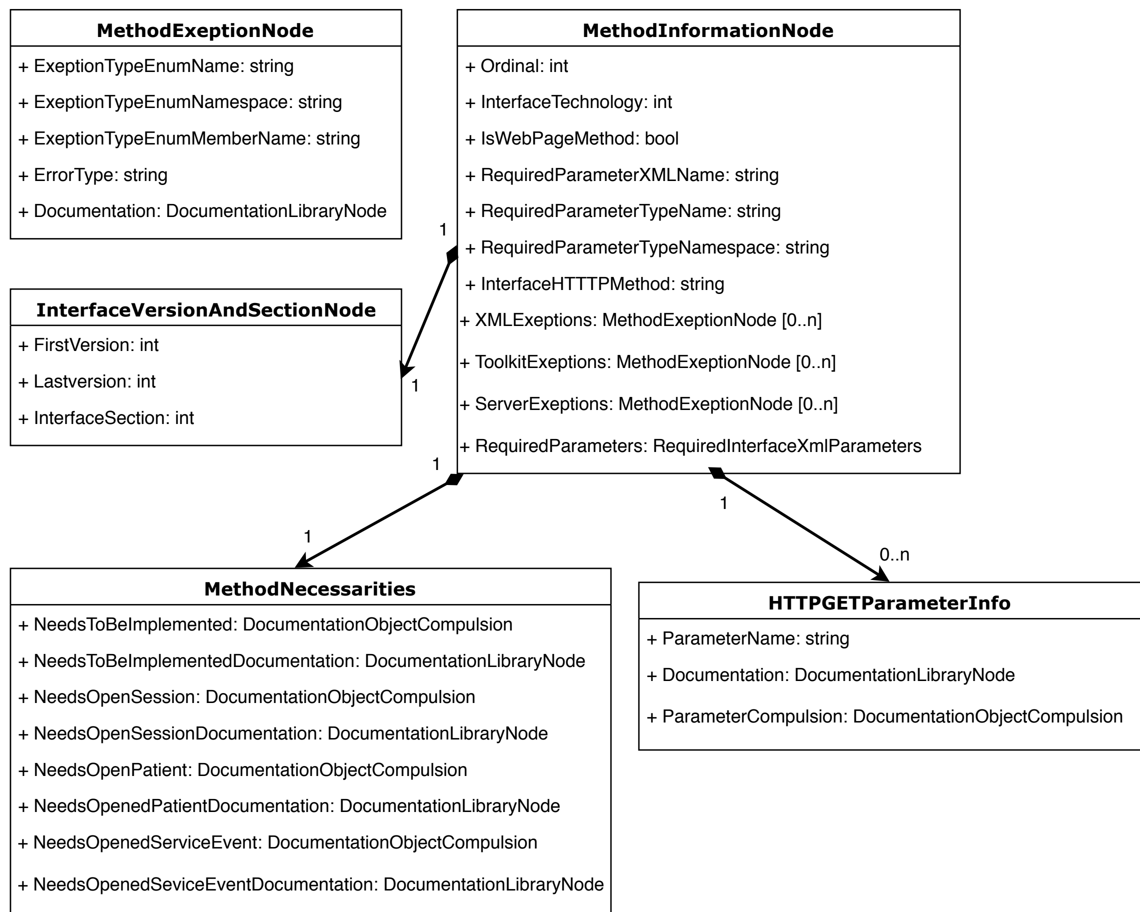
Luvussa 4.2 esiteltyjen attribuuttiluokkien tietojen tallentamista varten toteutettiin tietomalliin osa samoista luettelotyypeistä, joita lisättiin dokumentoitavan järjestelmän lähdekoodiin rajapinnan osia kuvaavissa attribuuttiluokissa. Lisätyt luettelotyypit ovat:

- InterfaceDocumentationSection
- InterfaceDocumentationVersion

- DocumentationObjectCompulsion
- InterfaceTechnology
- RequiredInterfaceXMLParameters
- ServerErrorType.

Luettelotyyppien lisäämisen ansiosta voitiin rakentaa tietorakenteita tallentamaan tietoa edellisessä luvussa esiteltyjen attribuuttien avulla dokumentoiduista rakenteista. InterfaceVersionNode-luokka tallentaa tiedon ensimmäisestä sekä viimeisestä versiosta, johon rajapinnan osa kuuluu. InterfaceVersionAndSectionNode-luokka tallentaa versioiden lisäksi myös rajapinnan osan tiedon.

Rajapintafunktioita varten toteutettiin MethodInformationNode-luokka, joka tallentaa metodille lisätyn attribuutin tiedot. Tämä luokka muodostetaan, jos funktiolle on lähdekoodissa merkitty DocumentationXmlInterfaceMethodAttribute- tai eRAToolkitInterfaceMethodDocumentation-attribuutteja.

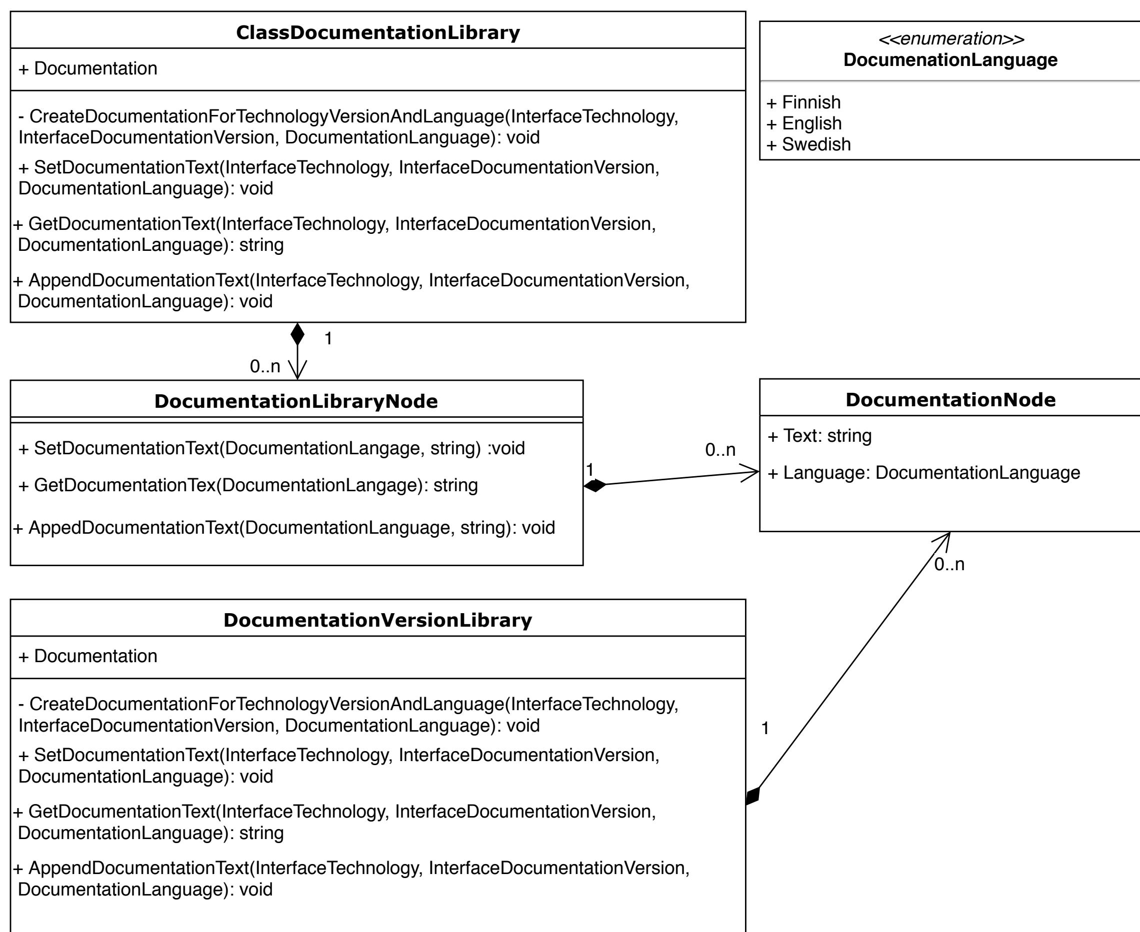


Kuva 4.4. *Metodien dokumentointiin tarvittava eRA-järjestelmän attribuuttiluokan tietoja tallentava kokonaisuus*

Kuvassa 4.4 on MethodInformationNode-luokan rakenne. Dokumentaation tallentamiseen tarvittavan DokumentationLibraryNode-luokan rakenne esitellään aliluvussa 4.3.3.

4.3.3 Dokumentaation tallentaminen

Dokumentaatiota tallennetaan tietomalliin usealla eri luonnollisella kielellä. Käyttöliittymässä on tuki suomen, englannin ja ruotsin kielelle. DocumentationNode-luokka sisältää tallennetun dokumentaatiotekstin ja tiedon mitä kieltä on tallennettu. DocumentationLibraryNode sisältää System.Collections.ObjectModel.ObservableCollection-tyyppisen [44] listan DocumentationNode-olioita. Lista on toteutettu ObservableCollection-tyypillä, sillä sen toteuttamien ominaisuuksien ansiosta käyttöliittymä on helpompi toteuttaa reagoimaan kokoelmassa tapahtuviin muutoksiin. Tällä luokalla voidaan myös asettaa ja hakea dokumentaatioteksti halutulla kielellä sekä tarkistaa onko dokumentaatiota tallennettu. Metodien ja luettelotyyppien dokumentointiin on toteutettu omat luokkansa, sillä dokumentaatioteksti tulee saada tallennettua jokaiselle rajapintaversiolle.



Kuva 4.5. *Dokumentointiin käytettävät rakenteet.*

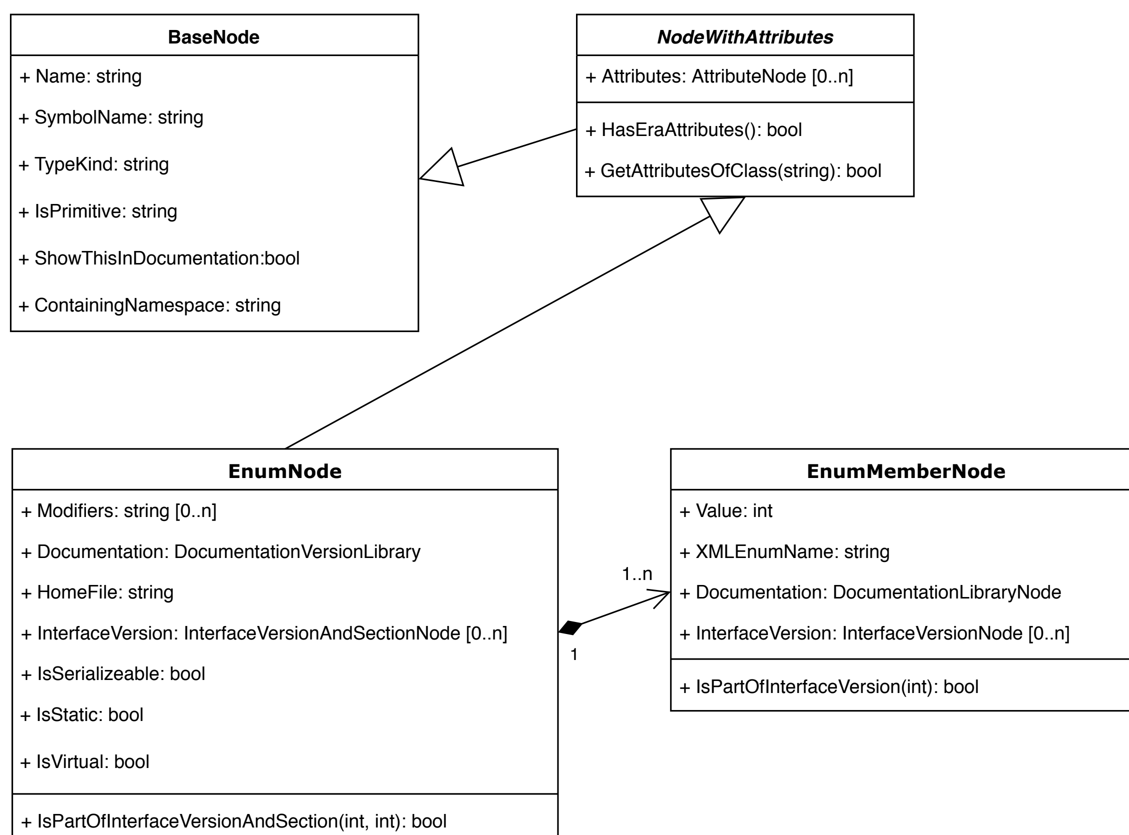
Luokan dokumentaation tallentamiseen on toteutettu ClassDocumentationLibrary-luokka, johon on mahdollista tallentaa eri dokumentaatio eri rajapintateknologialle, rajapintaversiolle ja dokumentaatiokielelle. Tallennettava rakenne on toteutettu sisäkkäisillä kirjastorakenteilla ja DocumentationLibraryNode-luokalla. DocumentationVersionLibrary-luokan dokumentaatio on järjestetty kirjasto-rakenteeseen, jonka avaimena on rajapintaversio ja arvona DocumentationNode-olio. Dokumentoinnin tallentamiseen toteutut rakenteet

näkyvät UML-kaaviona kuvassa 4.5.

4.3.4 Luettelotyyppien tallentaminen

Luettelotyypin tietojen tallentaminen tapahtuu EnumNode-luokkaan. EnumNode-luokka sisältää listan luettelotyypin määreistä, listan attribuuttiluokista ja listan luettelotyypin jäsenistä. Luettelotyypin jäsenen tieto tallennetaan EnumMemberNode-luokkaan, joka sisältää arvon, DocumentationLibraryNode-luokkaan tallennettavan dokumentaation, attribuuttilistauksen sekä ominaisuuden sarjallistamisessa käytetyn XML-nimen.

Luettelotyyppi kuuluu johonkin rajapintaversioon ja -osaan, joten sillä on listaus InterfaceVersionAndSectionNode-olioita. Näitä voi olla useita, sillä luettelotyyppi voi kuulua esimerkiksi rajapinnan yleiseen osaan versioissa 1.1 - 1.6 ja rajapinnan arkisto-osaan versioissa 1.7 - 1.8. Myös kaikille luettelotyypin jäsenille on listaus InterfaceVersionNode-oliota, joista voidaan päätellä mihin rajapintaversioihin kukin luettelotyypin jäsen kuuluu. Yleensä luettelotyyppiin lisätään jäseniä, mutta vanhoja arvoja ei poisteta.

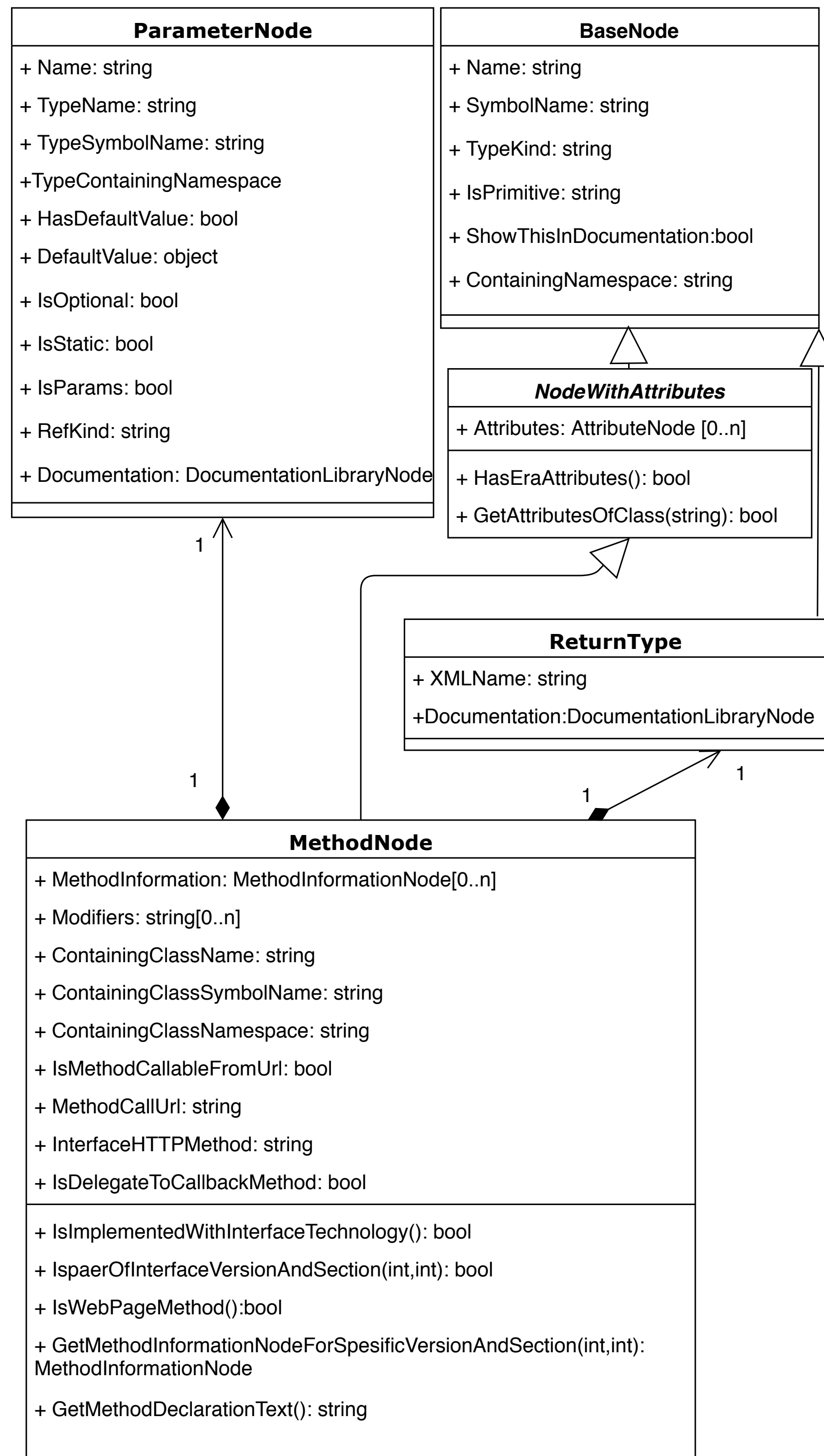


Kuva 4.6. *Luettelotyypin tallentamiseen käytettävät rakenteet.*

Kuvassa 4.6 nähdään luettelotyypin tietojen tallentamiseen käytettävät rakenteet EnumNode sekä EnumMemberNode. Jokaiselle luettelotyypin arvolle muodostetaan oma EnumMemberNode-olio.

4.3.5 Metodien tallentaminen

Rajapinnasta luetun metodin informaatio tallennetaan MethodNode-luokkaan.



Kuva 4.7. *Metodin tallentamiseen käytettävät rakenteet.*

Kuvassa 4.7 näkyvä MethodNode-luokka tallentaa funktion tiedot. Kuvassa esiintyvän MethodInformationNode-rakenteen tarkempi kuvaus löytyy alaluvusta 4.3.2 MethodInformationNode-rakenteita voi olla useita, riippuen siitä, onko metodi muuttunut eri rajapinta-

versioiden välillä.

4.3.6 Luokkien tallentaminen

Luokan tallentamiseen käytetään ClassNode-luokkaa. Luokka periytyy abstraktista NodeWithAttributes-luokasta.

Kuva 4.8. *Luokan tallentamiseen käytettävät rakenteet.*

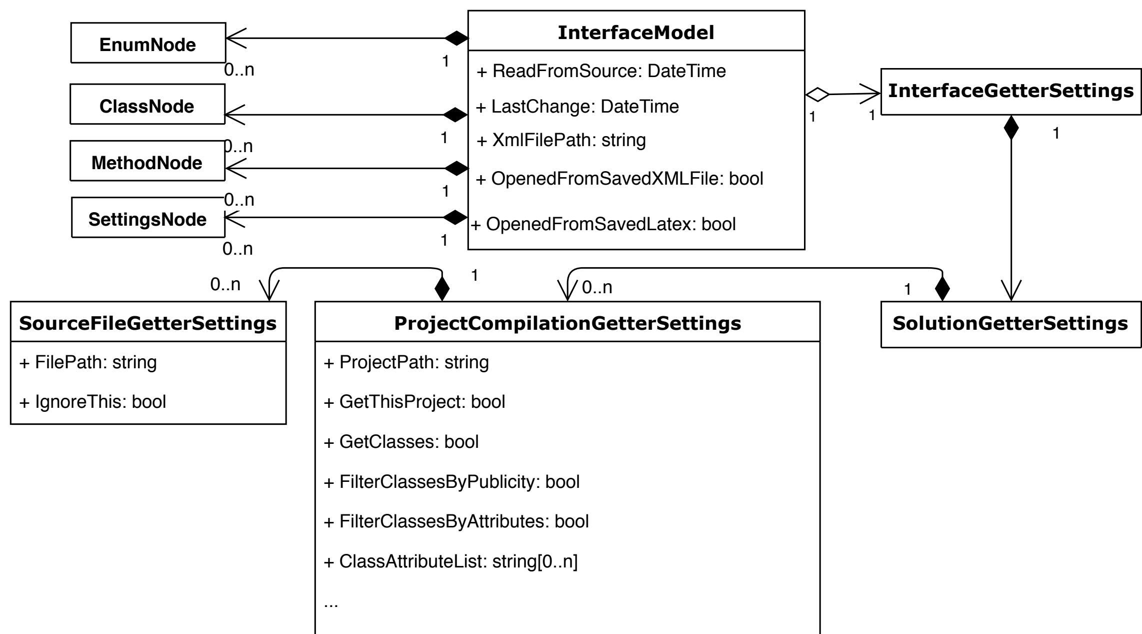
Luokan tietojen tallentamiseen tarkoitetut luokat ClassNode, PropertyNode sekä FieldNode ovat esiteltynä kuvassa 4.8. PropertyNode-luokka kuvaa luokan yhden ominaisuuden tietoja. FieldNode-luokka kuvaa luokan kentän ominaisuuksia. XMLPropertyCompulsion-luokan tarkoitus on tallentaa tieto siitä, tuleeko luokan ominaisuuden olla sarjallistettuna XML-rajapinnan käyttöä varten. Pakollisuustiedot saadaan selville ominaisuuksille kirjatuista DocumentationXmlRequestParameter- ja DocumentationXmlResponseParameter-

attribuuteista.

4.3.7 Rajapinnan tietomalli

Edellä esitettyjen luokkarakenteiden lisäksi tietomalliin kuuluu rakenteita, joilla kerrotaan jäsentäjämoduulille, mitä lähdekooditiedostoja jäsentäjän tulee analysoida sekä mitä rakenteita lähdekoodista tulee etsiä ja lisätä tietorakenteeseen.

Dokumentoitavan järjestelmän .NET-rajapintaan kuuluu asetuksia, joita halutaan näyttää dokumentaatiossa. Dokumentoitavat asetukset ovat luokan ominaisuuksia, jotka ovat primitiivisiä tyyppejä tai vakioarvotyyppisiä muuttujia, joiden arvo on helppo esittää dokumentaatiossa merkkijonona.



Kuva 4.9. *Rajapinnan tietomallia kuvaava rakenne.*

Kuvassa 4.9 on koko dokumentoitavan rajapinnan rakenteita kuvaava InterfaceModel-luokka. Kuvassa näkyy myös jäsentäjäasetuksia sisältävä InterfaceGetterSettings-luokka. Yksinkertaistettuna InterfaceModel-luokka on vain kokoelma luettelotyyppejä, luokkia, metodeja sekä asetuksia.

4.4 C#-moduuli

C#-jäsentäjämoduulin toimintaan kuuluu Visual Studion ratkaisun kääntäminen ja haluttujen lähdetiedostojen lukeminen ja analysointi. Ratkaisun analysoinnissa lähdetään liikkeelle siitä, että koko ratkaisu ladataan Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace-muuttujaan [43] ja käännetään aina, vaikka kaikkia lähdetiedostoja ei välttämättä haluttaisi analysoida. Tämä tehdään siksi, että dokumentoitavassa lähdekoodissa on paljon viittauksia ratkaisun sisäisiin projekteihin, ja lähes kaikkia projekteja käytetään viit-

teenä osissa, joissa rajapinta on määritelty. Koko ratkaisu kääntämällä käyttäen Roslyn Workspace APIa ei ole mahdollista syntyä tilannetta, että toimiva projekti ei käänny puuttuvien käännöstiedostojen takia. Jos koko ratkaisua ei käännettäisi, semanttisen mallin muodostaminen ei onnistuisi puuttuvien viitteiden takia.

Jäsentäjämoduulin toiminnan tuloksena palautetaan tietomallimoduulissa määritellyn InterfaceModel-luokan mukainen olio, joka kuvaa rajapinnassa sijaitsevat luettelotyypit, luokat, metodit ja rajapinnat. Analysoitaessa lähdekoodia jäsentäjälle annetaan parametrina tietomallimoduulissa määritelty InterfaceGetterSettings-tyyppinen olio, joka sisältää tietoa, mitkä ratkaisut, projektit ja tiedostot analysoidaan, sekä mitä rakenteita halutaan lukea muodostettavaan tietomalliin. Muuttujassa voidaan antaa myös asetuksia, kuinka rajata haluttuja luettelotyyppejä, luokkia ja metodeja esimerkiksi niiden määreiden, attribuuttiluokkien tai paluuarvojen avulla. InterfaceGetterSettings-oliossa ilmaistut lähdekooditiedostot käydään läpi yksi kerrallaan ja muodostetaan tiedostosta Roslynin avulla syntaksipuu. Tämän jälkeen syntaksipuun ja ratkaisun käännöksen avulla muodostetaan yhden tiedoston semanttinen malli Roslynin avulla.

Ratkaisu ladataan ensin Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace-muuttujaan kutsumalla OpenSolutionAsync-kutsua ja antamalla parametrina tiedostopolku ladattavaan ratkaisuun. Paluuarvona saatavan Microsoft.CodeAnalysis.Solution-muuttujan Projects-ominaisuutena on lista Microsoft.CodeAnalysis.Project-oliota [45]. Project-oliolta voidaan saada Microsoft.CodeAnalysis.Compilation-olio [39]. Project-oliolta voidaan pyytää luettelo Microsoft.CodeAnalysis.Document-olioita [40], joista projekti koostuu. Yksi document-olio kuvaa yhtä lähdetiedostoa. Document-oliolta voidaan pyytää MicrosoftCodeAnalysis.SyntaxTree-olio [50], eli syntaksipuu kutsumalla Document-olion GetSyntaxTreeAsync-kutsua.

Jäsentäjämoduuli lukee lähdekoodia tiedosto kerrallaan ja muodostaa ensin CompilationUnitNode-olion, jonka on tarkoitus sisällyttää yhden käännettävän tiedoston tiedot. CompilationUnitNode-luokkaa käytetään jäsentämisen apuvälineenä, ja kun kaikki halutut tiedon on luettu, siirretään CompilationUnitNode-luokan sisältämät luokat, luettelotyypit ja metodit InterfaceModel-olioon, joka on jäsentäjä moduulin tuotoksena saatava rajapintaa kuvaava malli. CompilationUnitNode-luokan muodostamiseksi annetaan parametrina SyntaxTree-olio sekä Microsoft.CodeAnalysis.SemanticModel-olio [48], joka saadaan Compilation-oliolta kutsumalla GetSemanticModel-funktiota parametrina valittu SyntaxTree-olio.

Lähdekooditiedoston analysoinnissa käydään läpi syntaksipuun solmuja juurisolmusta alaspäin kohti puun lehtiä. C#-jäsentäjämoduulissa toteutetussa tiedoston analysoinnissa käytetään joukkoa funktioita, jotka ottavat parametrina solmun. Tästä solmusta voidaan hakea Roslynin metodeilla halutun tyyppisiä jälkeläissolmuja (engl. descendant node), joita tarkastellaan. Kaikille tietomallimoduulissa esitellyille rajapintaa kuvaaville luokille on oma analysointifunktionsa C#-jäsentäjämoduulissa. Jokainen jäsentäjämoduulin funktio palauttaa listan tarkasteluun valitun ns. isäntäsolmun jälkeläisenä olevista tietyntyyppisistä rakenteista.

Esimerkiksi luokan jäsentämiseen kutsutaan ensin luokkien rakentamiseen tarkoitettua funktiota, joka etsii kaikki luokkien määrittelemiseen käytetyt solmut sille parametrina annetun isäntäsolmun alta. Kullekin löydetylle luokan määrittelysolmulle, joka on tyyppiä Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax [38] Roslynin muodostamassa syntaksipuussa, haetaan symboli semanttisesta mallista. Sitten symbolista tarkistetaan, onko luokalla attribuutteja, joista ollaan kiinnostuneita. Jos ei, lopetetaan kyseisen solmun käsittely. Seuraavaksi rakennetaan luokkaa kuvaava ClassNode. Luokalle haetaan sitten lista ominaisuuksia ja ominaisuuksia rakentavalle funktiolle annetaan parametrina tutkittavan luokan solmu. Ominaisuuksia rakentava funktio hakee kaikki ominaisuuksia määrittelevät solmut luokkasolmun alta ja hakee semanttisesta mallista ominaisuutta kuvaavan symbolin ja rakentaa sen tietojen avulla kutakin kuvaavan PropertyNode-luokan.

Jäsentäjän toiminnan periaatteena on aina etsiä valitun isäntäsolmun jälkeläisistä solmuja, jotka määrittelevät halutun tyyppistä rakennetta. Näille jälkeläisille kutsutaan niiden tyypin perusteella näiden tyyppiä vastaavaa tietomalliluokkaa rakentavaa funktiota. Sitten tämä funktio kutsuu jälleen yksi jälkeläissolmu isäntäsolmuksi valittuna seuraavaa jäsentäjäfunktiota. Seuraavassa esimerkissä mainitut luokat löytyvät Microsoft.CodeAnalysis.CSharp.Syntax-nimiavaruudesta [42]. Esimerkiksi tälläinen kutsuketju voi mennä niin, että ensin etsitään NamespaceDeclarationSyntax-solmuja nimiavaruuksien etsimiseen kirjoitetulla funktiolla, tämän jälkeen etsitään yhden NamespaceDeclarationSyntax-solmun jälkeläisistä ClassDeclarationSyntax-solmuja luokkien etsimiseen tarkoitetulla funktiolla, tämän ClassDeclarationSyntax-solmun jälkeläisistä etsitään esimerkiksi MethodDeclarationSyntax-solmuja luokan metodien etsimiseen tarkoitetulla funktiolla. Tämän MethodDeclarationSyntax-solmun alta voidaan etsiä vaikkapa ParameterSyntax-solmuja parametrien etsimiseen kirjoitetulla funktiolla. Näin edetään kunnes on käyty läpi kaikki kiinnostavat solmutyypit jokaisen kiinnostavan solmutyypin alta. Jokainen jäsentäjäfunktio etsii ja lisää rakentamansa olion tai listan olioita tietorakenteeseen.

4.5 Tietomallien vertaaminen

Yksi työkalun tärkeimmistä ominaisuuksista on mahdollisuus vertailla tietomalleja keskenään. Vertailtavaksi voidaan valita kaksi tietomallia kerrallaan. Vertailun tulokset näytetään käyttöliittymässä niin, että vertaillut rakenteet esitetään vierekkäin ja eroavaisuudet merkitään erilaisin värein.

Vertailussa täytyy voida valita, mitä ominaisuuksia tietomallista vertaillaan, sillä vertailussa voidaan olla kiinnostuneita esimerkiksi vain tietyn rajapintaversion lähdekoodin ja dokumentaation erosta. Koska tietomalli voidaan ladata ohjelmaan joko LaTeX-dokumentista, lähdekoodista tai aikaisemmin tallennetusta sarjallistetusta XML-tietomallista, voi näiden tietosisältö erota toisistaan, vaikka ne kuvaisivat samaa rajapintaa. LaTeX-dokumentaatiosta muodostettu tietomalli sisältää vähemmän tietoa kuin lähdekoodista luettu tietomalli, koska dokumentaatiossa ei haluta esittää lukijalle epärelevanttia

tietoa.

4.5.1 Vertailutekniikan toteutustavan valinta

Vertailtavat tietomallit ovat esitellyn tietomallimoduulin InterfaceModel-luokkarakenteita. Vaikka luokat ovat C#-kielen viittaustyyppejä, ei haluta vertailla luokkien osoitteita muistissa, vaan luokkien ominaisuuksien arvoja. Koska kaikki vertailtavat luokat ovat tietomallimoduulista, vertailijan toimintaa suunnitellessa voitiin siis tietää, että kaikki vertailtavat luokat ovat ennalta tunnettuja ja että luokkarakenteisiin voidaan myös tehdä muutoksia vertailun helpottamiseksi. Vertailussa täytyy vertailla keskenään vain olioita, joiden tyyppi on sama, sillä rajapinnan samankaltaisuuden tarkastelussa ei ole järkevää vertailla esimerkiksi luettelotyyppiä kuvaavaa EnumNode-luokkaa ja luokan ominaisuutta kuvaavaa PropertyNode-luokkaa, vaikka luokan ominaisuus olisi tyypiltään luettelotyypin instanssi, jota EnumNode-luokassa kuvataan. Vertailtavista luokista täytyy saada tietää vertailun tuloksena, ovatko luokat samanlaisia sekä listata, mitkä ovat olioiden erot. Kuitenkaan vertailtavista luokista ei aina tahdota vertailla kaikkia mahdollisia ominaisuuksia ja aliluokkia. Vertailua halutaan usein rajoittaa koskemaan valitun rajapintaversion mukaisia tietoja. Tällöin vertailijan tulee tunnistaa, mitä ominaisuuksia luokasta tulee vertailla. Käyttökokemuksen kannalta vertailijan toimintaa olisi hyvä olla mahdollista muuttaa käyttöliittymän kautta valittavilla asetuksilla.

Toteutustapaa harkittiin kolmen eri vaihtoehdon väliltä. Ensimmäinen vaihtoehto oli luokan sarjallistetun tekstin vertaaminen toiseen sarjallistettuun tekstiin. Sarjallistaminen voidaan toteuttaa C#-kielen valmiilla XML-sarjallistamiskirjastolla ja sarjallistaa kaikki luokan ominaisuudet tekstiksi. Saatuja merkkijonoja voidaan sitten verrata merkkijonojen vertailemiseen tarkoitetulla työkalulla, esimerkiksi pisimmän yhteisen osajonon (engl. longest common subsequence) etsimiseen perustuvalla algoritmilla. Vertailtaessa rakenteita sarjallistetussa XML-dokumentissa, tulisi ensin paikantaa osat, joita tahdotaan vertailla. Paikantaminen on teknisesti helppoa, sillä XML-dokumentin lukeminen ja merkittyjen rakenteiden etsiminen XML-jäsentäjällä on toteutettu monilla valmiilla työkaluilla. Sarjallistaminen on helppo toteuttaa C#-kielen valmiilla kirjastoilla, mutta sarjallistamiseen perustuva ratkaisu luultavasti voi olla hidas, sillä sarjallistetun eRA-järjestelmän rajapinnasta ladatun tietomallin koko XML-tiedostona on yli 45 000 MB ja sarjallistetussa XML-tiedostossa yli 900 000 riviä ilman siihen lisättyjä kommenttitekstejä. Molemmat tietomallit joudutaan ensin sarjallistamaan merkkijonoiksi ja sitten vertailemaan ja mahdollisesti jäsentemään takaisin tietomalliksi, jos tietomallien tietoja tahdotaan muokata.

Toinen harkittu vaihtoehto vertailun toteutustavaksi oli kirjoittaa vertailtaville rakenteille C#-kielen määrittelemä System.Runtime.IEquatable<t>-rajapinnan [41] toteuttava Equals-funktio. Equals-funktiot täytyisi itse toteuttaa niin, että jokaiselle luokalle täytyisi määrittää oma toteutuksensa. Suurempien rakenteiden vertailu voitaisiin sitten suorittaa ketjuttamalla Equals-funktioiden kutsuja niin, että isäntäluokan Equals-funktio kutsuu aliluokan Equals-funktiota. Equals-funktion avulla olisi helppoa huomata, mitkä rakenteet

eivät vastaa toisiaan, mutta sen avulla ei voida suoraan palauttaa eroja, sillä Equals-funktion paluuarvona on vain totuusarvo. Jos jokaiselle luokan ominaisuudelle ei ole määritelty omaa Equals-funktiota, tuloksena saadaan vain, että oliot eivät vastaa toisiaan, mutta ei tiedetä mistä ominaisuudesta eroavuus johtuu. Equatable-rajapintaa toteuttamalla ei ole mahdollista reagoida käyttöliittymän kautta muokattaviin arvoihin, sillä Equals-metodi ei ota muita parametreja kuin vertailtavan luokan instanssin. Pelkkä rajapinnan toteuttaminen ei siis riittäisi, vaan tämän lisäksi jouduttaisiin kuitenkin muodostamaan luokka, joka toteuttaisi luokkien vertailun valituilla asetuksilla, jos rakenteet eivät olisi täydellisesti samanlaiset.

Kolmas vaihtoehtoinen toteutustapa oli hyödyntää reflektiota vertailualgoritmin teossa ja muodostaa oma vertailuluokka, joka toteuttaa vertailun käyttöliittymässä valituilla asetuksilla. Vertailun tulee palauttaa tietorakenne, josta käy ilmi, missä rakenteissa on eroja ja missä rakenteen alirakenteissa erot sijaitsevat. Reflektiota hyödyntämällä ei tarvitse kirjoittaa jokaiselle vertailtavalle luokalle omaa vertailufunktiota, sillä reflektion avulla voidaan lukea kaikki luokkaan kuuluvat ominaisuudet ja vertailla niiden arvoja. Koska reflektiolla voidaan tarkastella myös ominaisuuksien nimiä ja tyyppejä, niin reflektioon perustuvan ratkaisun avulla on helppo muodostaa vertailijasta muokkautuva, niin että asetuksissa voidaan listata vertailtavan luokan ominaisuuksien nimiä, joita halutaan vertailla.

Koska reflektioon perustuva vertailu vaikutti helpoiten muutettavissa olevalta vaihtoehdolta, päätettiin vertailu toteuttaa reflektiota hyödyntävillä algoritmeilla. Reflektion haittapuoleksi tunnistettiin se, että metadatan lataaminen kirjaston käännöksestä vie ylimääräistä muistia ja ominaisuuden arvon hakeminen muistista reflektion avulla on hitaampaa kuin arvojen vertaileminen suoraan tietomallista. Toteutettava sovellus ei kuitenkaan vaadi suurta toimintanopeutta ja käyttäjälle on hyväksyttävää muutamien sekuntien odotusaika harvoin toistuvalle vertailuoperaatiolle, joten hieman hitaampaa toiminta-aikaa ei pidetty esteenä.

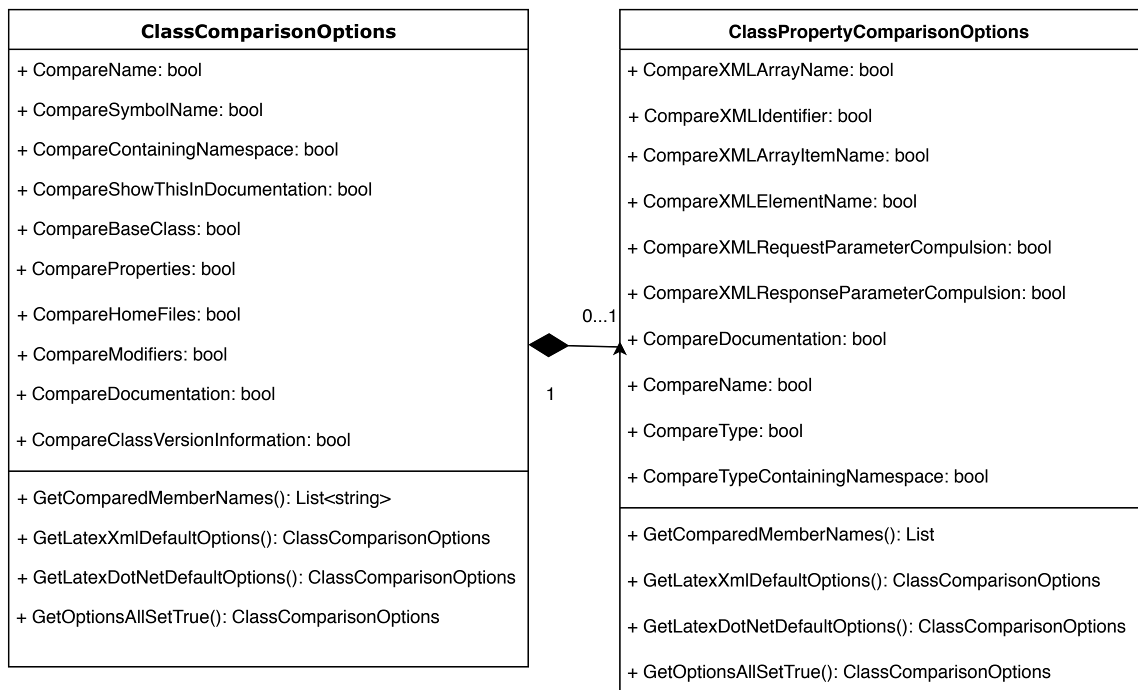
4.5.2 Vertailijan toiminta

Vertailussa päätettiin hyödyntää Kellerman Softwaren kirjoittamaa Compare Net Objects -kirjastoa [20]. Kirjastoa päätettiin hyödyntää vertailussa, koska sen avulla saatiin käyttöön valmiiksi toteutetut tulosrakenteet, vertailuun soveltuva rajapinta sekä mahdollisuus mukauttaa vertailijan toimintaa asetuksilla. Kirjastoon on tullut uusia korjauksia ja parannuksia säännöllisesti. Kirjastoa on myös ladattu yli kaksi miljoona kertaa .NET pakettien hallintajärjestelmän kautta NuGet-pakettina. Tästä voitiin päätellä, että kirjastolla oli paljon testattua käyttöä ja kirjastoa parannellaan ja ylläpidetään.

Kirjaston avulla voidaan vertailla kahta samantyyppistä oliota ja listata luokkien eroavaisuuksia. Olion sisältämien ominaisuuksien vertailu onnistuu myös silloin, kun nämä ovat luokkia tai listoja, jotka sisältävät olioita. Oletuksena kirjasto vertailee valitun luokan kaikkia ominaisuuksia. Vertailu voidaan asettaa loppumaan, kun määritelty lukumäärä eroavaisuuksia on löytynyt. Kirjasto mahdollistaa myös vertailufunktion määrittelemi-

sen halutulle luokalle. Osa luokista on mahdollista vertailla oletusvertailun avulla, mutta osalle luokista täytyi määritellä omat vertailufunktionsa. Yksi tällainen luokka oli dokumentaatiota sisältävä ClassDocumentationLibrary-luokka. Tälle luokalle toteutettiin KellermanSoftware.CompareNetObjects.TypeComparers.BaseTypeComparer-luokasta [19] periytetty luokka, joka vertailee pelkästään halutun teknologian, version ja dokumentaatiokielen mukaista dokumentaatiota ja lisää eroavaisuulistaan uuden merkinnän vain silloin, kun tietty dokumentaatio ei vastaa toista haluttua dokumentaatiota.

Vertailuasetuksille toteutettiin ComparisonOptions-luokka, joka sisältää tiedot kaikista ominaisuuksista, joita tahdotaan vertailla. Ohjelman käyttöliittymässä voidaan muokata ComparisonOptions-luokan arvoja valitsemalla vertailtavia ominaisuuksia päälle tai pois päältä. Kuvassa 4.10 nähdään luokkaa kuvaavan ClassNode-luokan vertailuasetukset. Valitsemalla totuusarvomuutujia arvoon tosi käyttäjä voi lisätä luokan ominaisuuksia vertailtaviin ominaisuuksiin.



Kuva 4.10. *Luokkien vertailun mukauttamiseen käytetyt asetukset, jotka koostuvat ClassComparisonOptions- sekä ClassPropertyComparisonOptions-luokista.*

GetComparedMemberNames-funktiolla voidaan palauttaa lista valituista ominaisuuksien nimistä, joita vertaillaan. Vertailtavat ominaisuuksien nimet palautetaan Compare Net Objects -kirjaston KellermanSoftware.CompareNetObjects.CompareConfig-tyypin memberToInclude-ominaisuudelle, jolla asetetaan vertailtavien ominaisuuksien nimet.

Vertailtavat tietomallit ladataan ohjelman muistiin vertailua varten. Toteutetulle CompareInterfaceModel-funktiolle annetaan parametrina tietomallit ja ComparisonOptions-olio vertailua varten. Vertailtavat oliot tunnistetaan samaa rakennetta kuvaaviksi olioksi InterfaceModel-tietomallista näiden nimellä. Esimerkiksi ClassNode-tyyppiset oliot ovat InterfaceModel-rakenteessa yhdessä listassa. Jos ClassNode-olita, jonka Name-

ominaisuus on sama, ei löydy toisen tietomallin listasta, ei vertailua ClassNode-oliolle suoriteta. Saman nimen lisäksi olioiden täytyy kuulua valittuun rajapintaversioon, teknologiaan sekä rajapintaosaan. Jos luokan ominaisuuksia vertaillaan, tulee ominaisuuksien tunnistaminen tehdä eri ominaisuuksilla riippuen siitä, millä tavalla InterfaceModel-tietomalli on muodostettu. Ladattaessa tietomalli LaTeX-dokumentaatiosta, joka kuvaa XML-rajapintaa, luokan ominaisuuksien tyyppejä tai nimiä ei ole ilmoitettu, vaan ominaisuudet tunnistetaan XMLIdentificationString-ominaisuuden avulla. XMLIdentificationString on sarjallistetun luokan ominaisuuden XML-elementin tunniste, joka on yksillöllinen luokan sisällä. Vertailtaessa .NET-rajapintaa kuvaavaa LaTeX-dokumentaatiota voidaan ominaisuudet tunnistaa näiden nimien avulla. Luokan ominaisuuksia vertailtaessa täytyy myös huomioida, että vertailtavat ominaisuutta kuvaavat PropertyNode-oliot kuuluvat vertailtavaan rajapintaosaan.

Esimerkiksi rajapintaluokkaa kuvaavan ClassNode-rakenteen vertailu toteutetaan kahdessa osassa: ensin vertaillaan luokan yksinkertaisia tietoja kuten nimeä, nimiavaruutta, määreitä, lähdetiedostoa ja dokumentaatiota. Edellä mainittuja vertaillaan, jos ne ovat valittuina ClassComparisonOptions-luokassa. Tämän jälkeen vertaillaan jokaista valittuun rajapintaversioon kuuluvaa luokan ominaisuutta. Luokan ominaisuuksien vertailun asetukset ovat ClassPropertyComparisonOptions-luokassa.

Vertailu suoritetaan Compare Net Objects -kirjaston KellermanSoftware.CompareNetObjects.CompareLogic -nimiavaruudesta löytyvällä Compare(object object1, object object2) -funktiolla [21]. Vertailun tuloksena palautetaan KellermanSoftware.CompareNetObjects.ComparisonResult-tyypin olio [18]. ComparisonResult-rakenteella on Differences-ominaisuus, joka on tyypiltään lista KellermanSoftware.CompareNetObjects.Difference-oliota. Jokaista löydettyä eroavaisuutta kohden vertailutoiminto tuottaa Difference-olion, joka sisältää esimerkiksi vertailtavien olioiden arvot, tyypit, ominaisuuden nimen ja viittauksen olioihin jotka sisältävät vertailtavat oliot.

DiffenrencesInClasses

+ Classes1: List<ClassNode>
+ Classes2: List<ClassNode>
+ ClassesOnlyIn1: List<ClassNode>
+ ClassesOnlyIn2: List<ClassNode>
+ ClassComparisonResults: Dictionary<string,ClassComparisonResult>
+ NotEqualClassPropertyNames :Dictionary<string<List<string>>

PropertyComparisonResult

+ PropertiesIdentifiedBy: string
+ Property1: PropertyNode
+ Property2: PropertyNode
+ Result: ComparisonResult
+ ComparedMembers: List<string>
+ NeedToHighlight: List<string>

ClassComparisonResult

+ ClassName: string
+ BasicInfoComparedMembers: List<string>
+ BasicInfoComparisonResults: ComparisonResult
+ NeedToHighlight
+ PropertiesOnlyOnClass1: List<PropertyNode>
+ PropertiesOnlyOnClass2: List<PropertyNode>
+ PropertyComparisonResults: List<propertyComparisonResult>

Kuva 4.11. *Rajapintaluokkien vertailun tulokset tallentava luokka DifferencesInClasses ja sen käyttämät luokat ClassComparisonResult sekä PropertyComparisonResult.*

Vertailtaessa ClassNode-olioita vertailun tuloksista muodostetaan DifferencesInClasses-luokan instanssi, jonka tarkoituksena on säilyttää vertailun tulokset käytäjälle helposti näytettävässä muodossa. Kuvassa 4.11 mainittu ClassComparisonResult-luokka on Compare Net Objects -kirjastosta peräisin oleva luokka. Jokaista vertailtua oliopariä varten lisätään DifferencesInClasses-olion ClassComparisonResults-kokoelmaan uusi ClassComparisonResult-olio. ClassComparisonResult sisältää ClassNode-olioden vertailun tulokset, sekä listan PropertyComparisonResult-oliosta, joita kirjataan vertailtaessa PropertyNode-oliota. Luokkien kentille ei toteutettu vertailua, sillä dokumentoitavassa rajapinnassa ei luokilla ole kenttiä, vaan pelkkiä ominaisuuksia.

Metodien ja luettelotyyppien vertailu toteutetaan samankaltaisesti kuin edellä esitelty luokkien vertailu. Näitä vertailuja varten on toteutettu omat vertailun asetuksia määrittävät rakenteensa sekä vertailun tulokset tallentavat rakenteensa.

4.6 LaTeX-moduuli

LaTeX-moduuli on vastuussa LaTeX-muotoisen dokumentaation lukemisesta tietomalliksi sekä tietomallin sarjallistamisesta dokumentaatioksi. LaTeX-moduuli on toteutettu C#-kielellä ja käännetty ohjelmistokirjastoksi, jota työkalun käyttöliittymämoduuli käyttää. LaTeX-moduuli käyttää tietomallimoduulissa määriteltyjä luokkia.

4.6.1 Dokumentaation jäsentäminen

Jäsennettäessä dokumentaatiota tulee kutsua kirjastoon toteututun LatexParser-luokan tarjoamaa GetInterfaceModelFromLatex-funktiota, jolle annetaan parametreina käyttä-

jän käyttöliittymässä valitsemat asetukset: rajapinnan versio, rajapintateknologia, dokumentaation kieli, sekä tiedostopolku LaTeX-dokumenttiin. Valitun rajapintateknologian perusteella kutsutaan kyseiselle teknologialle kirjoitettua jäsentäjäfunktiota. Kukin jäsentäjäfunktio lukee ensin dokumentaatiotekstin ja pilkkoo sen säännöllisiin lausekkeisiin ja merkkijonoluokan funktioihin perustuvilla apufunktioilla pieniin osiin. Ensin koko dokumentaatio pilkotaan lukuihin, jotka kuvaavat kutakin rajapintaosaa. Kaikki luvut käydään läpi yksi kerrallaan ja luvuista etsitään vakioarvoja, luokkia, metodeja, asetuksia sekä web-sivuja kuvaavat alaluvut. Rakenteita kuvaavat tekstikappaleet on merkitty ”\label”-tunnisteella, jonka hakasulkujen väliin sijoitetusta arvosta voidaan päätellä, minkälaista rakennetta tekstikappale kuvaa. Kun osio on tunnistettu, kutsutaan tekstikappaleen lukemiseen tarkoitettua funktiota, joka rakentaa yhden kappaleen perusteella kyseistä tyyppiä kuvaavan luokan instanssin, joka on määritelty tietomallimoduulissa. Kaikki dokumentaatioon kirjoitettu tieto on merkitty riveittäin tunnisteilla, joista voidaan päätellä, minkälaista dataa dokumenttiin kirjoitetut rivit pitävät sisällään.

Jäsennysvaiheessa dokumentista siistitään dokumentin sisäisiin kappaleisiin ja lukuihin viittaavat linkitykset, sillä niitä ei haluta tallentaa tietomallin dokumentaatiokenttiin. Linkkejä voi olla vain tietyllä tunnisteella merkityn tekstin sisällä.

Koska dokumentaatiosta on haluttu piilottaa kaikki epärelevantti data lukijalta, ei LaTeX-dokumentista muodostettu tietomalli sisällä enää kaikkea sitä tietoa, mitä tietomalliin on alun perin tallennettu. Esimerkiksi LaTeX-tiedostosta jäsennetty tietomalli sisältää vain yhteen rajapintaversioon kuuluvat tiedot yhdellä luonnollisella kielellä, sillä yksi LaTeX-dokumentaatiotiedosto sisältää dokumentaation yhdestä rajapintaversiosta yhdellä kielellä.

LaTeX-moduulilla voidaan myös lisätä valmiiseen rajapinnasta ladattuun tietomalliin kommenttitekstit samannimisille rakenteille, joita dokumentaatiosta löytyy. Tämä ominaisuus on toimii kahdessa osassa niin, että LaTeX-dokumentti jäsennetään rajapinnan tietomalliksi, ja tästä tietomallista kopioidaan rakenteiden kuvaustekstit kaikkiin toisen tietomallin samannimiseille tietorakenteille. Jos samannimisiä rakenteita ei löydetä, ei niitä lisätä muokkauksen kohteena olevaan tietomalliin. Tämä ominaisuus on hyödyllinen silloin, kun muodostetaan uuden rajapinnan dokumentaatiota. Tällöin edellisen dokumentaatioversion tekstit saadaan kopioitua rajapinnasta päivitettyyn tietomalliin. Tekstit päivittyvät kaikille niille rakenteille ja niiden ominaisuuksille, jotka ovat säilyneet samannimisenä edellisestä versiosta. Usein nämä vanhat kuvaukset toimivat hyvin pohjana uudelle kuvaukselle tai niitä ei välttämättä tarvitse muuttaa ollenkaan.

4.6.2 Dokumentaation generointi

Dokumentaation generoinnissa LaTeX-moduulille annetaan valmis InterfaceModel-olio, tieto dokumentaation kielestä, rajapinnan teknologiasta, rajapinnan versiosta sekä tiedostopolku tallennettavalle dokumentille. Kaikille rajapinnan teknologioille on omat dokumentointifunktionsa, jotka toimivat samankaltaisesti. Dokumenttia kirjoitetaan luku kerral-

laan, niin että luvun sisälle tulevat alaluvuiksi metodit, mahdolliset web-sivut, mahdolliset asetukset, vakioarvot, ja luokat. Kunkin luvun kohdalla funktio hakee InterfaceModel-tietomallista lukua vastaavan rajapinnan osaan kuuluvat rakenteet, ja kutsuu näiden sarjallistamiseen tarkoitettuja funktiota. Rajapinnan metodit kirjoitetaan dokumentaatioon niiden sisältämän järjestysnumeron mukaisessa järjestyksessä, joka on luettu lähdekoodin attribuutista. Rakenteet, joilla ei ole järjestysnumeroa attribuuttiluokissa kirjoitetaan alalukujen sisään aakkosjärjestyksessä.

Tietomallissa olevien luokkien sarjallistamista varten on toteutettu jokaista kolmea dokumentoitavaa rajapintatyyppiä varten oma sarjallistamisfunktionsa. Sarjallistamisfunktiot sijaitsevat LaTeX-moduulissa, sillä ei haluttu, että tietomallimoduuliin tulee liikaa ylimääräistä koodia. Tietomallissa sijaitsevat luokat pyrittiin näin pitämään puhtaasti tiedon varastointiin tarkoitettuna rakenteina. Sarjallistamiseen käytettiin vain merkkijonojen rakentamiseen tarkoitettuja metodeja, sillä jos tietomallin luokat olisi sarjallistettu C#-kielen System.Runtime.Serialization-nimiavaruudesta löytyvän sarjallistamiskirjaston [53] avulla, olisi sarjallistettavat ominaisuudet pitänyt merkitä attribuuteilla tietomallimoduulin luokkiin. Tämä olisi ongelma siinä vaiheessa, kun halutaan tuottaa muunkaltaista dokumentaatiota, kun sarjallistettavat ominaisuudet on jo valittu luokkiin.

4.6.3 LaTeX-dokumentin sisäisten hyperlinkkien kirjoittaminen

Linkityksellä tarkoitetaan tässä aliluvussa LaTeX-dokumentin sisäistä hypertekstilinkkiä, jossa dokumentin PDF-tiedostoa tarkastellessa käyttäjä voi painaa linkin sisältämää tekstiä, jolloin katselusovellus kohdistuu linkin osoittamaan kohtaan. LaTeX-dokumentaatiossa on käytetty Hyperref-kirjaston [62] mukaista viittaustapaa, jossa osat joihin voidaan viitata merkitään ”\label<linkin tyyppi><osan nimi> ”-tunnisteella. Linkin lisääminen tekstiin tapahtuu kirjoittamalla tekstiin ”\<linkin tyyppi>Link<osan nimi>” -tunniste. Linkin tyypistä riippuen linkki näytetään LaTeX-tiedostosta muodostettavasta PDF-tiedostossa hieman ei tavoin.

Ne kohdat tekstissä, joihin linkkejä tarvitsee kirjoittaa, käydään kirjoitusvaiheessa läpi funktiolla, joka ottaa parametrinaan kirjoitettavan merkkijonon sekä tietorakenteen, johon on kerätty kaikki tietomallin sisältämät rakenteiden nimet kokoelmina, joissa yksi kokoelma sisältää vain yhden tyyppisiä olioita. Merkkijono pilkotaan osiin säännöllisen lausekkeen avulla, joka katkaisee merkkijonon merkkien ”-”, ” ”, ”.”, ”<”, ”>” ja ”\” kohdalta. Jokaista merkkijonon osaa etsitään jokaisesta nimikokoelmasta, ja löydettäessä korvataan osa linkillä ja kirjoitetaan kaikki osat LaTeX-dokumenttiin. Lisättävän linkin tyyppi päätellään siitä, mistä kokoelmasta nimi on löytynyt.

4.7 Käyttöliittymä

Graafinen käyttöliittymä toteutettiin Windows Presentation Foundation (WPF) [26] -kirjaston avulla. Käyttöliittymäsovellus on model–view–viewmodel -suunnittelumallin (MVVM)[3] mukaisella arkkitehtuurilla toteutettu. Viestien välittämiseen kerrosten välillä käytettiin GalaSoftin MvvmLight-kirjastoa [22]. Käyttöliittymäsovellus käyttää luokkakirjastoa, C#-jäsentäjäkirjastoa sekä LaTeX-dokumentointikirjastoa. Käyttöliittymän perusnäkymä on tietomallin tutkimiseen ja dokumentointiin tarkoitettu näkymä.

Kuva 4.12. *Käyttöliittymänäkymä, jossa valittu tarkasteltavaksi yksittäinen .NET-rajapinnan metodi.*

Käyttöliittymän perusnäkymässä, josta kuvakaappaus on esitetty kuvassa 4.12, voidaan valita kerrallaan yksi rakenne listasta ja esittää sen tiedot ikkunan oikeaan laitaan avautuvassa yksityiskohtanäkymässä. Rakenteet, joita käyttöliittymässä voidaan näyttää ovat

luettelotyyppejä, luokkia, metodeja sekä asetuksia. Rakenteet valitaan näytön vasemmassa laidassa olevasta listasta. Listatut rakenteet riippuvat siitä, mitä asetuksia on valittu näytön ylälaidassa sijaitsevista pudotusvalikoista eli rajapinnan tarkasteluasetuksista, joita ovat rajapinnan dokumentointikieli, rajapintateknologia, rajapintaversio sekä rajapintaosa. Kuvassa 4.12 punaisella taustalla olevilla metodeilla on puuttellinen dokumentaatio.

Kuva 4.13. *Vertailunäkymä jossa valittu näkyviin luettelotyyppien vertailu.*

Vertailunäkymässä näytön vasemmassa osassa näkyvät vertailun tulokset listattuina ra-

kenteen tyypin mukaan. Kuvassa 4.13 on valittu luettelotyyppejä sisältävästä listasta vertailun tulos ja avattu se näytön oikealle puolelle vertailun tarkastelunälkymään, jossa vertaillut rakenteet esitetään vierekkäin. Arvot, jotka poikkeavat toisistaan esitetään punaisella taustalla. Kuvan tilanteessa vertaillaan kaikkia arvoja, joita luettelotyypin tapauksessa voidaan vertailla. Vertailtavien arvojen asetuksia voidaan muuttaa Show Comparison Options -painiketta painamalla. Vertailtavien arvojen valinnassa muutetaan kuvassa 4.10 esitetyn ClassComparisonOptions-luokan totuusarvoja, jotka määrävät vertailtavat rakenteet ja ominaisuudet.

5 ERA-JÄRJESTELMÄN INTEGRAATIOJAPINNAN DOKUMENTOINTI TYÖKALUN AVULLA

Työkalua käytettiin eRA-järjestelmän dokumentointiin, ja sen avulla muodostettiin XML-rajapintadokumentaatiosta uusi 2.2-versio. Uuden rajapintaversion dokumentointiin kulunut aika väheni verrattuna siihen, kuinka kauan aikaisemmalla manuaalisella menetelmällä on kestänyt uuden rajapintadokumentaatioversion tuottamisessa.

5.1 Attribuuttiluokkien lisääminen eRA-järjestelmän rajapintaan

eRA-järjestelmän rajapintoihin lisättiin 2.1-version mukaiset attribuutit jo aikaisemmin toteutetun 2.1-version rajapintadokumentaation pohjalta. Taulukossa 5.1 nähdään eRA-järjestelmän rajapintoihin lisättyjen attribuuttien kokonaismäärä. Lukuihin on laskettu sekä 2.1- että 2.2-version dokumentoimiseen käytettävät attribuutit, sillä osaa attribuuteista käytetään molemmissa versiossa.

Attribuutin nimi	Lisätty määrä
DocumentationClassAttribute	326
DocumentationPropertyAttribute	405
DocumentationXMLRequestParameterAttribute	734
DocumentationXmlResponseParameterAttribute	635
DocumentationXmlInterfaceMethodAttribute	101
DocumentationXMLPageAttribute	4
eRAToolkitInterfaceMethodDocumentationAttribute	72
eRAToolkitInterfaceCallbackMethodDocumentationAttribute	10
eRAToolkitInterfaceSettingDocumentationAttribute	22

Taulukko 5.1. *eRA-järjestelmän lähdekoodiin lisätyt attribuuttiluokat.*

Version 2.1 rajapinnan dokumentaatio oli jo muodostettu ennen dokumentoinityökalun kehittämistä. Silti eRA-järjestelmän rajapintaan haluttiin lisätä tätä versiota merkitsevät attribuuttiluokat, jotta dokumentointityökalun toimintaa voitiin testata.

5.2 Dokumentointityökalun toiminnan testaaminen tuottamalla vanha dokumentaatio uudelleen

Työkalun tuottaman dokumentaation oikeellisuutta testattiin toteuttamalla vanhan 2.1-rajapintaversion dokumentaatio uudestaan lähdekoodista ja vertaamalla sitä vanhan 2.1-rajapintaversion jo aikaisemmin kirjoitettuun dokumentaatioon. Vanhan rajapinnan dokumentaatio voitiin tuottaa uudestaan työkalulla kirjaamalla vanhan dokumentaation mukaiset rajapintaversiota kuvaavat attribuutit rajapinnan lähdekoodiin. Lähdekoodista tuotettiin työkalulla tietomalli, jonka tyhjiin dokumentaatiorakenteisiin ladattiin vanhan 2.1-rajapintaversion LaTeX-dokumentin sanalliset kuvaukset. Tämän jälkeen muodostettiin työkalun avulla valitun version dokumentaatio ja verrattiin kahta LaTeX-tekstidokumenttia Notepad++-tekstieditorin Compare-liitännäisen [61] avulla. Tällä tavalla voitiin varmistua, että lähdekoodin jäsentäminen sekä dokumentaatiotekstien lukeminen onnistuivat. Uudestaan tuotettu dokumentaatio ei ole täysin identtinen, sillä esimerkiksi XML-rajapintadokumentaation metodien pyyntö- ja vastausrakenteiden kirjaamista yksinkertaistettiin hieman edellisestä versiosta. Dokumentaation selkeyden kannalta ei haluttu kirjata luokan ominaisuuksia kahta iteraatiosyvyyttä syvemmälle.

Vertailtaessa vanhaa 2.1-version dokumentaatiota ja saman rajapintaversion uudelleen tuotettua versiota, löydettiin edellisen rajapintaversion dokumentaatiosta virheitä, kuten puuttuvia luokan ominaisuuksia, väärin nimettyjä ominaisuuksia, väärin merkittyjä ominaisuuden tyyppejä, väärin nimettyjä funktioiden parametreja, puuttuvia hyperlinkkejä sekä tapauksia, joissa oli unohdettu merkitä luokan periytyminen. Virheet huomattiin testattaessa tietomallien vertailijan toimintaa sekä vertailtaessa vanhan rajapintaversion LaTeX-dokumentaatiota alkuperäisen samaa versiota vastaavan LaTeX-dokumentaation kanssa tekstitiedostojen eroavaisuuksia etsivällä työkalulla. Tältä osin voidaan voidaan katsoa, että dokumentointityökalusta on ollut apua myös dokumentaation laadun parantamisessa. Ilman työkalua edellä mainitut virheet olisivat luultavasti jääneet löytämättä ja siirtyneet myös seuraavan rajapintaversion dokumentaatioon.

5.3 Havainnot dokumentointityökalun käytöstä dokumentoinnissa

Työkalu vähensi dokumentointiin kulutettua aikaa, kun tuotettiin eRA-järjestelmän integraatiorajapinnan uuden version dokumentaatio. Dokumentaatiota jo aikaisemmin kirjoittanut ja toteutetun työkalun avulla muodostetun dokumentaatioversion kirjoittanut ohjelmistokehittäjä arvioi dokumentointiin käytetyn ajan puolittuneen verrattuna edellisten dokumentaatioversioiden kirjoittamiseen.

eRA-järjestelmän integraatiorajapinnan tietomallin muodostamiseen kestää työkalulla noin 30 sekuntia. Eniten aikaa tietomallin muodostamisessa menee tarkasteltavien lähdekooditiedostojen syntaksipuiden analysointiin sekä analysoitavien ratkaisujen lähdekoodin kääntämiseen.

6 JATKOKEHITYS

Dokumentointityökaluun voisi olla mahdollista sekä hyödyllistä lisätä muitakin ominaisuuksia tässä työssä toteutettujen toiminnallisuuksien lisäksi. Tässä luvussa pohditaan työkalun jatkokehitysmahdollisuuksia, näistä saatavia hyötyjä sekä muutoksia, joita työkaluun tulisi tehdä, jotta ominaisuudet olisi mahdollista saada toimimaan.

Työkaluun toteutettiin tietomalleja vertaileva metodi. Metodi osaa muodostaa tietorakenteen, johon kirjataan vertailun tulokset. Vertailun lopuksi työkalu havainnollistaa rakenteen avulla erot käyttöliittymässä. Tulevaisuudessa dokumentointityökaluun saatetaan toteuttaa ominaisuus, jolla eroavaisuuksien pohjalta voidaan automaattisesti yhdistää valitut ominaisuudet toiseen tietomalliin.

6.1 Uusien attribuuttiluokkien muodostaminen

Työkaluun toteutettu jäsentäjä osaa lukea ja tallentaa tietomalliin aliluvussa 4.2 esiteltyjen attribuuttiluokkien sisältämät tiedot. Attribuuttiluokkien rakenne ja toiminta on suunniteltu eRA-järjestelmälle sopiviksi. Dokumentoidessa muiden ohjelmistojen rajapintoja voivat attribuuttiluokat olla riittämättömiä tai niiden rakentajissa tulee syöttää parametrina tieto, joka ei ole ohjelmiston rajapinnan kannalta oleellista. Tällöin tulee toteuttaa uusia attribuuttiluokkia tai kommenttirakenteita, joilla saadaan merkittyä tarvittavia lisätietoja lähdekoodiin. Vastaavasti lähdekoodin jäsentäjään tulee tällöin toteuttaa uudet toiminnallisuudet näiden tietojen lukemiseen.

Työkalun käyttöliittymä on riippuvainen toteutetuista attribuuttiluokista, sillä rakenteiden näyttäminen sekä dokumentaation kirjoittaminen on toteutettu käyttöliittymään sillä periaatteella, että tietomallista löytyvät attribuuttiluokissa kerrotut tiedot. Jos attribuuttiluokkia ei käytetä lähdekoodissa lainkaan tai lähdekoodiin otetaan käyttöön uusia attribuuttiluokkia tulee käyttöliittymän toimintaa muokata vastaavasti.

Uusien attribuuttiluokkien muodostamiseen ei ole yksinkertaista tapaa dokumentointityökalun käyttäjälle. Työkalun käytön kannalta käyttäjälle miellyttävin tapa muodostaa uusia attribuutteja olisi vain kertoa työkalun käyttöliittymän kautta uuden attribuuttiluokan nimi ja kertoa mitä tietoja attribuutin rakentajan parametreilla merkitään ja mihin osioihin tietomallia parametrien arvot tallennetaan.

6.2 Erilaisten ohjelmointikielten jäsentäminen toteutettuun tietomalliin

Rajapintaa kuvaavaa tietomallia lähdettiin suunnittelemaan ja toteuttamaan niin että siihen tallennetaan C#-ohjelmointikielellä toteutetun rajapinnan tiedot. Toteutuksesta pyrittiin kuitenkin saamaan mahdollisimman hyvin laajennettavissa oleva niin, että vain jäsentäjämoduuleja lisäämällä olisi mahdollista muodostaa tietomalli.

Vaikka tietomallia on kirjoitettu C#-kielellä, voidaan tietomalliin tallentaa myös muulla ohjelmointikielellä kirjoitettua rajapintaa. Koska dokumentoitu rajapinta on kirjoitettu C#-ohjelmointikieltä käyttäen, on tietomalli toimiva luultavasti myös muille staattisesti tyypitetyille olio-pohjaisille kielille. Esimerkiksi C++, Java tai TypeScript ovat staattisesti tyypitettyjä olio-pohjaisia kieliä, jotka sisältävät hyvin samankaltaisia rakenteita kuin C#-kieli. Tietotyyppien eroavaisuudet eivät haittaisi tietomallia tallentaessa, jos tietotyypin nimi tai arvot voidaan muuntaa luvuiksi tai merkkijonoiksi, sillä tieto, jota tallennetaan tietomallin luokkiin tallennetaan joko merkkijonoina tai lukuarvoina. Jos haluttaisiin tallentaa muita staattisesti tyypitettyjä kieliä, niin luultavasti tietomallia tulisi laajentaa, sillä tällä hetkellä se ei tue edes kaikkia rakenteita, joita on mahdollista toteuttaa C#-kielellä, koska tietomalli on suunniteltu tallentamaan vain rajapinnan kannalta olellisia tietoja.

Toteutetussa C#-jäsentäjässä attribuuttiluokat ovat tärkeässä osassa rakenteiden tunnistamisessa ja versioiden ilmoittamisessa. Jos ohjelmointikielestä ei löydy samankaltaista ominaisuutta voi tiedon kirjoittaa lähdekoodiin esimerkiksi kommentin sisään.

Dynaamisesti tyypitettyjen kielten muodostaminen tietomalliksi onnistuu kyllä, mutta tietomalliin ei voida lisätä automaattisesti kaikkea tietoa tyypeistä, koska dynaamisissa kielissä ei funktioiden parametreille, paluuarvoille tai luokkien attribuuteille ole tyyppimäärittelyjä. Esimerkiksi Python-kielellä toteutetun HTTP POST -metodin pyyntö- tai vastausrakenteiden skeemaa ei voi suoraan päätellä metodin määrittelystä. Jotta pyyntöjen ja vastausten skeema voitaisiin esittää dokumentaatiossa, tulee tämä tieto merkitä lähdekoodiin jollain määritellyllä kommenttirakenteella. REST-rajapinnan vastausrakenteen skeeman dokumentoinnissa tulee määritellä rakenteisen merkintäkielen avaimia vastaavan arvon tyyppi, esimerkiksi lista ja tämän jälkeen listan yhden alkion rakenne. Listan alkio voi olla jokin ennelta määritetty luokka, jolloin lähdekoodissa olevaan skeeman merkintädokumentaatioon voisi kirjata, että alkiot ovat sarjallistettuja tyyppejä sekä kirjata tyypin nimi. Jos alkiot eivät ole ennaltamääriteltyjä luokkia, voitaisiin kertoa, mitä primitiivistä tietotyyppiä kukin alkion elementti edustaa, ja mikä on elementin avain.

```
'''
!InterfaceDocumentation!
Method Request: XML
Schema:
Type: List
```

```
Items: predefinedType namespace.namespace.class

Method Response: XML
Schema:
Type: Item
Properties:
string firstName
int age
string lastName
!InterfaceDocumentation!
'''
```

Ohjelma 6.1. *Dynaamisesti tyypitetyn ohjelmoitikielen lähdekoodiin lisättävä dokumentointia helpottava teksti.*

Ohjelmassa 6.1 esimerkki kuinka esimerkiksi Python-kielelle voidaan määritellä lähdekoodiin lisättävä kommenttiosion sisään kirjoitettava kuvaus. Esimerkissä kerrotaan yksittäisen metodin pyyntö ja vastausrakenteen muoto kuvitteellisella kuvausrakenteella.

6.3 Ohjelmiston rakenteen visualisointi

Työkalu on tarkoitettu ensisijaisesti kuvaamaan rajapinnan rakennetta, mutta työkalulla olisi mahdollista myös dokumentoida kokonaisen ohjelmiston rakennetta. Kokonaisen ohjelmiston dokumentoinnissa hyödyllistä voisi olla esimerkiksi UML-luokkakaavioiden automaattinen piirtäminen lähdekoodin perusteella. Tietomalli sisältää lähes kaiken tarvittavan tiedon, jonka avulla UML-luokkakaavioita on mahdollista piirtää. Luokkien välisten assosiaatioviivojen kuvaustekstien ja määrien esittämiseen ei ole tietoa tietomallissa. Myös koostumisen ja muodostumisen assosiaatioita ei voida luotettavasti päätellä tietorakenteesta. Sen sijaan periytyminen ja normaalit tunnuksettomat assosiaatioviivat on mahdollista piirtää tietomallin tiedoilla. Jotta esimerkiksi koostumisen ja muodostumisen assosiaatioviivat voitaisiin piirtää, tulisi lisätä uusia attribuuttiluokkia lähdekoodiin, tai lisätä dokumentointityökalun käyttöliittymään tapa valita luokkien välisiä assosiaatioita, sekä assosiaatioissa ilmoitettuja määriä.

Piirtämiseen kannattaisi käyttää jotain valmista piirtokirjastoa, esimerkiksi Javalla toteutetun PlantUML-ohjelman [65] avulla voidaan piirtää UML-kaaviota. PlantUML-työkalulla voidaan piirtää luokkakaavioita tekstimuotoisesta PlantUML-muotoisella syntaksilla kirjoitetusta lähdetiedostosta. Dokumentointityökaluun tulisi kirjoittaa oma piirtomoduuli, joka kirjoittaisi tietomallin luokkien pohjalta PlantUML-syntaksia ja tallentaisi tämän tiedostoon. Sitten dokumentointityökalu voisi kutsua PlantUML-työkalua komentoriviltä piirtämään kaaviokuvan. Dokumentointityökalun käyttöliittymään tulisi lisätä näkymä, jossa voitaisi valita mitä osia ja luokkia piirrettävään kaavioon valitaan.

Toinen tapa visualisoida rajapintaa voisi olla rajapintakutsujen tilakaavio. Tilakaaviosta

kävisi ilmi, mikä on sallittu järjestys rajapintafunktioiden kutsumiselle. Tilakaavion piirtäminen voisi olla mahdollista, koska rajapintakutsuille on mahdollista määrittää, mitä esiehtoja järjestelmän tilan tulee toteuttaa, ennen kuin rajapintakutsua saadaan kutsua. Työn kirjoitushetkellä työkaluun toteutettuja esiehtoja metodeille ovat “vaatii avatun istunnon”, “vaatii avatun potilastapahtuman” ja “vaatii avatun palvelutapahtuman”. Vielä tietomalliin ei kuitenkaan ole mahdollista tallentaa tietoa siitä, miten kutsujen kutsuminen vaikuttaa järjestelmän tilaan. Jotta tilakoneeseen voitaisiin piirtää siirtymiä, tulisi esimerkiksi “avaa potilas” -kutsulla olla tieto, että kutsun kutsumisen jälkeen järjestelmä on tilassa, jossa potilastapahtuma on avattu.

6.4 Ohjelmakoodin tuottaminen tietomallin avulla

Mahdollinen kiinnostava ominaisuus voisi olla ohjelmakoodin tuottaminen tietomallista. Tietomalli ei kuitenkaan sisällä tarpeeksi tietoa, jotta tietomallin avulla voitaisiin muodostaa toimiva rajapinta. Tietomallin sisältämä tieto mahdollistaa kuitenkin tynkätoteutuksien muodostamisen. C#-kieliselle rajapinnalle voitaisiin hyödyntää Roslyniä, sillä Roslynin avulla voidaan muodostaa syntaksipuita, joista voidaan tuottaa ohjelmakoodia.

Tietomallista ei voida tuottaa samaa rajapinnan toteutusta kuin mistä se on luettu, sillä metodien toimintaa ei ole tallennettu. Usein rajapinta voi olla kiinteänä osana muuta projektia, jota ei ole ladattu mukaan tietomalliin esimerkiksi sijaiten osittain samoissa tiedostoissa muun toteutuksen kanssa. Tällöin, jos toteutuksen tuottaisi tietomallista, eivät lähdetiedostot olisi enää samanlaisia. Jotta esimerkiksi käyttöliittymän kautta tietomalliin lisätyn kokonaan uuden luokan voisi lisätä lähdekoodiin, olisi se luultavasti helpoin muodostaa kokonaan uuteen tiedostoon. Microsoft.CodeAnalysis.Charp.SyntaxFactory-luokka mahdollistaa uusien syntaksipuiden muodostamisen [49].

Syntaktisesti oikein muodostetusta syntaksipuusta voidaan tuottaa ohjelmakoodia. Vaikka puusta saadaan tuotettua syntaktisesti oikea, se ei vielä takaa sitä, että tuotettu ohjelmakoodi saataisiin käännettyä.

Ohjelmassa 6.2 luodaan SyntaxFactory-luokan metodein uusi CompilationUnit-olio, joka voidaan muuttaa lähdekooditiedostoksi. Koodissa muodostetaan uusi nimiavaruus, sen sisälle luokka ja luokalle yksi kenttä. Kun CompilationUnit-olio on luotu, voidaan kutsua sen ToString-metodia, joka palauttaa syntaksipuuta vastaavan ohjelmakoodin merkkijonona. Merkkijonoon code tallennettu sisältö nähdään tulostettuna ohjelmassa 6.3. Uusien rakenteiden tuottaminen onnistuu, kun ohjelmoija tietää, mitä viittauksia tarvitaan ja mihin nimiavaruuteen luokka sijoitetaan.

Myös jo olemassa olevaa koodia voidaan muuttaa lataamalla tiedoston syntaksipuu muistiin ja muuttamalla sieltä haluttuja rakenteita uusiksi rakenteiksi. SyntaxTree-oliot ovat muuttumattomia (engl. immutable) oliota, joten jos syntaksipuuta tahdotaan muokata, ReplaceNode-metodi kopioi koko puun uuteen muuttujaan. Jos muokkauksia tehdään hyvin paljon, voi prosessi olla hidas etenkin suurten syntaksipuiden kohdalla.

```
1  // muodostetaan uusi syntaksipuu rakentamalla yksi CompilationUnit-olio
2  SyntaxTree tree = SyntaxFactory.CompilationUnit().WithUsings(
3  // Kirjataan using System
4  SyntaxFactory.SingletonList<UsingDirectiveSyntax>(
5  SyntaxFactory.UsingDirective(
6  SyntaxFactory.IdentifierName("System"))))
7  // muodostetaan nimiavaruus nimeltä ABC
8  .WithMembers(SyntaxFactory.SingletonList<MemberDeclarationSyntax>(
9  SyntaxFactory.NamespaceDeclaration(SyntaxFactory.
10 IdentifierName("ABC"))
11 // nimiavaruuden sisään luodaan X-luokan määrittely
12 .WithMembers(SyntaxFactory.SingletonList<MemberDeclarationSyntax>(
13 SyntaxFactory.ClassDeclaration("X")
14 //luokalle muodostetaan yksi int tyyppinen kenttä
15 //nimeltä kokonaislukuKentta ja laitetaan sen määreeksi public
16 .WithMembers(SyntaxFactory.SingletonList<MemberDeclarationSyntax>(
17 SyntaxFactory.FieldDeclaration(
18 SyntaxFactory.VariableDeclaration(SyntaxFactory.PredefinedType(
19 SyntaxFactory.Token(SyntaxKind.IntKeyword))).WithVariables(
20 SyntaxFactory.SingletonSeparatedList<VariableDeclaratorSyntax>(
21 SyntaxFactory.VariableDeclarator(SyntaxFactory.
22 Identifier("Y")))))
23 .WithModifiers(SyntaxFactory.TokenList(
24 SyntaxFactory.Token(SyntaxKind.PublicKeyword))))))))
25 .NormalizeWhitespace().SyntaxTree;
26 //muodostetusta syntaksipuusta voidaan saada ohjelmakoodi
27 //ja tallentaa se muuttujaan code
28 string code = tree.ToString();
```

Ohjelma 6.2. *SyntaxFactory-luokan metodeilla rakennettu syntaksipuu, joka muunnetaan C# ohjelmakoodiksi.*

```
1 using System;
2
3 namespace ABC
4 {
5     class X
6     {
7         public int Y;
8     }
9 }
```

Ohjelma 6.3. *Ohjelmassa 6.2 merkkijonoon code tallentuva arvo, joka kuvaa yksinkertaisen luokan määrittelyä.*

```
1  // haetaan syntaksipuusta kaikki luokkien määrittelyt
2  IEnumerable<ClassDeclarationSyntax> nodes = tree.GetRoot()
3  .DescendantNodes().OfType<ClassDeclarationSyntax>().ToList();
4  // käänetään syntaksipuu ja muodostetaan sen avulla semanttinen malli
5  CSharpCompilation compilation = CSharpCompilation.Create("ABC")
6  .AddSyntaxTrees(tree.GetRoot().SyntaxTree);
7  SemanticModel model = compilation.GetSemanticModel(tree);
8  // haetaan semanttisesta mallista luokan syntaksipuulla sitä vastaava
   symboli
9  ISymbol symbol =  model.GetDeclaredSymbol(nodes.First());
10 // Tarkistetaan onko symboli se jota tahdotaan muokata
11 if (symbol.ToString() == "ABC.X" ) {
12 MemberDeclarationSyntax oldMemberDeclaration = nodes.First()
13 .DescendantNodes().OfType<MemberDeclarationSyntax>().First();
14 // muodostetaan uusi luokka ja tallennetaan se muuttujaan newMember
15 SyntaxList<MemberDeclarationSyntax> newMember = SyntaxFactory
16 .SingletonList<MemberDeclarationSyntax>(SyntaxFactory.
17 FieldDeclaration(SyntaxFactory.VariableDeclaration(
18 SyntaxFactory.IdentifierName("B")).
19 WithVariables(SyntaxFactory.SingletonSeparatedList<
20 VariableDeclaratorSyntax>(SyntaxFactory.VariableDeclarator(
21 SyntaxFactory.Identifier("newField")))))
22 .WithModifiers(SyntaxFactory.TokenList(
23 SyntaxFactory.Token(SyntaxKind.PrivateKeyword))));
24 // korvataan vanha luokka uuden luokan syntaksipuulla
25 SyntaxNode newTree = tree.GetRoot()
26 .ReplaceNode(oldMemberDeclaration, newMember).NormalizeWhitespace();
27 string newCode = newTree.ToString();}
```

Ohjelma 6.4. *Syntaksipuun muokkaaminen. Ohjelmassa käytetty tree-muuttuja on ohjelmassa 6.2 muodostettu syntaksipuu, joka on SyntaxTree-tyyppiä.*

Jos työkalulla halutaan muokata lähdekoodissa jo sijaitsevia rakenteita, tulee paikantaa lähdekoodikirjastosta tiedosto, jossa muokattava rakenne sijaitsee. Tämän jälkeen muodostaa CompilationUnit-rakenne työkalun muistiin ja paikantaa puusta muokattavan rakenne.

Semanttiselta mallilta voidaan sitten kysyä luokan symbolista nimeä, jolla oikean luokan voi tunnistaa puusta. Sitten ClassDeclarationSyntax-oliolta voidaan hakea jälkeläissolmuja samalla tavoin. Kun on löydetty muokattava solmu, voidaan käyttää ReplaceNode-metodia [47], jolle annetaan löydetty vanha solmu viitteenä sekä lista uusista solmuista, joilla korvataan vanha solmu ja sen jälkeläisinä olevat solmut. Ohjelmassa 6.4 on syntaksipuun luokan muokkaamiseen tarkoitettu koodi. Ohjelmassa on valittu vain ensimmäinen luokka sekä ensimmäinen kenttä syntaksipuusta jos näitä ei löytyisi, ohjelma ei toimisi. Merkkijonon newCode sisältämä teksti on kuvattu ohjelmassa 6.5.

Työkalun käyttöliittymään on työlästä toteuttaa muokkausmahdollisuudet lähdekoodille. Jos käyttäjä tekee virheen tietomallia muokatessaan, ei semanttista mallia voida välttämättä muodostaa, sillä virheestä riippuen koodi ei välttämttä enää käänny. Ohjelmassa

```
1 using System;
2
3 namespace ABC
4 {
5     class X
6     {
7         private B newField;
8     }
9 }
```

Ohjelma 6.5. *Muokatun syntaksipuun avulla muodostettu ohjelmakoodi, joka on tallennettu ohjelmassa 6.4 merkkijonoon newCode.*

6.5 nähdään että muodostetun ohjelman syntaksi on oikea, mutta ohjelmaa ei saada käännettyä, sillä tyyppiä B ei ole määritelty.

Ennen kuin käyttöliittymään on toteutettu toiminnallisuudet lähdekoodin muokkaamiseen, on vaikeaa arvoida nopeuttaisiko käyttöliittymän kautta tapahtuva muokkaaminen lähdekoodin tuottamista vai olisiko ohjelmistokehittäjän nopeampaa kirjoittaa uudet ominaisuudet suoraan lähdekoodiin ilman työkalun käyttämistä.

Jos tietomallin avulla voidaan tuottaa lähdekoodia, mahdollistaisi tämä ominaisuus rajapinnan toteuttamisen dokumentaatio edellä, jolloin dokumentaatiota käytettäisiin toteutuksen spesifikaationa, mutta dokumentaatiosta voitaisiin tuottaa myös automaattisesti tynkätoteutus rajapinnalle.

6.5 Rajapintakomentojen testirunkojen automaattinen tuottaminen

XML-rajapinnoille olisi mahdollista muodostaa automaattisesti yksinkertaisia yhden metodin testaamiseen tarkoitettuja testirunkoja. Tässä puhutaan testirungoista, sillä kokonaisen toimivan testin muodostaminen täysin automaattisesti ei ole tietomallin tietojen pohjalta mahdollista. Testirungot voisivat olla esimerkiksi Visual Studion testiprojektiin muodostettuja testimetodeja. Jokaiselle rajapinnan metodille muodostettaisiin oma testifunktionsa. Testifunktion toiminta voisi yksinkertaisesti olla lähettää HTTP-pyyntö URL-osoitteeseen oikeanmuotoisella XML-viestillä ja tarkistaa palvelimen vastaus. Lähetettävän ja vastaanotettavan XML-elementin muoto saadaan tietoon rajapintametodille annetun DocumentationXmlInterfaceMethod-attribuutin RequiredParameters-, RequiredParameterType- sekä ResponseParameterType-ominaisuuksista. Tarvittavien parametrityyppien luokat luetaan tietomallista ja tutkitaan XMLIdentificationString- sekä DocumentationXmlRequestParameter- sekä DocumentationXmlResponseParameter-attribuuttien tiedoista, mitkä XML-elementit tulee sisällyttää onnistuneeseen pyyntö- sekä vastausrakenteeseen.

Ongelma, johon tietomallista ei löydy ratkaisua, on tarvittavien XML-kenttien täydentämi-

nen oikeanmuotoisilla arvoilla. Toinen ongelma on toteuttaa testirungot niin, että metodeja kutsutaan silloin, kun testattavan rajapinnan takana oleva ohjelmisto on oikeassa tilassa. Esimerkiksi joidenkin metodien vaatimuksena on avattu istunto tai avattu palvelutapahtuma. Tällöin tulisi tunnistaa, mitä metodeja täytyy suorittaa, ennen kuin järjestelmän tila on sallittu metodin kutsumiselle. Kolmas ongelma testitapausten muodostamisessa on virheellisten pyyntöjen lähettämiseen liittyvä ongelma. Vaikka tietomallista voidaan tietää mitkä virhekoodit ovat mahdollisia metodista, ei voida tietää, millä syötteellä virheen tulisi tapahtua. Osa virhekoodeista vastaavasti voi johtua siitä, että pyyntö kutsutaan silloin, kun järjestelmä ei ole pyynnön kutsumiseen sallitussa tilassa.

Testirunkojen generointi voidaan toteuttaa niin, että työkalulla voidaan luoda Roslynin Workspace API:n avulla uusi testiprojekti. Roslynin avulla voidaan muodostaa uusia syntaksipuita, jotka voidaan muuttaa C#-kieliseksi tiedostoksi, mukaillen edellisessä luvussa esitettyä tapaa.

Testirunkojen muodostamiseen tarkoitettu toiminnallisuus ei kuitenkaan hyödytä rajapinnan dokumentoinnissa, joten sitä ei kannata sisällyttää samaan työkaluun. Kuitenkin tietomallia sekä jäsentäjää voisi mahdollisesti hyödyntää tämänkaltaisen testausrunkoja toteuttavan ohjelman toteuttamisen

6.6 Tietomallin sarjallistaminen OpenApi Documentation -kuvauskieleksi

Työkalulle on mahdollista toteuttaa moduuli, jonka avulla voitaisiin sarjallistaa rajapinnan tietomalli OpenApi Documentation -kuvauskieleksi. OpenApi Documentation -kuvauskielestä voidaan muodostaa erilaisia dokumentaatiota esimerkiksi Swagger2Markup-työkalun avulla [71]. OpenApi Documentation -mallista voidaan myös tuottaa tynkätoteutuksia useille eri ohjelmointikielille käyttämällä esimerkiksi Swagger Codegen työkalua [69]. Dokumentointavan eRA-järjestelmän tapauksessa OpenApi Documentation -malli voitaisiin tuottaa REST XML -rajapinnasta, mutta käärerajapintojen toteutusta ei voida muuntaa OpenApi Documentation -malliksi, sillä ne eivät ole REST-rajapintoja.

Tietomallin sarjallistaminen OpenApi Documentation -kuvauskieleksi voidaan mahdollistaa toteuttamalla dokumentointityökaluun uusi moduuli, jonka tehtävänä on muuntaa InterfaceModel-luokan sisällä sijaitsevat rakenteet kuvauskieleksi. OpenApi Documentation -kuvauskielen rakenne on YAML- tai JSON-syntaksia. Yksinkertaisetettuna OpenApi kuvauskielessä metodit tunnistetaan URL-osoitteiden avulla. Näiden URL-osoitteita kuvaavien elementtien sisälle sijoitetaan eri HTTP-metodeja kuvaavia elementtejä. HTTP-elementtien sisälle voidaan kirjata parametri- ja vastaus-elementtejä eri HTTP-tilakoodeille. Tietomallista löytyvät kaikki vaaditut tiedot ja tietomallista voidaan eritellä tarvittavat tiedot melko helposti, sillä ne on ryhmitelty tietomalliin lähes vastaaviksi rakenteiksi.

6.7 Rajapintadokumentaation automaattinen toteuttaminen jatkuvassa integraatiossa

Jos ohjelmistoa kehitetään jatkuvan integraation menetelmillä (engl. Continuous Integration, CI), voitaisiin dokumentointityökalun toiminta lisätä jatkuvan integraation automaatiopalvelimelle. Automaatiopalvelin tarkkailee lähdekoodiin tehtyjä muutoksia versionhallintajärjestelmän avulla ja palvelimella voitaisiin automaattisesti muodostaa uusin rajapintadokumentaatio järjestelmän uusimman lähdekoodin ja vanhojen dokumentaatioden pohjalta. Tällöin työkaluun kannattaisi luultavasti toteuttaa komentorivikäyttöliittymä, jonka avulla automatisointi olisi helppo toteuttaa.

Tietomalli muodostetaan uusimmasta lähdekoodiversiosta aina, kun versionhallinnassa sijaitsevaan lähdekoodiin tulee muutos. Muodostettu tietomalli tallennetaan tiedostoon. Edellisen dokumentaation dokumentaatiotekstit ladataan uusimpaan tietomalliin LaTeX-dokumentista, joka tuotetaan aina, kun rajapintaan on tullut muutoksia. Muutokset voidaan tarkistaa ajamalla työkalun tietomalleja vertaileva funktio. Tämän jälkeen työkalulla tulisi tarkistaa, onko kaikilla rakenteilla kuvaukset. Kuvaukset voivat puuttua silloin, kun rajapintaan on lisätty uusi rakenne tai jo olemassa olevan rakenteen nimeä on muutettu. Tapauksessa, jossa kuvauksia puuttuu, voidaan integraatiopalvelimen testituloksessa kertoa, että kuvauksia puuttuu ja uutta versiota ei kannata julkaista, ennen kuin rajapinnan dokumentaation puuttuvat kuvaukset on kirjoitettu.

Edellä kuvatussa prosessissa on se ongelma, että dokumentaatiotekstien kirjoittaminen ei ole automaattista, vaan uusien rakenteiden ilmestyessä tietomalliin tarvitaan aina ihminen, joka kirjoittaa kuvaukset. Tässä mielessä työkalua ei käytettäisi näin dokumentoimiseen, vaan sen varmistamiseen, että dokumentaatio muistettaisiin joskus kirjoittaa. Kuitenkin tällaisen testauksen heikkoutena on se, että rajapinnan kehittäjä saattaa unohtaa merkitä attribuuttiluokat uusiin toteutettuihin rajapinnan rakenteisiin, jolloin niitä ei lisätä automaattisesti työkalulla tietomalliin, eikä prosessilla huomata, että rajapinta on muuttunut.

7 YHTEENVETO

Ohjelmointirajapintojen käyttämisen kannalta on tärkeää, että rajapinnoilla on kattava ja mahdollisimman virheetön dokumentaatio. Tässä työssä pyrittiin nopeuttamaan rajapintadokumentaation muodostamista, pienentämään ohjelmoijan työtaakkaa dokumentaation kirjoittamisessa sekä parantamaan tuotetun dokumentaation laatua. Dokumentaation muodostamista automatisoitiin lukemalla dokumentoitavan rajapinnan lähdekoodista tarvittavia tietoja dokumentaatiopohjan toteuttamiseen. Lähdekoodista muodostetiin tietomalli, jolle on mahdollista kirjoittaa kuvaustekstejä käyttöliittymässä. Koska dokumentaatio muodostetaan lähdekoodin pohjalta, voidaan varmistua siitä, että dokumentaatiossa kuvattujen rakenteiden mallit ovat ajantasaisia ja vastaavat toteutetun rajapinnan rakenteita.

Työssä suunniteltu ja toteutettu rajapintadokumentointityökalu toteutettiin .NET-ympäristön C#-kielellä. Toteutettu työkalu koostuu neljästä eri moduulista: käyttöliittymä-, dokumentaatio-, jäsentäjä- sekä tietomallimoduulista. Työkalun arkkitehtuurista toteutettiin modulaarinen, jotta työkalu olisi laajennettavissa tukemaan useita eri dokumentaatiotyyppejä sekä ohjelmointikieliä. Työkalulle toteutettiin lähdekoodin jäsennysmoduuli C#-kielelle sekä dokumentaatiomoduuli määritellyn muotoiselle LaTeX-dokumentaatiolle. Kyseiset tekniikat moduuleille valittiin, sillä työssä dokumentoitu eRA-järjestelmän rajapinta on toteutettu C#-kielellä. Lisäksi LaTeX-dokumentaation malli oli olemassa ja sitä oli käytetty dokumentoitavan eRA-järjestelmän rajapintadokumentaation edellisissä versioissa.

Dokumentoitavan eRA-järjestelmän rajapinnan rakenteet tunnistettiin lähdekoodista niille asetettujen attribuuttiluokkien avulla. C#-kielen kääntäjäympäristöön perustuvan Roslyn-jäsentäjän avulla toteutettiin jäsentäjämoduuli, joka rakentaa tietomallimoduulin mukaisen rajapinnan tietomallin lähdekoodin pohjalta. Tietomallin kokoamisessa käytettiin Roslynillä muodostettua syntaksipuuta sekä lähdekoodin käännösestä ja syntaksipuusta tuotettua semanttista mallia.

Työlle ennalta asetetut vaatimukset täytettiin. Työssä ohjelmoidun dokumentointityökalun avulla voidaan dokumentoida rajapintaa usealla luonnollisella kielellä. Rajapinnasta tunnistetaan rakenteita versio- sekä osiokohtaisesti. C#-kielellä kirjoitetusta rajapinnasta voidaan lukea tietomalli, ja metodien dokumentoinnissa voidaan dokumentoida sekä REST-tyyppisiä HTTP-metodeja että .NET-luokkakirjaston metodeja. Työkalun rakenteesta muodostettiin laajennettavissa oleva, niin että samaa tietomallia sekä käyttöliittymää voidaan käyttää myös muidenkin ohjelmointikielten kuin C#-kielen dokumentoitiin. Työkalun avulla voidaan myös havaita eroavaisuuksia tietomallien välillä.

Työkalun käyttö mahdollistaa dokumentaation muodostamisen nopeammin, kuin aikaisemmalla menetelmällä, jossa dokumentoijan tuli muistaa rajapintaan kuuluvat rakenteet ja muodostaa niille dokumentaatio manuaalisesti. Työkalua käyttämällä voidaan huomata käyttöliittymässä, rakenteelta puuttuvat kuvaukset. Lisäksi kaikki rajapintaan ohjelmointivaiheessa merkityt rakenteet ladataan automaattisesti dokumentointikäyttöliittymään. Näin voidaan olla varmempia, että dokumentaatio vastaa rajapinnan toteutusta.

Työkalun heikkoudeksi voi ajatella sen, että työkalun toteuttamisessa kului paljon aikaa verrattuna siihen, kuinka kauan dokumentaation kirjoittamisessa on yleensä kestänyt. Työkalun käyttöliittymä on riippuvainen toteutetuista attribuuttiluokista. Työkalu tukee tällä hetkellä vain C#-ohjelmointikieltä, joten se ei sellaisenaan suoraan sovellu kaikkien rajapintojen dokumentointiin. Käyttöliittymä tukee tällä hetkellä vain kuvausten muokkaamista, ja itse rakenteiden muokkaaminen ei ole mahdollista. Vertailun tulosten pohjalta ei voida automaattisesti muokata tietomallien sisältöjä. Näitä ominaisuuksia ei vielä toteutettu työn kirjoittamisen aikana, vaan ne jäävät mahdollisesti tulevaisuudessa kehitettäviksi ominaisuuksiksi. Työkalun toteutuksessa priorisoitiin kuvausten tallentamiseen sekä dokumentaation tuottamiseen tarkoitettuja ominaisuuksia, jotta ne olisi mahdollista saada valmiiksi seuraavan rajapintaversion dokumentointia varten.

Toteutettua työkalua voidaan pitää onnistuneena, sillä työlle asetetut vaatimukset täytettiin. Työkalu oli kuitenkin työmäärältään odotettua laajempi, joten työkalulle asetetussa tavoitteellisessa aikataulussa ei pysytty. Tavoitteena oli saada työkalun dokumentointia mahdollistava osuus toimimaan syyskuun lopussa, mutta dokumentointiosuudet saatiin tarpeeksi toimiviksi vasta muutamaa kuukautta myöhemmin joulukuun alussa.

LÄHTEET

[1] Adobe Systems. *Document management - Portable document format - Part1: PDF 1.7*. URL: https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf (viitattu 26. 02. 2019).

[2] A. V. Aho, R. Sethi ja J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986. ISBN: 0-201-10194-7.

[3] C. Anderson. The Model-View-ViewModel (MVVM) Design Pattern. Teoksessa: *Pro Business Applications With Silverlight*. Berkeley, CA: Apress, 2012, 461–499. ISBN: 1430235004. DOI: 10.1007/978-1-4302-3501-9_13.

[4] T. Archer. *Inside C#*. microsoft Press, 2001. ISBN: 0735612889. URL: http://blog.shuo1.com/zms/books/c/Inside%20C%20Sharp.pdf.

[5] O. Ben-Kiki ja C. Evans. YAML Ain't Markup Language (YAML™) Version 1.2e. 2009 (). URL: http://yaml.org/ (viitattu 30. 10. 2018).

[6] T. Berners-Lee, L. Masinter ja M. McCahill. *Uniform Resource Locators (URL)*. 1994. URL: https://tools.ietf.org/html/rfc1738 (viitattu 26. 02. 2019).

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau ja W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 2008. URL: https://www.w3.org/TR/2008/REC-xml-20081126/ (viitattu 22. 08. 2018).

[8] K. Brown, S. Cai, M. Jones, T. Petersen, G. Hogenson, G. Warren ja M. Sebolt. *Solutions and projects in Visual Studio*. URL: https://docs.microsoft.com/en-us/visualstudio/ide/solutions-and-projects-in-visual-studio?view=vs-2017 (viitattu 14. 02. 2019).

[9] A. Buckley. A Model of Dynamic Binding in .NET. Teoksessa: vol. 3798. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, 149–163. ISBN: 0302-9743. DOI: 10.1007/11590712_12.

[10] A. Butterfield ja G. E. Ngondi. application programming interface. *A Dictionary of Computer Science* (2016). (Viitattu 06. 07. 2018).

[11] N. P. Chapman. *LR Parsing: Theory And Practice*. Campbridge University Press 1987, 1987.

[12] D. Clark. *Beginning C# Object-Oriented Programming*. 1. painos. Berkeley, CA: Apress, 2011. ISBN: 9781430235309. DOI: 10.1007/978-1-4302-3531-6.

[13] R. David ja T. A. Proebsting. A research C# compiler. *Software: Practice and Experience* 34.13 (2004), 1211–1224. DOI: 10.1002/spe.610.

[14] Ecma international. The JSON Data Interchange Syntax. *ECMA-404 The JSON Data Interchange Standard* 2017 (). URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (viitattu 30. 10. 2018).

[15] S. Endrikat, R. Hanenberg, R. Robbes ja A. Stefik. How do API documentation and static typing affect API usability? Teoksessa: ACM, 2014, 632–642. ISBN: 0270-5257. DOI: 10.1145/2568225.2568299.

[16] R. Fielding, U. C. Irvine, J. Gettys, J. Mogul, H. Frystyk, P. Leach, T. Berners-Lee, L. Masinter, W. MIT, C. W3C, Xerox, Microsoft ja Compaq. *Hypertext Transfer Protocol – HTTP 1.1*. 1999. URL: https://tools.ietf.org/html/rfc2616 (viitattu 03. 12. 2018).

[17] R. Fielding ja R. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)* 2.2 (2002), 115–150. DOI: 10.1145/514183.514185.

[18] G. Finzer, S. Hailey ja KellermanSoftware. *ComparisonResult.cs*. URL: https://github.com/GregFinzer/Compare-Net-Objects/blob/master/Compare-NET-Objects/ComparisonResult.cs (viitattu 13. 12. 2018).

[19] G. Finzer ja KellermanSoftware. *BaseTypeComparer.cs*. URL: https://github.com/GregFinzer/Compare-Net-Objects/blob/master/Compare-NET-Objects/TypeComparers/BaseTypeComparer.cs (viitattu 13. 12. 2018).

[20] G. Finzer ja KellermanSoftware. *Compare-Net-Objects*. URL: https://github.com/GregFinzer/Compare-Net-Objects (viitattu 03. 12. 2018).

[21] G. Finzer ja KellermanSoftware. *CompareLogic.cs*. URL: https://github.com/GregFinzer/Compare-Net-Objects/blob/master/Compare-NET-Objects/CompareLogic.cs (viitattu 13. 12. 2018).

[22] GalaSoft. *MVVMLight*. URL: http://www.mvvmlight.net/ (viitattu 29. 08. 2018).

[23] J. Gerard, M. Hoffman, B. Wagner, M. Wenzel ja Microsoft. *C# Keywords*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ (viitattu 11. 12. 2018).

[24] J. Gerard, M. Hoffman, B. Wagner, M. Wenzel ja Microsoft. *C# Operators*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/ (viitattu 11. 12. 2018).

[25] ISO. *ISO/IEC 14977 . 1996 (E) Extended BNF*. pub-ISO, 1996, 19. URL: https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf.

[26] M. Jones, P. Kulikov, L. Latham, M. Wenzel ja Microsoft. *Windows Presentation Foundation*. 2018. URL: https://docs.microsoft.com/en-us/dotnet/framework/wpf/ (viitattu 03. 12. 2018).

[27] M. Jones, L. Latham, M. Wenzel ja Microsoft. *Application Domains*. 2017. URL: https://docs.microsoft.com/en-us/dotnet/framework/app-domains/application-domains (viitattu 22. 08. 2018).

[28] Kansallinen Terveysarkisto. *Mitä Kanta-palvelut ovat*. URL: https://www.kanta.fi/fi/mita-kanta-palvelut-ovat (viitattu 20. 02. 2019).

[29] L. Latham, M. Hoffman, M. Wentzel ja Microsoft. *COM Callable Wrapper*. 2017. URL: https://docs.microsoft.com/en-us/dotnet/framework/interop/com-callable-wrapper (viitattu 22. 08. 2018).

[30] G. F. Levy. Numeric ActiveX components. *Software: Practice and Experience* 31.2 (2001), 147–189. DOI: AID-SPE360>3.0.CO;2-V.

[31] Microsoft. *CSharp Language Spesification 5.0*. Heinäkuu 2013. URL: https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf (viitattu 06. 07. 2018).

[32] Microsoft. *.NET Compiler Platform Roslyn Overview*. 2018. URL: https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview (viitattu 20. 08. 2018).

[33] Microsoft. *Assembly Manifests*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/sbscs/assembly-manifests (viitattu 23. 08. 2018).

[34] Microsoft. *COM Technical Overview*. 2018. URL: https://docs.microsoft.com/en-gb/windows/desktop/com/com-technical-overview (viitattu 26. 02. 2019).

[35] Microsoft. *.NET*. URL: https://dotnet.microsoft.com/ (viitattu 12. 12. 2018).

[36] Microsoft. *AppDomain.GetAssemblies Method*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.appdomain.getassemblies?redirectedfrom=MSDN&view=netframework-4.7.2%5C#System_AppDomain_GetAssemblies.

[37] Microsoft. *Assembly.GetTypes Method*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.gettypes?redirectedfrom=MSDN&view=netframework-4.7.2%5C#System_Reflection_Assembly_GetTypes (viitattu 31. 10. 2018).

[38] Microsoft. *ClassDeclarationSyntax Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.csharp.syntax.classdeclarationsyntax?view=roslyn-dotnet (viitattu 28. 01. 2019).

[39] Microsoft. *Compilation Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.compilation?view=roslyn-dotnet (viitattu 03. 12. 2018).

[40] Microsoft. *Document Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.document?view=roslyn-dotnet (viitattu 03. 12. 2018).

[41] Microsoft. *IEquatable Interface*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.iequatable-1?view=netframework-4.7.2 (viitattu 03. 12. 2018).

[42] Microsoft. *Microsoft.CodeAnalysis.CSharp.Syntax Namespace*. (Viitattu 20. 02. 2019).

[43] Microsoft. *Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace*. URL: http://source.roslyn.io/%5C#Microsoft.CodeAnalysis.Workspaces.MSBuild/MSBuild/MSBuildWorkspace.cs,751dabb506683b7c (viitattu 03. 12. 2018).

[44] Microsoft. *ObservableCollection Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.observablecollection-1?view=netframework-4.7.2 (viitattu 03. 12. 2018).

[45] Microsoft. *Project Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.project?view=roslyn-dotnet (viitattu 03. 12. 2018).

[46] Microsoft. *Reflection in the .NET Framework*. URL: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection (viitattu 26. 02. 2019).

[47] Microsoft. *ReplaceNode Method*. URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.syntaxnodeextensions.replacenode?view=roslyn-dotnet (viitattu 13. 12. 2018).

[48] Microsoft. *SemanticModel Class*. viitattu 03.12.2018. URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.semanticmodel?view=roslyn-dotnet.

[49] Microsoft. *SyntaxFactory Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.csharp.syntaxfactory?view=roslyn-dotnet (viitattu 03. 12. 2018).

[50] Microsoft. *SyntaxTree Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.syntaxtree?view=roslyn-dotnet (viitattu 03. 12. 2018).

[51] Microsoft. *System.Attribute Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.attribute?view=netframework-4.7.2 (viitattu 12. 12. 2018).

[52] Microsoft. *System.Reflection Namespace*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.reflection?view=netframework-4.7.2 (viitattu 12. 12. 2018).

[53] Microsoft. *System.Runtime.Serialization Namespace*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization?view=netframework-4.7.2 (viitattu 03. 12. 2018).

[54] Microsoft. *System.Type Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.type?view=netframework-4.7.2 (viitattu 12. 12. 2018).

[55] Microsoft. *System.Xml.Serialization Namespace*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization?view=netframework-4.7.2 (viitattu 03. 12. 2018).

[56] Microsoft. *What is a DLL?* URL: https://support.microsoft.com/en-au/help/815065/what-is-a-dll (viitattu 30. 10. 2018).

[57] Microsoft. *Workspace Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.workspace?view=roslyn-dotnet (viitattu 12. 12. 2018).

[58] Microsoft, M. Hoffman, M. Jones, L. Latham, R. Petrusha ja M. Wenzel. *Assembly Contents*. 2017. URL: https://docs.microsoft.com/en-us/dotnet/framework/app-domains/assembly-contents (viitattu 02. 11. 2018).

[59] Microsotf. *Interface (C# Reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface (viitattu 26. 02. 2019).

[60] MuleSoft. *RAML -The simplest way to design APIs*. URL: https://raml.org/ (viitattu 03. 12. 2018).

[61] *Notepad++ Compare Plugin*. URL: https://sourceforge.net/projects/npp-compare/ (viitattu 12. 12. 2018).

[62] H. Oberdiek ja S. Rahtz. *Hyperref – Extensive support for hypertext in LATEX*. URL: https://ctan.org/tex-archive/macros/latex/contrib/hyperref (viitattu 23. 08. 2018).

[63] *Open Data Description Language (OpenDDL)*. URL: http://openddl.org/ (viitattu 30. 10. 2018).

[64] A. Oy. *Atostek eRA*. URL: https://www.atostek.com/era/ (viitattu 25. 02. 2019).

[65] PlantUML. *PlantUML in a nutshell*. URL: http://plantuml.com/ (viitattu 03. 12. 2018).

[66] M. P. Robillard ja R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering* 16.6 (2011), 703–732. DOI: 10.1007/s10664-010-9150-8.

[67] SmartBear Software. *Swagger Inspector*. URL: https://swagger.io/tools/swagger-inspector/ (viitattu 03. 12. 2018).

[68] SmartBear Software. *SwaggerHub - Hosted, Interactive API Documentation*. URL: https://swagger.io/tools/swaggerhub/hosted-api-documentation/ (viitattu 03. 12. 2018).

[69] Smartbear Solutions. *Swagger Codegen*. URL: https://github.com/swagger-api/swagger-codegen (viitattu 14. 02. 2019).

[70] S. Software. *Swagger - The Best APIs are Built with Swagger Tools*. URL: https://swagger.io/ (viitattu 04. 12. 2018).

[71] *Swagger2Markup*. URL: https://github.com/Swagger2Markup/swagger2markup (viitattu 03. 12. 2018).

[72] The OpenAPI Spesification. *The OpenAPI Specification*. URL: https://github.com/OAI/OpenAPI-Specification (viitattu 04. 12. 2018).

[73] A. Troelsen. *Pro C# 5.0 and the .NET 4.5 Framework*. Sixth; Sixth. Berkeley, CA: Apress, 2012. ISBN: 1430242337. DOI: 10.1007/978-1-4302-4234-5.

[74] *Unified Modelling Language*. URL: https://www.omg.org/spec/UML/2.5.1/ (viitattu 26. 02. 2019).

[75] D. Van Heesh. *Doxygen*. URL: http://www.doxygen.nl/index.html (viitattu 03. 12. 2018).