Import/Export of URN Models in Z.151 XML File Format with jUCMNav

Yan Gao

CSI 6900 report submitted to Prof. Daniel Amyot
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Under the auspices of the Ottawa-Carleton Institute for Computer Science

uOttawa

University of Ottawa
Ottawa, Ontario, Canada
January 2010

© Yan Gao, Ottawa, Canada, 2010

Acknowledgment

I would like to give a special thank you to my supervisor, Dr Daniel Amyot, for his guidance and support. I thank him for the challenges he brought upon me, especially the challenge to write the report. ☺ His enthusiasm towards his work is a huge inspiration for me.

I owe a special thank you to my boyfriend Alexandre Martel.

Finally, many thanks for the snowing weather during 2009 Christmas and 2010 New Year. Ottawa is beautiful when it is not so cold!

Table of Contents

Acknowledgment .... **i**

Table of Contents .... **ii**

List of Figures .... **iv**

List of Tables .... **v**

List of Acronyms .... **vi**

Chapter 1. Introduction **1**

1.1. Motivation *1*

1.2. Contributions *1*

1.3. Report Outline *1*

Chapter 2. Background **3**

2.1. User Requirement Notation (URN) *3*

2.2. jUCMNav Project *3*

2.3. JAXB 2.0 .... *4*

Chapter 3. Import/Export between Z.151 and jUCMnav **5**

3.1. Introduction *5*

3.2. Comparison of Different Approaches *5*

3.3. Approach to Import/Export of Models in Z.151 Format *7*

3.4. Configurations *8*

3.4.1 Setting Up the JAXB Environment 8

3.4.2 Adding Extension Points in jUCMNav 10

3.5. Compiling Z.151 Schema Elements into Java Classes *11*

3.6. Package Structure *12*

3.7. Naming Conventions *14*

3.8. Classes in Packages Marshal and UNMarshal *15*

3.8.1 Classes in Package marshal 15

3.8.2 Classes in Package unmarshal 18

3.8.3 Class MHandler 21

3.8.4 Class EObjectImplUMHandler 22

3.9. Code Generator for Creating Skeleton Classes 23

Chapter 4. Handling Differences between jUCMNav and Z.151 Metamodels 25

4.1. In jUCMNav's Metamodel but not in Z.151's Metamodel 25

4.1.1 KPIModel in jUCMNav's Metamodel 25
4.1.2 Classes Defined but Unused in jUCMNav's Metamodel 27
4.1.3 Classes Defined in jUCMNav Metamodel, but not in Z.151 28
4.1.4 Interface Defined in jUCMNav's Metamodel 29
4.1.5 Attributes Defined in jUCMNav's Metamodel, but not in Z.151 30
4.1.6 ComponentKind has Other in jUCMNav, but not in Z.151 31
4.1.7 IntentionalElementType has INDICATOR in jUCMNav, but not in Z.151 32
4.1.8 DeviceKind has Other in jUCMNav, but not in Z.151 32
4.1.9 ScenarioStartPoint and ScenarioEndPoint in jUCMNav, but not in Z.151 32

4.2. In Z.151's Metamodel but not in jUCMNav's Metamodel 33

4.2.1 ID in Z.151, but not in jUCMNav 33
4.2.2 GRLnode Has a Size (Width/Heigth) in Z.151, but not in jUCMNav 34
4.2.3 Z.151 Supports Visualization of XOR/IOR as Means-end 34
4.2.4 Workload Decomposed in Z.151, but not in jUCMNav 34
4.2.5 Z.151 supports Movable GRL LinkRef Labels, but not jUCMNav 35

4.3. In Both Metamodels, but Unused in jUCMNav 36

4.3.1 Component Types 36
4.3.2 Collapsed Actor References 36

4.4. Mismatches between the Two Metamodels 36

4.4.1 Differences between Belief and Belief Link 36
4.4.2 Concern 37
4.4.3 DecompositionType 37
4.4.4 Association between EvaluationStrategy and StrategiesGroup 38
4.4.5 Role includingComponent for Component Self-association 38
4.4.6 Association between ScenarioDef and StrategiesGroup 38
4.4.7 URN Data Model 38

Chapter 5. Testing 39

5.1. Test Items 39

5.2. Approach 40

5.3. Test Cases 40

Chapter 6. Conclusions 42

6.1. Contributions 42

6.2. Future work 42

References 44

Appendix A: Sample Generated Skeleton Class 46

List of Figures

Figure 1 GRL Model for Supporting Z.151 import/export Using JAXB 6
Figure 2 Model-to-Model Transformation [9] 7
Figure 3 Import/Export of URN Models in Z.151 XML Files 8
Figure 4 Add JAR file in Java Build Path 9
Figure 5 Configure Java Compiler JDK Compliance Level to Java 1.5 10
Figure 6 Package Diagram for Export 13
Figure 7 Package Diagram for Import 14
Figure 8 seg.jUCMNav.importexport.Z151.marshal Class Hierarchy (1) 16
Figure 9 seg.jUCMNav.importexport.Z151.marshal Class Hierarchy (2) 17
Figure 10 seg.jUCMNav.importexport.Z151.unmarshal Class Hierarchy (1) 20
Figure 11 seg.jUCMNav.importexport.Z151.unmarshal Class Hierarchy (2) 21
Figure 12 Skeleton Classes Directory Structure 24
Figure 13 KPIModelElements in jUCMNav’s Metamodel 26
Figure 14 KPIModelLinks in jUCMNav’s Metamodel 26
Figure 15 KPIStrategy in jUCMNav’s Metamodel 27
Figure 16 GRL Strategy in jUCMNav’s Metamodel 28
Figure 17 Class FailurePoint and Anything in jUCMNav Metamodel 29
Figure 18 URNcore/URNabstract in jUCMNav Metamodel 30
Figure 19 ScenarioStartPoint and ScenarioEndPoint in jUCMNav’s Metamodel 33
Figure 20 ScenarioStartPoint and ScenarioEndPoint not in Z.151’s Metamodel 33
Figure 21 Workload in jUCMNav’s Metamodel 35
Figure 22 Workload in Z.151’s Metamodel 35
Figure 23 jUCMNav Belief 37
Figure 24 Test Models 40
Figure 25 Z151importexport Test Directory 41

List of Tables

Table 1 Interfaces and Classes. 30
Table 2 Attributes in jUCMNav’s Metamodel but not in Z.151 31

List of Acronyms

Acronym	Definition
ATL	ATLAS Transformation Language
EMF	Eclipse Modeling Framework
GRL	Goal-oriented Requirement Language
ITU-T	International Telecommunication Union
JAXB	Java Architecture for XML Binding
jUCMNav	Java Use Case Map Navigator (URN tool)
KLOC	Kilo Lines of Code
UCM	Use Case Maps
UML	Unified Modeling Language
URN	User Requirements Notation
XML Declaration	Typically appears as the first line in an XML document

Chapter 1. Introduction

1.1. Motivation

This report presents the implementation of the import/export of User Requirements Notation (URN) models in Z.151 format with jUCMNav. Z.151 [4] is an ITU-T standard for the URN language. jUCMNav [7] is an Eclipse plug-in for the design and analysis of URN models. The current jUCMNav files are in XML/XMI format [11] complying with jUCMNav's metamodel [8]. To better comply with the Z.151 standard, jUCMNav needs to support importing/exporting of URN models in Z.151 XML files.

1.2. Contributions

This project provides an import/export mechanism for URN models in Z.151 format with jUCMNav that has been implemented using JAXB. The implementation supports a bi-directional transformation between the standard Z.151 metamodel (November 2008) and the jUCMNav metamodel version 0.23 (December 21, 2009). Through this project, jUCMNav becomes the first tool in the world to support the Z.151 file format.

The import and export were tested with a comprehensive set of existing URN models provided by Prof. D. Amyot's team.

Along the way, several errors in the URN standard were also detected and fixed. These corrections will be incorporated in the URN standard.

1.3. Report Outline

This report is structured as follows. Chapter 2 presents background concepts related to URN, jUCMNav and JAXB. In chapter 3, we present the implementation details: rationale for the choice of the technology to be used in the implementation, configuration to set up the project, package and class structure, and explanation of attributes and methods in the super classes. Chapter 4 discusses major differences between jUCMNav's metamodel

and Z.151's metamodel and reports on how we handle these differences in the implementation. Chapter 5 covers our validation approach based on testing and chapter 6 discusses conclusions and future work.

Chapter 2. Background

This chapter describes the background on the User Requirements Notation (URN), jUCMNav and JAXB.

2.1. User Requirement Notation (URN)

The User Requirements Notation (URN) defines a set of modeling concepts and notations, which software or requirement engineers can use for the elicitation, analysis, specification, and validation of requirements. URN consists in two subsets languages: Goal-oriented Requirement Language (GRL) for goal-oriented modeling, and Use Case Map (UCM) for scenario modeling. URN became an ITU-T standard (Z.151) in November 2008 [4]. This standard dictates the use of an XML-based interchange format for URN models, and an XML schema is specified. However, no tool had implemented support for this format before this project.

2.2. jUCMNav Project

jUCMNav is an Eclipse plug-in, which provides users with a graphical editor to draw and analyze URN models [7]. Requirements engineers can draw their functional requirement and non-functional requirements in graphical URN models, and then analyze these URN models using scenario traversal and goal evaluation. A URN model can be transformed to other languages such as MSC, SDL, TTCN and UML.

The project has started in year 2005 and it is very active open source project. Project statistics provided by Ohloh[1] indicate that 10 different developers contributed around 50,000 commented lines of code (LOC) to jUCMNav's code base (which is now close to 275 KLOC) since the beginning of my project, 8 months ago. In addition, jUCMNav's metamodel was modified 4 times during this period.

[1] http://www.ohloh.net/p/11712?p=jUCMNav, accessed January 5, 2010

2.3. JAXB 2.0

Java Architecture for XML Binding (JAXB) provides two main features: marshalling of Java objects to XML files and unmarshalling of XML files to Java objects. The tutorial on JAXB’s home website [9] provides the basic design structure of our implementation.

Chapter 3. Import/Export between Z.151 and jUCMnav

3.1. Introduction

The URN tool jUCMNav is developed from an EFM-based metamodel for URN. jUCMNav contains around 80 KLOC of Java code generated automatically from the metamodel. The metamodel for standard URN Z.151 [4] has a corresponding standard XML schema used for persistence. There are substantial differences between jUCMNav's metamodel (created in 2005 and in constant evolution) and the standard URN Z.151 metamodel (standardized in 2008); for example, jUCMNav's metamodel contains non-standard exploratory concepts such as Key Performance Indicator Model (KPI model) [2] and facilities for code reuse. The metamodel for URN standard Z.151 was established in 2008. It is independent from code reuse and covers exactly whatever is standardized.

We need to provide jUCMNav with the functionalities to import and export URN models in the format standardized in URN standard Z.151. This will allow users to create and analysis URN models with jUCMNav and also be able to use the same URN models in other URN tools complying with URN standard Z.151.

3.2. Comparison of Different Approaches

In this section, we compare two candidate approaches to implement Z.151 import/export: JAXB and ATL.

JAXB is Java-based. Since the jUCMNav project is implemented in Java, it is inexpensive to implement the required new functionalities in the same language: Developers already have knowledge of Java. Integration with the jUCMNav project is easy (see 3.4). Import/export Z.151 is mainly about objects creation and attributes assignment. Most of the code is quite similar and we can create code to generate Java files with skeleton code structure. This helps us not to miss any classes or any attributes for attributes

assignment. It is easy to maintain and update to be aligned with future changes in the jUCMNav and Z.151 metamodels. Adding and removing classes or attributes are trivial operations in Java, and they do not require much testing since actions are isolated from each other.

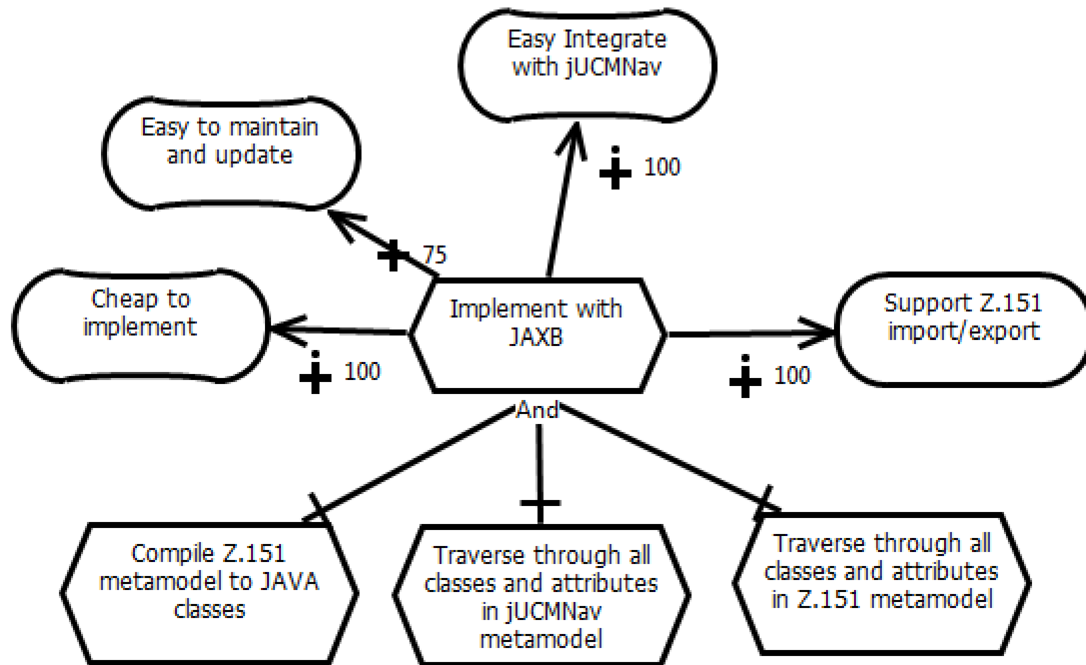

Figure 1 GRL Model for Supporting Z.151 import/export Using JAXB

ATLAS Transformation Language (ATL) is a model transformation language designed by OBEO and INRIA (France) [3][1]. It is a hybrid declarative/imperative language. It is harder to implement our import/export in ATL than in JAXB. First of all, very few developers have the knowledge ATL and learning it is expensive. Second, model-to-model transformations involve several layers (see Figure 2). For ATL application, source models and target models are in the "MSI" format, source metamodels and target metamodels are in the "KM3" [6] format, and the transformation model is composed of ATL rules. In addition, to take XML files as input or output, an injector and an extractor (Ant scripts) have to be implemented. The integration with the jUCMNav project is not trivial either. To launch transformations programmatically, we need to write an Ant script using the AM3 Ant Tasks. It is hence difficult to maintain the support of Z.151 import/export as all these layers need to be updated upon any change in the jUCMNav or

Z.151 metamodels. Finally, ATL rules are somewhat hard to write and read for beginners.

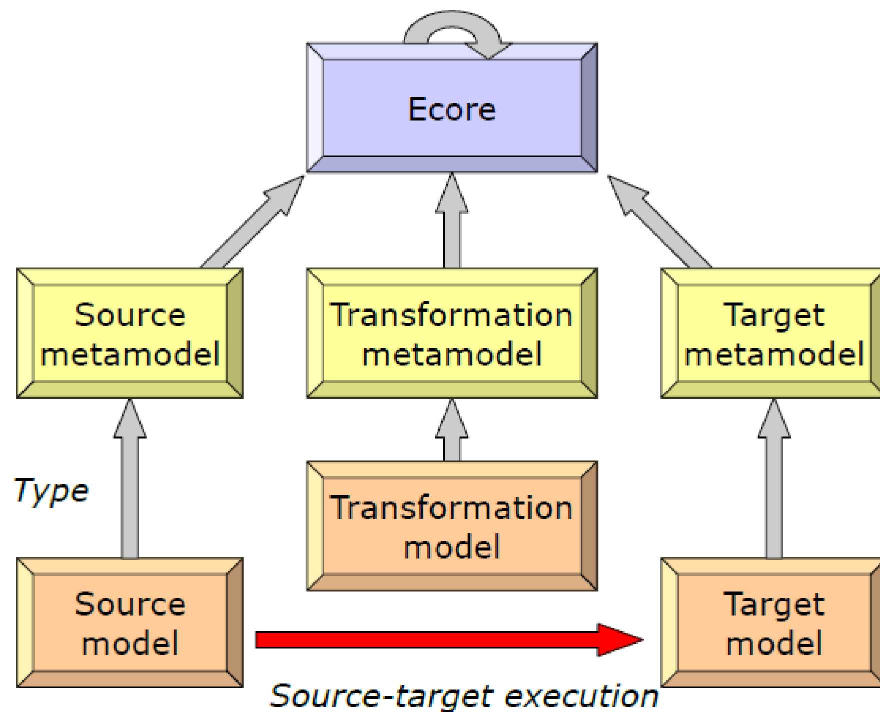

Figure 2 Model-to-Model Transformation [9]

Given the above observations, using JAXB to implement the Z.151 import/export is a better option compared with ATL.

3.3. Approach to Import/Export of Models in Z.151 Format

The Z.151 metamodel is captured as an XML schema in the Z.151 standard. We store it in the Z151.xsd file. JAXB provides a solution for implementation with benefits of cheap implementation, easy configuration, and easy maintenance. We use JAXB to import and export URN models in Z.151 XML files. Figure 3 describes the architecture used to import/export URN models in Z.151 XML files.

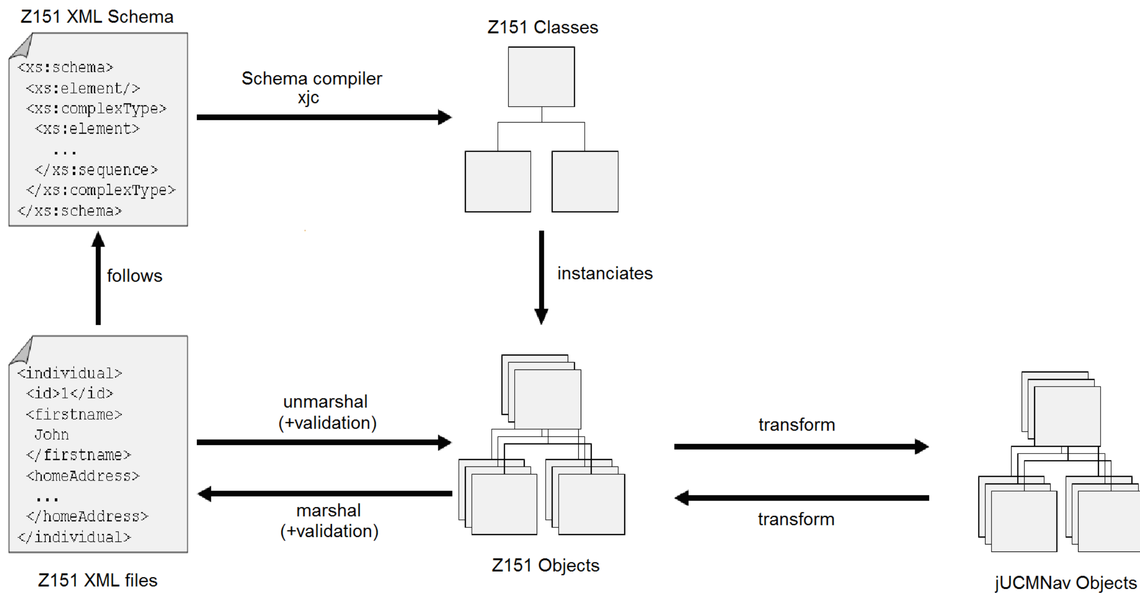

Figure 3 Import/Export of URN Models in Z.151 XML Files

We use the schema compiler `xjc` [12] to compile Z151.xsd into Z.151 Java classes. These generated Z.151 Java classes are used for unmarshalling Z.151 XML files to Z.151 objects during the import and for marshalling Z.151 objects to XML files during the export. Then, the import programmatically transforms in-memory Z.151 objects to jUCMNav (EMF) objects whereas the export transforms in-memory jUCMNav (EMF) objects to Z.151 objects.

3.4. Configurations

3.4.1 Setting Up the JAXB Environment

To make JAXB available for the jUCMNav project, we need to add two jar files *jaxb-api.jar* and *jsr173_1.0_api.jar* from JAXB version 2.0 (or above) into the /seg.jUCMNav/lib folder and also add them into Java's build path (see Figure 4). Please visit JAXB [5][9] home page for downloading.

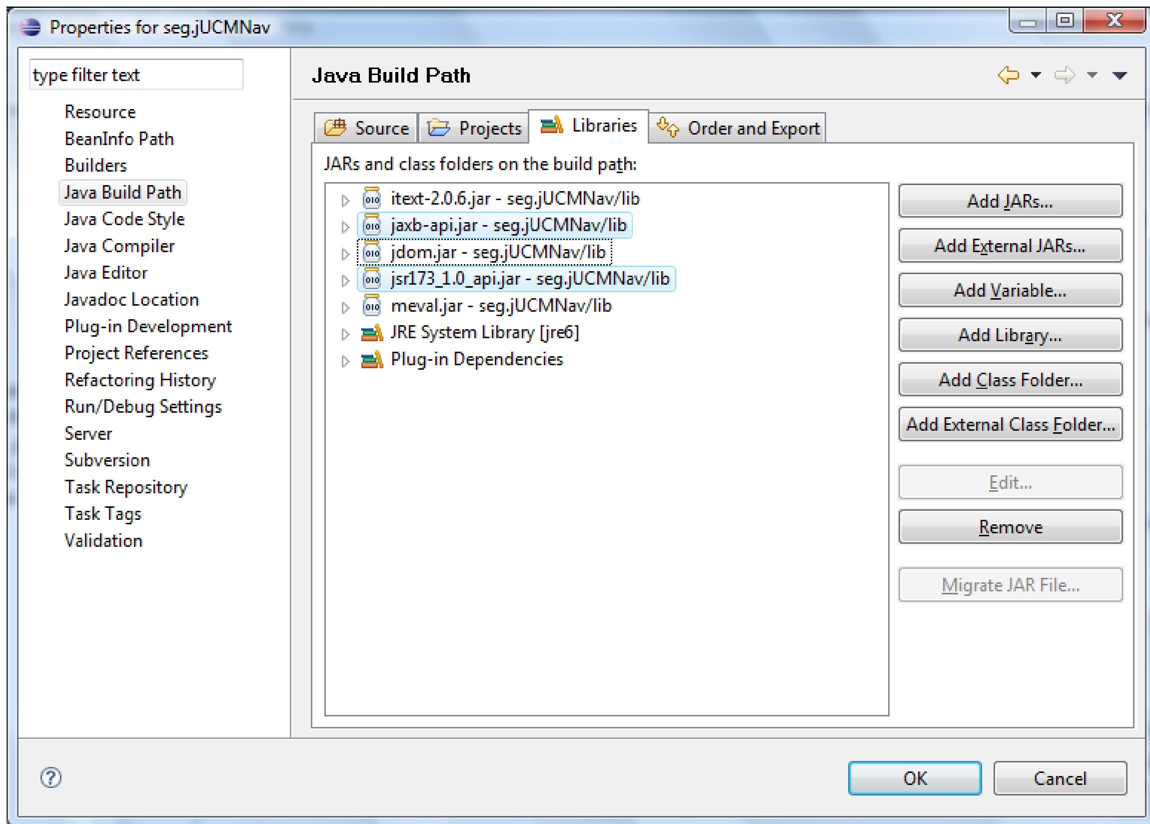

Figure 4 Add JAR file in Java Build Path

It is better to use JAXB 2.0 (or above) than JAXB 1.0 because JAXB 2.0 significantly reduces the number of generated classes. However, using JAXB 2.0 forces the jUCMNav project to be compliant with Java 1.5 rather than Java 1.4. We have no choice to configure Java compiler JDK compliance level to 1.5 (see Figure 5). Since the project is supposed to move from Java 1.4 to 1.5 soon anyway, this change is in line with the project's goals and constraints.

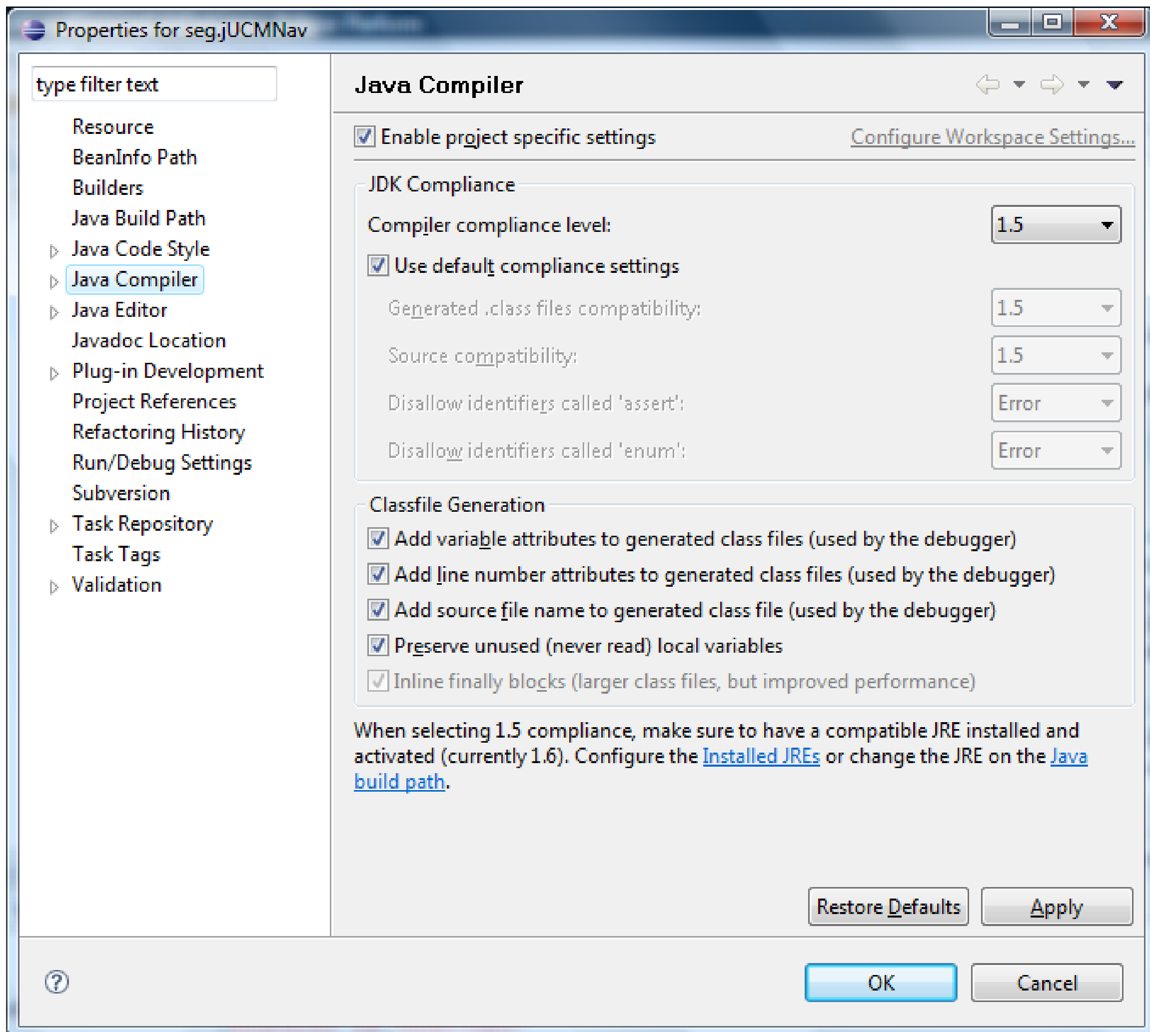

Figure 5 Configure Java Compiler JDK Compliance Level to Java 1.5

3.4.2 Adding Extension Points in jUCMNav

We added this line to the *seg.jUCMNav.plugin.properties* file.

```
Z151 = URN Standard Z151 (*.z151)
```

We also added the following into the import and extension points sections of the *plugin.xml* file:

```
<extension
        id="seg.jUCMNav.importexport.ExportURN"
        name="%exportToImage"
        point="seg.jUCMNav.URNExport">
...
```

```
        <exporter
              class="seg.jUCMNav.importexport.z151.ExportZ151"
              extension="z151"
              id="seg.jUCMNav.ExportZ151"
              name="%Z151"
              useStream="false"/>
     </extension>


     <extension
           id="seg.jUCMNav.importexport.ImportURN"
           name="%import"
           point="seg.jUCMNav.URNImport">
  ….
        <importer
              class="seg.jUCMNav.importexport.z151.ImportZ151"
              extension="z151"
              id="seg.jUCMNav.ImportZ151"
              importInSelectedFile="true"
              name="%Z151"
              useStream="true"/>
     </extension>
```

After inserting the above into the *plugin.xml* file, a new selection for importing and exporting URN standard Z.151 files becomes available for users in jUCMNav through the import/export interface.

3.5. Compiling Z.151 Schema Elements into Java Classes

The Z.151 metamodel is defined in a schema (saved in the *Z.151.xsd* file). To support marshalling/unmarshalling between XML files and Java Z.151 objects programmatically, our first step is to compile the elements in *Z151.xsd* into a set of Java classes. *jaxb-xjc.jar* is the JAR file used to compile Z151.xsd file to Java classes. It is in the lib directory of the downloaded JAXB file. Below is a sample command line to compile the Z.151 schema into Java classes:

```
java -jar jaxb-xjc.jar -d c:\tmp z151.xsd
     -p seg.jUCMNav.importexport.z151.generated
```

After running this command line, we find Java classes in the *c:\tmp* folder, and all generated Java classes are part of package *seg.jUCMNav.importexport.z151.generated*. The following explains command line options:

-d <dir>

By default, the XJC binding compiler will generate the Java content classes in the current directory. Use this option to specify an alternate output directory. The output directory must already exist; the XJC binding compiler will not create it for you.

p <pkg>

Specifying a target package via this command-line option overrides any binding customization for package name and the default package name algorithm defined in the specification.

3.6. Package Structure

There are four packages in the implementation:

- `seg.jUCMNav.importexport.Z151`
- `seg.jUCMNav.importexport.Z151.marshal`
- `seg.jUCMNav.importexport.Z151.unmarshal`
- `seg.jUCMNav.importexport.Z151.generated`

The package `seg.jUCMNav.importexport.Z151` has two Java classes `ImportZ151` and `ExportZ151`. Class `ImportZ151` implements the `URNImport` extension point. It is used to load a Z.151 XML file and create an instance of `urn.URNspec`. Class `Export151` implements the `URNExport` extension point. It takes an instance of `urn.URNspec` and exports it into a Z.151 XML file.

The package `seg.jUCMNav.importexport.Z151.generated` includes all Java classes generated from schema defined in Z.151 with using the JAXB schema compiler `xjc`. The Java classes in this package present the Z.151 metamodel and they have preserved the hierarchy relationships defined in the Z.151 metamodel. We need to point out that we shall never modify the code in these classes, especially due to the fact we have to re-compile the schema to generate new Java classes for any future changes in the Z.151 metamodel.

The package `seg.jUCMNav.importexport.marshal` contains all Java classes which are used to transform instances of jUCMNav classes to instances of Z.151 classes. This is the transformation of objects during the export process. The package depends on Z.151.generated, URN, URNcore, GRL, and UCM packages. Whenever Z.151.generated, URN, URNcore, GRL, and UCM package change, we need to update classes in this

package to reflect the changes and ensure the export works properly. Figure 6 is a package diagram showing dependencies among packages involving in the export process.

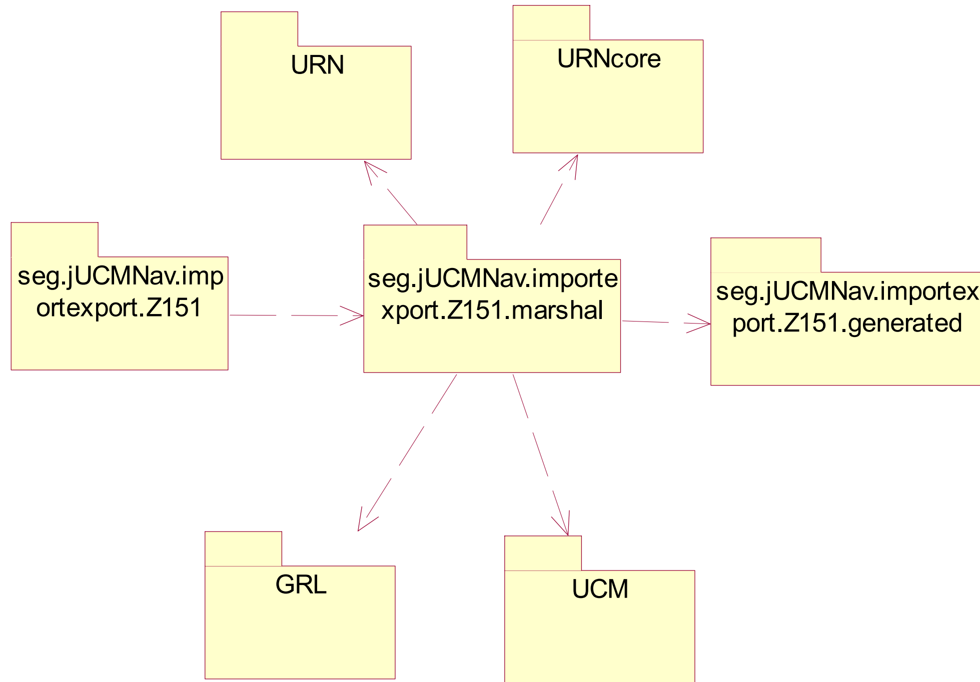

Figure 6 Package Diagram for Export

The package `seg.jUCMNav.importexport.unmarshal` contains all Java classes which are used to transform instances of Z.151 classes to instances of jUCMNav classes. This is the transformation of objects during the import process. The package depends on Z.151.generated, URN, URNcore, GRL, and UCM packages. Whenever changes occur in Z.151.generated, URN, URNcore, GRL, and UCM packages, we will find we are in the situation to update classes in this package to reflect the changes and make the import work properly. Figure 7 is a package diagram showing dependencies among packages involving in import process.

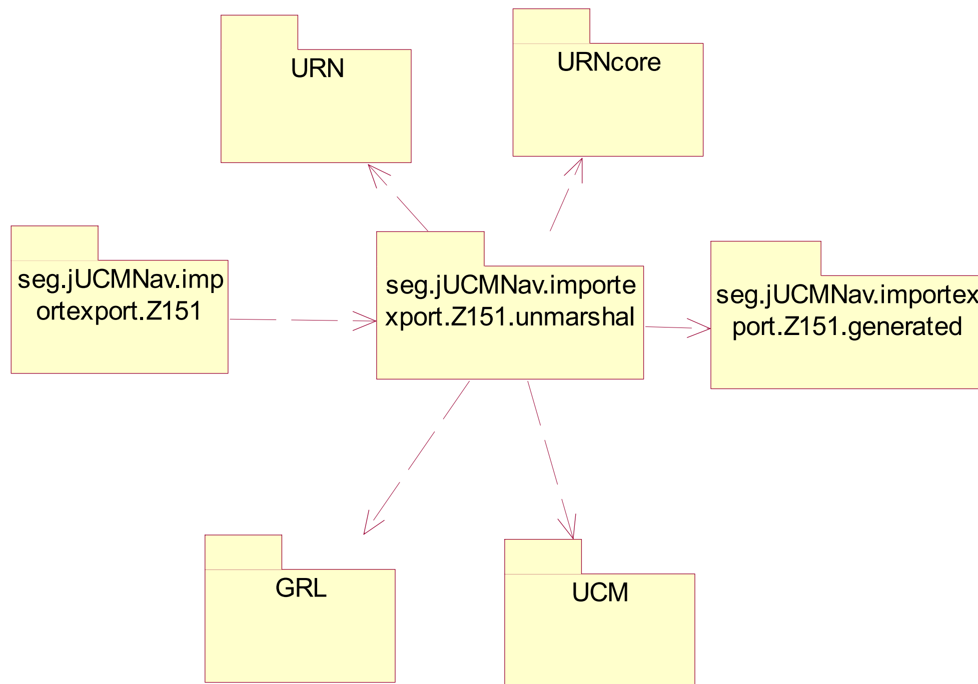

Figure 7 Package Diagram for Import

3.7. Naming Conventions

Each class in package `seg.jUCMNav.importexport.Z151.marshal` is a handler class used to generate output objects of a specific class in the Z.151 metamodel with input objects of corresponding jUCMNav classes. Handler classes in this package are named in this way: the name of the input jUCMNav class concatenated with a string "MHandler". For example, handler class is named as ActorMHandler takes an object of class `grl.Actor` and creates an object of class `seg.jUCMNav.importexport.Z151.generated.Actor`.

Each class in package `seg.jUCMNav.importexport.Z151.unmarshal` is a handler class to take input objects of a specific class in the Z.151 metamodel to generate output objects of corresponding jUCMNav objects. We name the handler classes in this package with the name of the input Z.151 class concatenated with a string "UMHandler". For example, handler class is named as ActorUMHandler because it takes an object of

class `seg.jUCMNav.importexport.Z151.generated.Actor` and creates an object of class `grl.Actor`.

3.8. Classes in Packages Marshal and UNMarshal

As we have mentioned previously, the Z.151 metamodel and the jUCMNav metamodel define classes with hierarchy relationships. Hierarchy relationships increase code reuse, minimize the amount of repetitive code and increase maintainability. Package `seg.jUCMNav.importexport.Z151.marshal` is in charge of transforming Z.151 objects to jUCMNav objects and package `seg.jUCMNav.importexport.Z151.unmarshal` does the transformation from jUCMNav objects to Z.151 objects. We organize classes in these two packages with hierarchy relationships.

3.8.1 Classes in Package marshal

Package `seg.jUCMNav.importexport.Z151.marshal` is used for jUCMNav-to-Z.151 object transformation. Each class in this package is designed to take input objects of a single class in jUCMNav metamodel and create output Z.151 objects of an appropriate Z.151 class. All classes in the jUCMNav metamodel have a corresponding handler class in this package. All classes in the package marshal are direct sub-classes or descendent classes of MHandler (see Figure 8, Figure 9). MHandler has common methods required by all other classes.

One handler class per class in the jUCMNav metamodel has been written for jUCMNav-to-Z.151 object transformations. If a jUCMNav class implements hierarchy relationships, a group of handler classes which follows a similar hierarchy tree will be used to transform the objects of the jUCMNav class to the objects of corresponding Z.151 classes. For example, handler classes ContributionMHandler, ElementLinkMhandler, GRLmodelElementMHandler, and URNmodelElementMHandler are used to convert an object of jUCMNav class Contribution to an object of Z.151 class Contribution. The reason why we need a group of handler classes is that Class Contribution inherits class ElementLink, class ElementLink inherits GRLmodelElement, and GRLmodelEle-

ment inherits URNmodelElement and each handler class only deals with the transformation of the attributes which are defined within the Z.151 class but not the inherited attributes.

Figure 8 seg.jUCMNav.importexport.Z151.marshal Class Hierarchy (1)

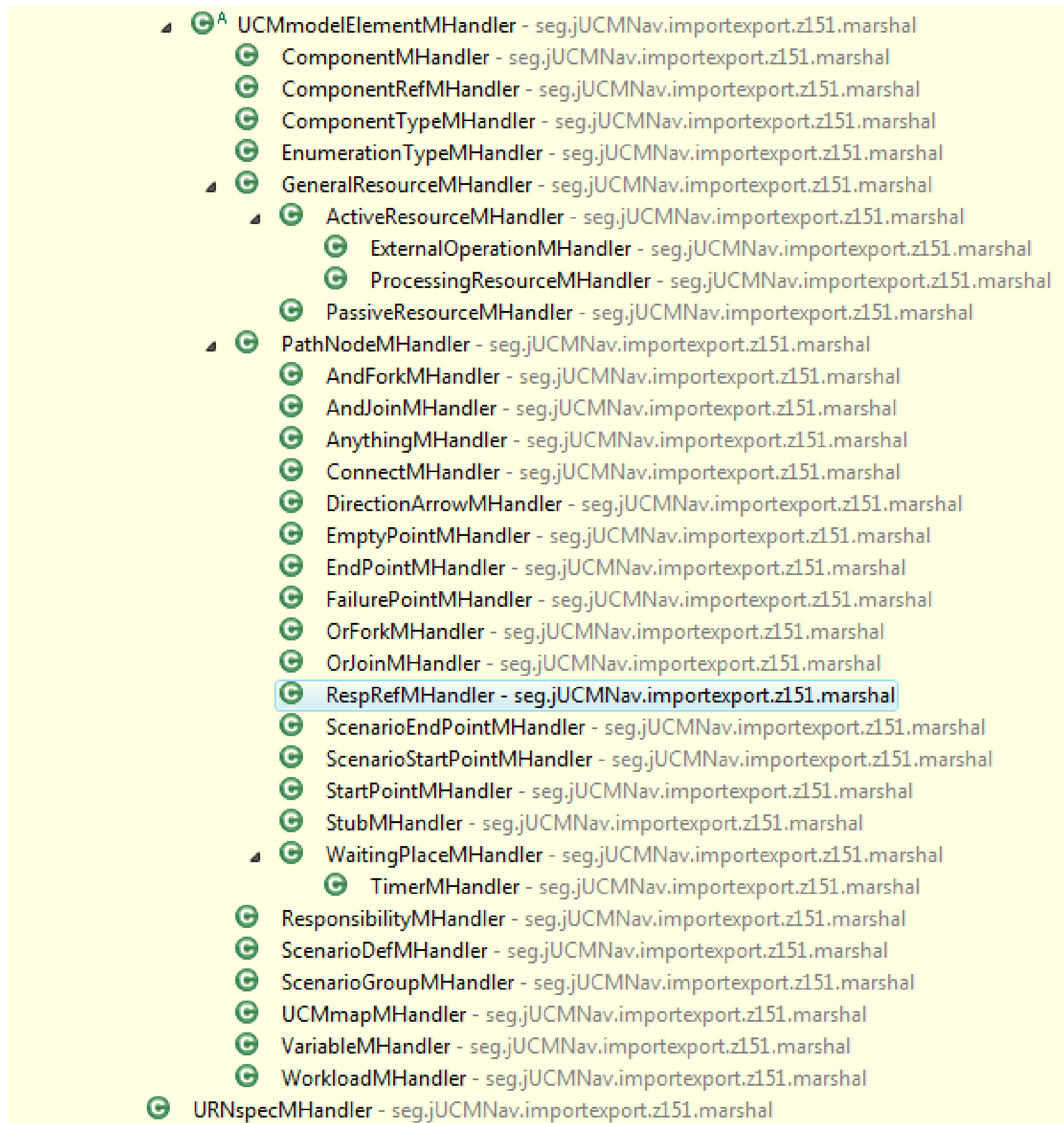

Figure 9 seg.jUCMNav.importexport.Z151.marshal Class Hierarchy (2)

All handler classes implements the abstract method handle() in class MHandler. It takes a jUCMNav object, creates a corresponding Z.151 object, and transforms the value of attributes from the source jUCMNav object to the result Z.151 object. Each handler class only handles the attributes defined within to Z.151 classes, but it does not handle the attributes inherited from supper classes. This is by designed to reduce the amount of duplicate code and increase maintainability.

The signature of method implemented by all classes is `public Object handle(Object obj, Object target, boolean isFullConstruction)`. Object obj is a source jUCMNav object. Object target is the target Z.151 object passing in this method, usually null. And boolean isFullConstruction is to indicate whether we transform the values of attributes from jUCMNav object to returned Z.151 object or not. The method returns a Z.151 object transformed from jUCMNav object with/without assigned attributes depending on the value of the indicator isFullConstruction. If isFullConstruction is true, attributes are transformed from source jUCMNav object to the result Z.151 object.

3.8.2 Classes in Package unmarshal

Package unmarshal contains all handler classes for transforming Z.151 objects to jUCMNav objects. All classes in the package unmarshal are direct sub-classes or descendent classes of EObjectImplUMHandler (see Figure 10, Figure 11). Class EObjectImplUMHandler provides implementation of all common operations required by all other classes. One handler class per class in the Z.151 metamodel has been written for conducting transformations.

All handler classes implement the abstract method handle() in class EObjectImplUMHandlerprovides. It takes a Z.151 object, creates a corresponding jUCMNav object, and transforms the value of attributes from the source Z.151 object to the resulting jUCMNav object. Each handler class only handles the attributes defined within jUCMNav classes, but it does not handle the attributes inherited from jUCMNav super classes. This is designed to reduce the amount of duplicate code and to increase maintainability.

In the case a jUCMNav class implements a hierarchy tree, a group of handler classes which follows a similar hierarchy tree will be used to transform the objects of the Z.151 class to the objects of corresponding jUCMNav classes. For example, we use ContributionUMHandler, ElementLinkUMhandler, GRLmodelElementUMHandler, and URNmodelElementUMHandler to transfer an object of Z.151 class Contribution to jUCMNav class Contribution. Here is the reason why we need a group of handler classes: Class Contribution inherits class ElementLink, class ElementLink inherits GRLmodelElement, and GRLmodelElement inherits URNmodelElement in Z.151 metamodel.

Each handler class only deals with the transformation of the attributes which are defined within the Z.151 class but not the inherited attributes.

The signature of method implemented by all classes is `public Object handle(Object obj, Object target, boolean isFullConstruction)`. Parameter obj is a source Z.151 object. Parameter target is the target jUCMNav object passed in this method, usually null. The boolean isFullConstruction parameter is to indicate whether we transform the values of attributes from Z.151 object to returned jUCMNav object or not. The method returns a jUCMNav object transformed from the Z.151 object with/without assigned attributes depending on the value of the indicator isFullConstruction. If isFullConstruction is true, attributes are transformed from source Z.151 object to the result jUCMNav object.

Figure 10 seg.jUCMNav.importexport.Z151.unmarshal Class Hierarchy (1)

Figure 11 seg.jUCMNav.importexport.Z151.unmarshal Class Hierarchy (2)

3.8.3 Class MHandler

Class MHandler is the super class or ancestor class of all handler classes in package `seg.jUCMNav.importexport.Z151.marshal` and it has static attributes and methods which are used in subclasses or descendent classes.

Static attribute `ourClass2Conv` is an object of type Hashmap. Each handler class has an instance added in Hashmap `ourClass2Conv` with the class of input jUCMNav objects as a key.

Static attribute `id2object` is an object of type Hashmap. When a new output object with an ID is created for the first time, it will be added into `id2object` with the ID as the key. Moreover, the same object will be fetched by using the ID for later use.

Method `handle(Object obj, Object target, boolean isFullConstruction)` is an abstract method, which is implemented by all other classes in package `seg.jUCMNav.importexport.Z151.marshal`. The method accepts a jUCMNav object and returns a Z.151 object.

Method `process(Object obj, Object target, boolean isFullConstruction)` functions as a dispatcher: Based on an input jUCMNav object, an instance of a MHandler class is fetched from Hashmap ourClass2Conv. Then, the method `handle()` of the MHandler is called, and a Z.151 object is created and returned.

Method `process(Object obj, Object target, String methodName, boolean isFullConstruction)` functions similarly to the above method process(), apart from returning an object of type JAXBElement<Object>.

Method `processList(EList list, List<V> targetList, boolean isFullConstruction)` is for processing a list of input jUCMNav objects. Method `processList(EList list, List<JAXBElement<Object>> targetList, String methodName, boolean isFullConstruction)` is similar, apart from the returned object.

3.8.4 Class EObjectImplUMHandler

Class EObjectImplUMHandler is the super class or ancestor class of all handler classes in package `seg.jUCMNav.importexport.Z151.unmarshal` and it has static attributes and methods which are used in subclasses or descendent classes.

Static attribute `ourClass2Conv` is an object of type Hashmap. Each handler class has an instance added in Hashmap `ourClass2Conv` with the class of input Z.151 objects as a key.

Static attribute `id2object` is an object of type Hashmap. When a new output jUCMNav object with an ID is created at the first time, it will be added into `id2object` with the ID as the key. The same object will be fetched by using the ID for later use.

Method `handle(Object obj, Object target, boolean isFullConstruction)` is an abstract method, which is implemented by all other classes in package `seg.jUCMNav.importexport.Z151.unmarshal`. The method accepts a Z.151 object and returns a jUCMNav object.

Method `process(Object obj, Object target, boolean isFullConstruction)` is functioned as a dispatcher: Based on an input Z.151 object, an instance of an EObjectImplUMHandler class is fetched from Hashmap ourClass2Conv. The method `handle()` of the MHandler is called, and a Z.151 object is created and returned.

Method `processList(List<V> list, EList targetList, boolean isFullConstruction)` is for processing a list of input Z.151 objects.

3.9. Code Generator for Creating Skeleton Classes

Classes in Package marshal and unmarshal are used to create new objects and transform the values of the attributes from the input object to the newly created output objects. It is a process of mapping attributes between input objects and output objects. There are more than a hundred handler classes needed to be written and the contents of all classes are quite similar: class definition, method definition, the attribute mapping between input and output classes. We wrote a code generator for the purpose of generating Java files containing the skeleton classes. The generated skeleton Java files has the content of class name, method name, all getters and setters from input object, and related XML segment from Z151.xsd.

We benefit from these skeleton classes with reduced typing, copying, and pasting work. The second benefit is that we can view related XML segment and all getters and setters for input class in the same file. This is very important for us to go through all attributes in Z.151 and jUCMNav metamodel and make sure we do not miss the handling of any attributes. Without these generated skeleton classes, it could be tedious and mistake-prone: imagine how tedious it could be if we needed to type, copy and paste, and look for available getters and setters.

Class `SkeletonClassesGenerator` is a JUnit class. This class provides two JUnit test cases for us to generate skeleton classes: `testMarshalGenerator` and `tes-`

tUNMarshalGenerator. We can run them as JUnit tests and skeleton classes will be created in generatedSkeletonClasses directory. Running testMarshalGenerator method creates skeleton classes for handling each class in Z.151 metamodel. Running testUNMarshalGenerator method creates skeleton classes for handling classes in jUCMNav metamodel and these classes are organized in the same directory structure as in packages in jUCMNav metamodel (see Figure 12). Appendix A describes a sample generated skeleton class.

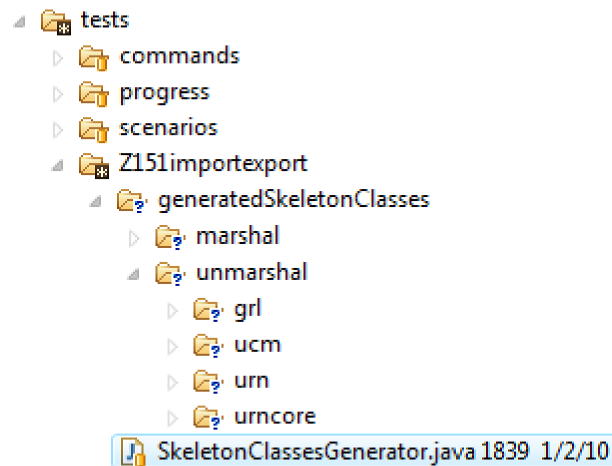

Figure 12 Skeleton Classes Directory Structure

Please note we need to delete the generatedSkeletonClasses directory to ensure a successful jUCMNav build, after running JUnit test cases in testUNMarshalGenerator.

Chapter 4. Handling Differences between jUCM-Nav and Z.151 Metamodels

jUCMNav's metamodel has been designed, implemented, maintained since 2005. There are quite a few differences between the jUCMNav and Z.151 metamodels. The most recent version of jUCMNav metamodel is version 0.23. We will list key differences between jUCMNav metamodel v0.23 and Z.151 metamodel in this section and then address them one by one.

4.1. In jUCMNav's Metamodel but not in Z.151's Metamodel

4.1.1 KPIModel in jUCMNav's Metamodel

Z.151 does not have any support for KPIModel, but jUCMNav supports KPIModel. All classes and relationships, which are specific to KPIModel, will be ignored in the exporting/importing processes. Ignored classes are KPIInformationConfig, KPIInformationElement, KPIEvalValueSet, KPIModlelLinkRef, KPIModleLink. These classes are colored in red (see Figure 13, Figure 14, Figure 15). We will not export the relationships from or to instances of these classes either. The only things that will be exported are instances of Indicators in jUCMNav. They will be exported as instances of IntentionalElement with TASK as IntentionalElementType and with a Metadata where name is "jUCMNav Indicator" and value is "Indicator".

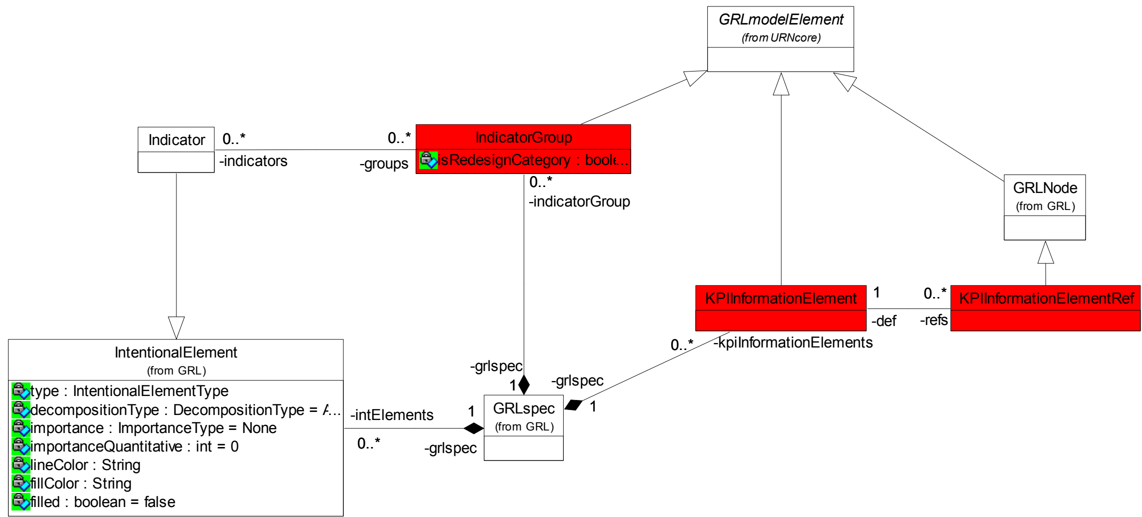

Figure 13 KPIModelElements in jUCMNav's Metamodel

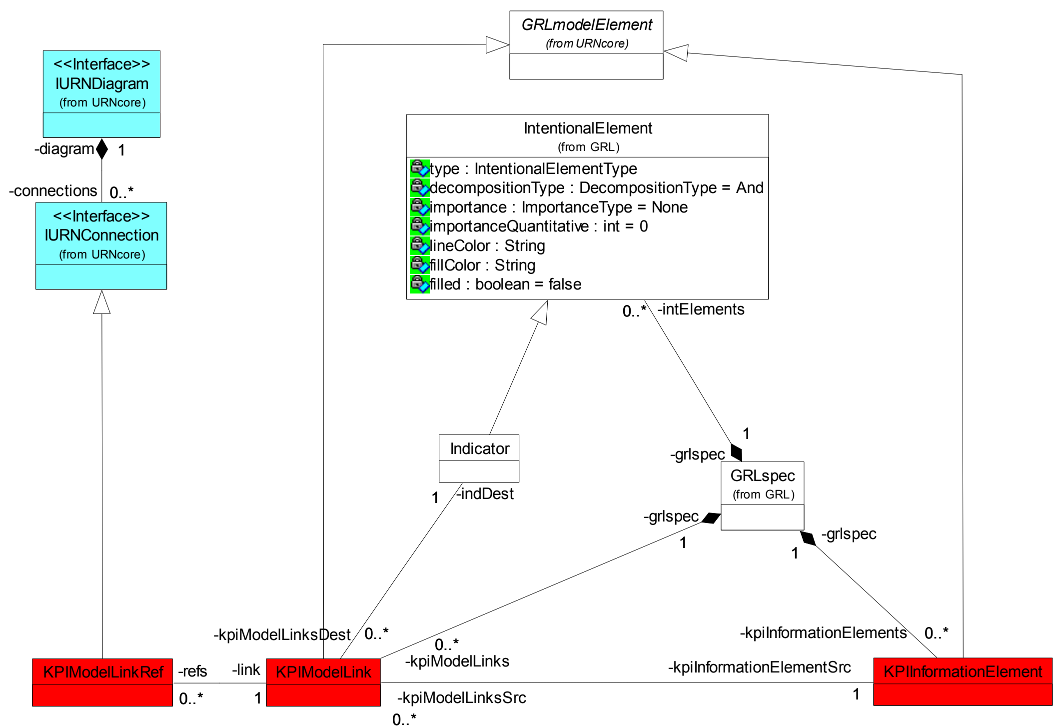

Figure 14 KPIModelLinks in jUCMNav's Metamodel

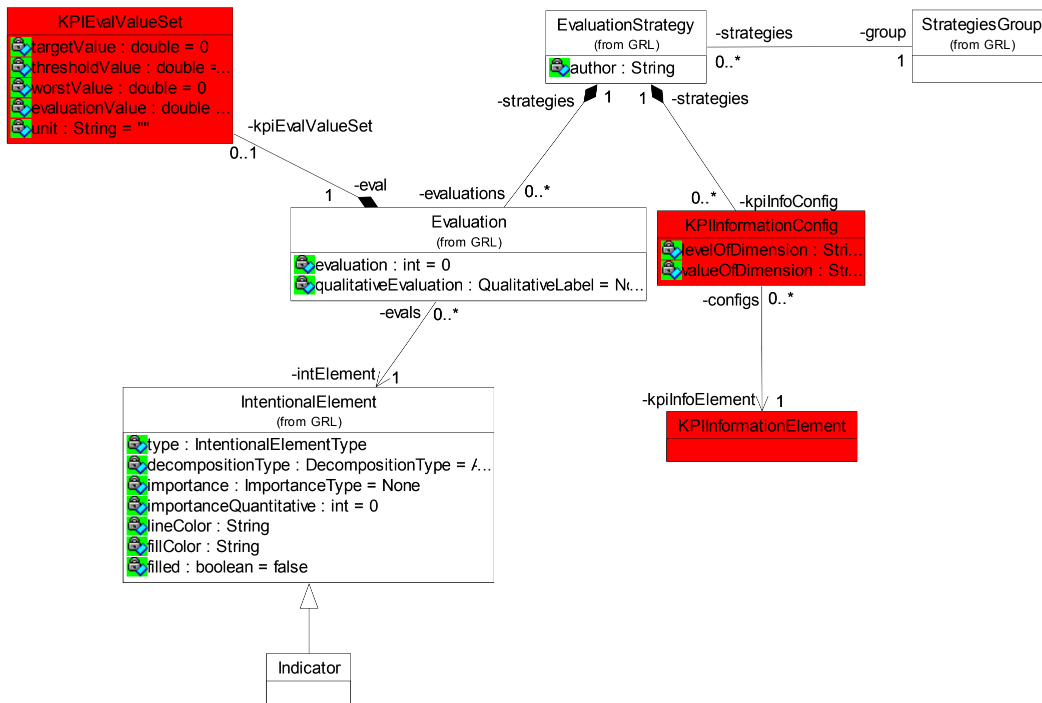

Figure 15 KPIStrategy in jUCMNav's Metamodel

4.1.2 Classes Defined but Unused in jUCMNav's Metamodel

Some classes are defined in jUCMNav metamodel, but actually never used. Here are the classes: ContributionChange, ContributionContext, and ContributionContextGroup. We will not import/export these classes. Consequently, the relationships from and to these classes are ignored as well. These ignored classes are highlighted in red in Figure 16.

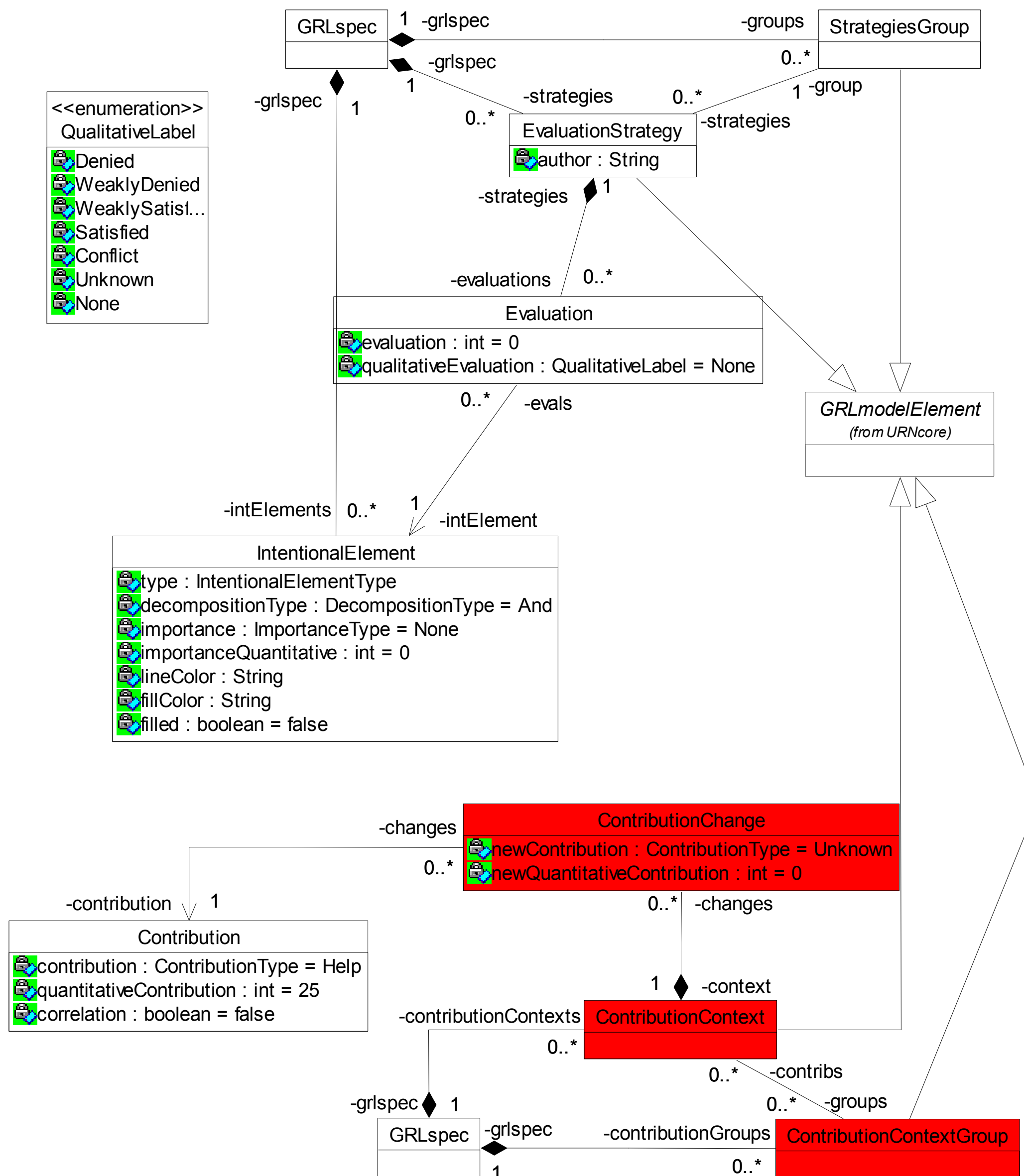

Figure 16 GRL Strategy in jUCMNav's Metamodel

4.1.3 Classes Defined in jUCMNav Metamodel, but not in Z.151

Class **FailurePoint** and **Anything** are subclasses of PathNode (see Figure 17). They are defined in jUCMNav, but they are not in Z.151. Class FailurePointMHandler and AnythingMHandler are the handler classes for export, and EmptyPointUMHandler are the handle class for import. Instances of these two classes are exported as instances of Emp-

tyPoint with a metadata. For FailurePoint objects, the name for metadata is "jUCMNav FailurePoint expression" and value for metadata is the value of the attribute expression. For Anything objects, the name of metadata is "jUCMNav Anything" and the value of metadata is an empty string. The import works by reconstructing FailurePoint objects and Anything objects from information in metadata.

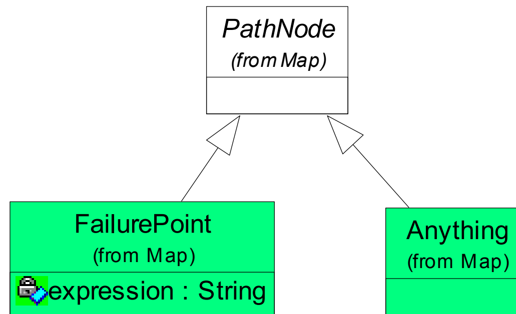

Figure 17 Class FailurePoint and Anything in jUCMNav Metamodel

4.1.4 Interface Defined in jUCMNav's Metamodel

Interfaces are used to organize code in a way such that all implementations of the interface have to provide attributes and implement functions defined in interface. jUCMNav's metamodel has several interfaces (see Figure 18), but Z.151 as a URN standard does not provide any interface for implementation purpose. However, Z.151 keeps the equivalent attributes and associations inside classes. All information will be preserved during export/import. Figure 18 shows all interfaces defined in jUCMNav's metamodel.

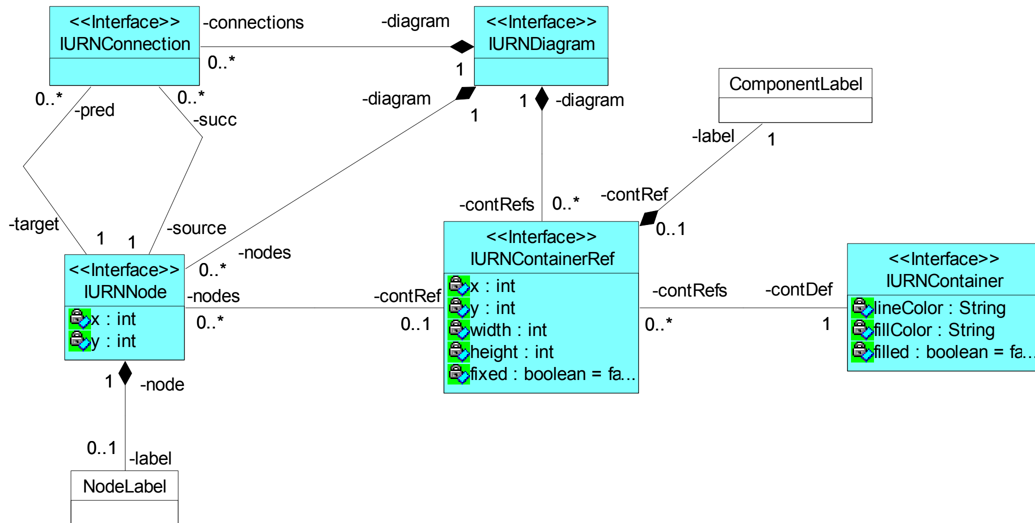

Figure 18 URNcore/URNabstract in jUCMNav Metamodel

Table 1 lists all interfaces in jUCMNav and all implementation classes:

Table 1 Interfaces and Classes.

Interface	Implementation Class
IURNConnection	BeliefLink, LinkRef, NodeConnection
IURNContainer	Actor, Component
IURNContainerRef	ActorRef, ComponentRef
IURNDiagram	GRLGraph, UCMap
IURNNode	GRLNode, PathNode

4.1.5 Attributes Defined in jUCMNav's Metamodel, but not in Z.151

Some attributes and associations are defined in jUCMNav's metamodel, but they are not defined in Z.151's metamodel. In the case of deprecated attributes, we will ignore them during import/export. If they are not deprecated, we export these data as metadata and import these data back as attributes/associations for jUCMNav objects to preserve as much data as we can. Table 2 lists attributes/associations defined in jUCMNav's metamodel but not in Z.151. We also list the way we handle them. The name of the metadata is created in this way: keyword *jUCMNav* followed by the class name and then the attrib-

ute's name or the role of the association. For example: "jUCMNav Actor includedActors". The value of the metadata will be the value of the attribute.

Table 2 Attributes in jUCMNav's Metamodel but not in Z.151.

Class	Attributes	Associations	Handling
Actor		Bi-direction self association: 0..1 includingActor and 0..* includedActors	Metadata
ActorRef		Bi-direction self association: 0..1 parent and 0..* children.	Metadata
AndFork, AndJoin, OrFork, OrJoin	orientation (Deprecated)		Ignored
Blief	author		Metadata
Component	Slot (Deprecated)		Ignored
ComponentRef	ReplicationFactor, role, anchored, fixed		Metadata
EndPoint	Local		Metadata
EvaluationStrategy	author		Metadata
Responsibility	context, empty		Metadata
StartPoint	local, failureKind		Metadata
Stub	Aopointcut, aspect, repetitionCount, shared, PointCut		Metadata

4.1.6 ComponentKind has Other in jUCMNav, but not in Z.151

jUCMNav has Component objects with ComponentKind Other, but Z151 does not have this kind. Since jUCMNav sets component to have TEAM as default ComponentKind,

we export the Component objects with ComponentKind Other in jUCMNav to Component object with ComponentKind Team in Z.151.

4.1.7 IntentionalElementType has INDICATOR in jUCMNav, but not in Z.151

jUCMNav has INDICATOR as IntentionalElementType for Indicators, but Z151 does not have it. We export this information to metadata to indicate the IntentionalElement object is actually an Indicator object and set IntentionalElementType to TASK. We import this information by reconstructing the Indicator object. Note that we drop all the related KPI links and objects.

4.1.8 DeviceKind has Other in jUCMNav, but not in Z.151

jUCMNav has DeviceKind Other, but Z151 does not have this kind. Since jUCMNav sets DeviceKind to have PROCESSOR as default, we export and import the value PROCESSOR instead.

4.1.9 ScenarioStartPoint and ScenarioEndPoint in jUCMNav, but not in Z.151

Import is handled by ScenarioDefUMHanlder, StartPointUMHandler, and EndPointUMHandler, and export is handled by ScenarioDefMHandler, ScenarioStartPointMHandler, ScenarioEndPointMHandler, StartPointMHandler and EndPointMHandler.

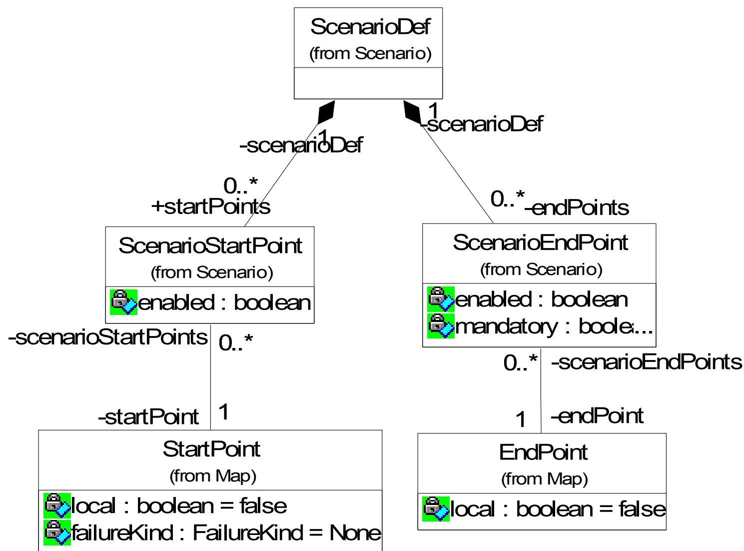

Figure 19 ScenarioStartPoint and ScenarioEndPoint in jUCMNav's Metamodel

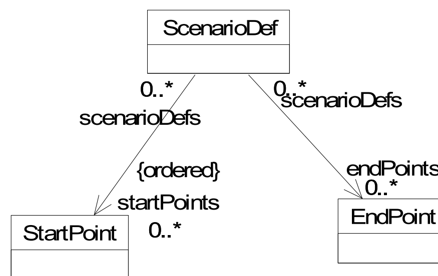

Figure 20 ScenarioStartPoint and ScenarioEndPoint not in Z.151's Metamodel

4.2. In Z.151's Metamodel but not in jUCMNav's Metamodel

4.2.1 ID in Z.151, but not in jUCMNav

In Z.151's metamodel, the following classes are required to have a universal unique identifier ID: InBinding, LinkRef, NodeConnection, OutBinding, PluginBinding, and URNlink. The attribute IDs for these Z.151 classes are required for successfully marshalling/unmarshalling between XML files and Z.151 objects. However, the corresponding classes in jUCMNav's metamodel do not have an attribute ID: InBinding, LinkRef, NodeConnection, OutBinding, PluginBinding, URNlink, and grl.BeliefLink.

We need to make up fake IDs for Z.151 objects created by the corresponding handler classes in package marshal for export. The generation of the fake IDs must satisfy the condition: the generated fake ID for the same jUCMNav input object must be always the same at run time. The reason is we use the same ID to store and fetch the same Z.151 output object from the HashMap `id2object` at run time.

This is the code used to generate IDs in class MHandler:

```
public String getObjectId(Object obj) {
        return "Z151_id_" + obj.getClass().getName() + "_"
                           + obj.hashCode();
}
```

4.2.2 GRLnode Has a Size (Width/Heigth) in Z.151, but not in jUCMNav

In class GRLnodeMHandler, we have assigned GRLnode objects a size with width and height set to zero. Other tools importing this information will hence need to infer appropriate sizes for GRL elements.

4.2.3 Z.151 Supports Visualization of XOR/IOR as Means-end

We do not import/export this information. All XOR/IOR are displayed simply with an XOR/IOR decomposition graphical syntax, no means-end symbol (Z.151, section 7.6.1) is used.

4.2.4 Workload Decomposed in Z.151, but not in jUCMNav

jUCMNav uses only one class to describe performance workloads associated with UCM start points (Figure 20) whereas Z.151 (section 8.6.1) defines a hierarchy of classes (Figure 21). During the export, class WorkloadMHandler creates Z.151 objects of the appropriate sub-class of class Workload decided by the values of attributes `closed` and `arrivalPattern` of the input jUCMNav workload objects. During the import, class WorkloadUMHandler creates jUCMNav workload objects and sets attributes accordingly with respect to the class of the input jUCMNav workload objects

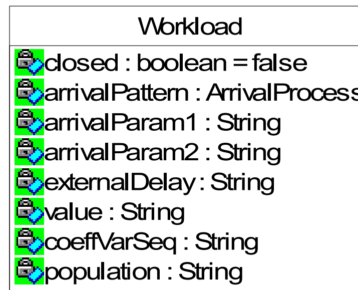

Figure 21 Workload in jUCMNav's Metamodel

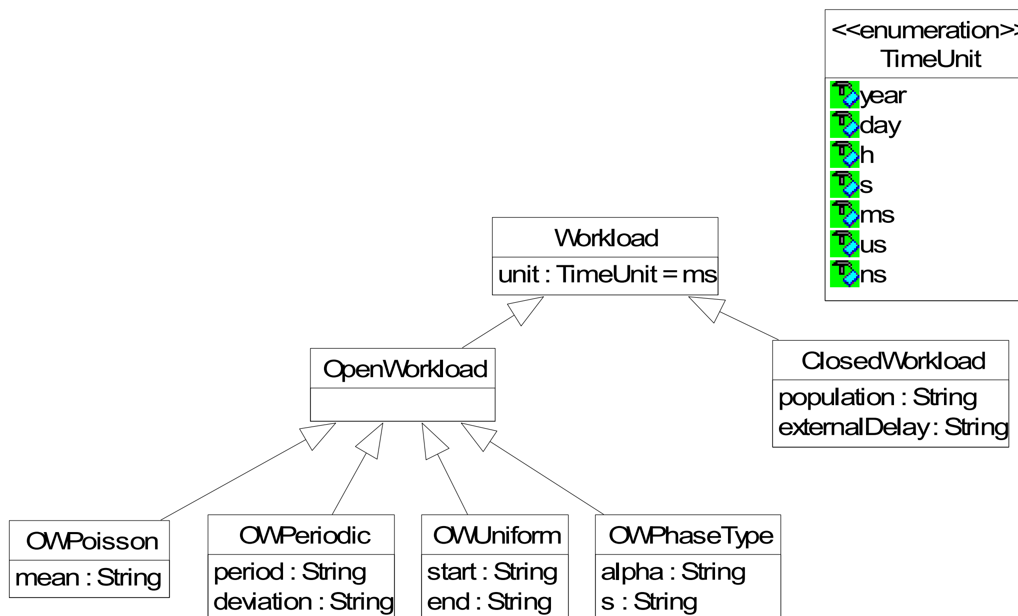

Figure 22 Workload in Z.151's Metamodel

4.2.5 Z.151 supports Movable GRL LinkRef Labels, but not jUCMNav

Labels associated with GRL link references have a position computed automatically by jUCMNav, whereas such labels can be moved around by modellers in Z.151 (section 7.6.8). When importing, this position information is dropped.

4.3. In Both Metamodels, but Unused in jUCMNav

4.3.1 Component Types

The values of UCM ComponentType (Z.151, section 8.2.1) are preserved during import and export even though they are not used in jUCMNav's GUI. This is handled in class ComponentMHandler and ComponentUMHandler.

4.3.2 Collapsed Actor References

Instances of GRL CollapsedActorRef (Z.151, section 7.6.6) are preserved during import and export even though they are not used in jUCMNav's GUI. This is handled in class CollapsedActorRefMHandler and CollapsedActorRef UMHandler.

4.4. Mismatches between the Two Metamodels

4.4.1 Differences between Belief and Belief Link

A Belif object in jUCMNav metamodel is equivalent to an instance of IntentionalElementRef and an instance of to IntentionalElement with IntentionalElementType Belief. However, nothing in jUCMNav metamodel is equivalent to IntentionalElement with IntentionalElementType Belief. When we export a Belif, we create a Z.151 IntentionalElement object with IntentionalElementType Belief and a Z.151 IntentionalElementRef object. Import ignores the Z.151 IntentionalElement object but covert the Z.151 IntentionalElementRef object to a Belief object

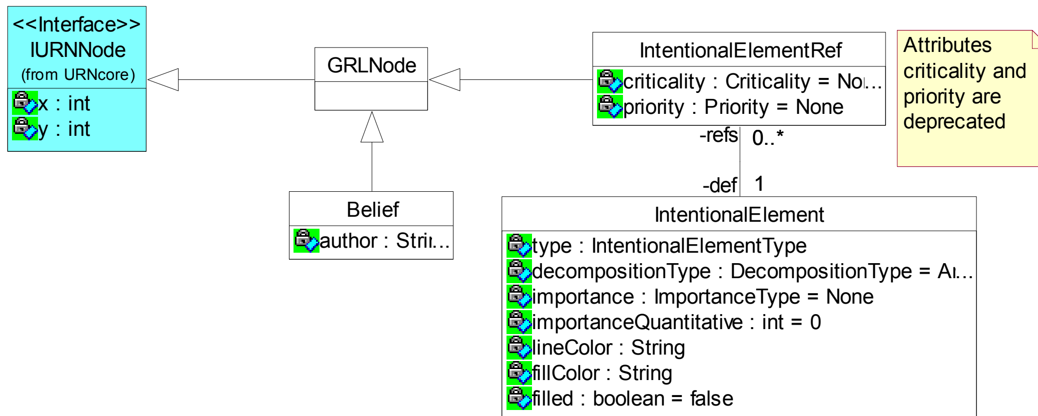

Figure 23 jUCMNav Belief

Belief links are instance of class BeliefLink in jUCMNav's metamodel. They are instances of LinkRef with type IntentionalElementType.BELIEF in Z.151's. When we export a BelifLink, we create a Z.151 LinkRef with type IntentionalElementType.BELIEF and a Z.151 ElementLink object. Import will ignore the Z.151 ElementLink object but covert the Z.151 LinkRef object to a BeliefLink objct.

4.4.2 Concern

Concern in (Z.151, section 6.1.5) targets URNmodelElement, whereas Concern targets only IURNDiagram in jUCMNav. Since GRLgraph and UCMmap implements interface IURNDiagram, we currently only import and export Concern objects associated with GRLgraph objects or UCMmap objects and ignore other Concern objects associated with objects with type of other URNmodelElement. However, a last-minute modification to jUCMNav's metamodel added another relationship between Concern and URNmodelElement, and hence this difference could be resolved in the future.

4.4.3 DecompositionType

DecompositionType in the metamodels is slightly different. Z.151 (section 7.4.6) has {AND, XOR, IOR}, whereas jUCMNav has {And, Or, Xor}. Since Or is equivalent to IOR, we export DecompositionType with Or in jUCMNav to be IOR in Z.151 and import DecompositionType with IOR in Z.151 to be Or in jUCMNav.

4.4.4 Association between EvaluationStrategy and StrategiesGroup

EvaluationStrategy in Z.151 may be associated to at least one StrategiesGroup in Z.151 (section 7.5.2), but to only one in jUCMNav. We export one StrategiesGroup associated with EvaluationStrategy and we only import the first StrategiesGroup associated with EvaluationStrategy. We notify users with a warning when we ignore StrategiesGroups.

4.4.5 Role includingComponent for Component Self-association

There are different multiplicities involved here: 0..1 Component can be includingComponent in jUCMNav, but we find 0..* includingComponent in Z.151 (section 8.4.1). We export at most one includingComponent associated with Component because at most one Component can be includingComponent in jUCMNav and we only import the first associated includingComponent from Z.151. We notify users with a warning when we ignore includingComponents.

4.4.6 Association between ScenarioDef and StrategiesGroup

ScenarioDef in Z.151 may be associated to at least one StrategiesGroup in Z.151 (section 8.5.2), but to only one in jUCMNav. We export one ScenarioDef associated with EvaluationStrategy and we only import the first StrategiesGroup associated with ScenarioDef. We notify users with a warning when we ignore StrategiesGroups.

4.4.7 URN Data Model

There are differences between the data languages used to describe conditions. For instance, Z.151 (section 9) supports special prefixes for references to original variable values in postconditions, and jUCMNav supports the keyword “else” (which is the logical complement of the other conditions involved in an OR-Fork or a Stub). These differences are currently not handled, and logical expressions are exported/imported as is.

Chapter 5. Testing

The import and export of URN models in Z.151 XML file format with jUCMNav is actually the process of transforming information stored in Java Z.151 objects into Java jUCMNav objects, and vice versa. Testing these features is a validation of the equivalence between source objects and target objects.

5.1. Test Items

We need to test the following four categories of items:

1. URN elements exist in both jUCMNav's and Z.151's metamodels
2. URN elements exist in jUCMNav's but not in Z.151's metamodels
3. URN elements do not exist in jUCMNav's but do in Z.151's metamodels
4. URN elements mismatch in jUCMNav's and Z.151's metamodels

URN models created by jUCMNav plug-in are used as inputs to test this feature. Where this is not sufficient, we manually create Z.151 XML files containing URN elements that jUCMNav is unable to create.

We test the implementation with explicitly created URN models (which are as simple and targeted as possible), and then with existing complex jUCMNav models created by Dainel Amyot's team. Many existing models contain more than 100 diagrams, whereas others were created explicitly as new test cases for specific URN constructs. Test cases with explicitly created models are important for us to pinpoint problems, whereas complex jUCMNav models improve test coverage. Together, these test cases cover all the classes, attributes, and associations found in jUCMNav's metamodel that are accessible to modellers through the tool GUI.

The following models were used to test the import and import features:

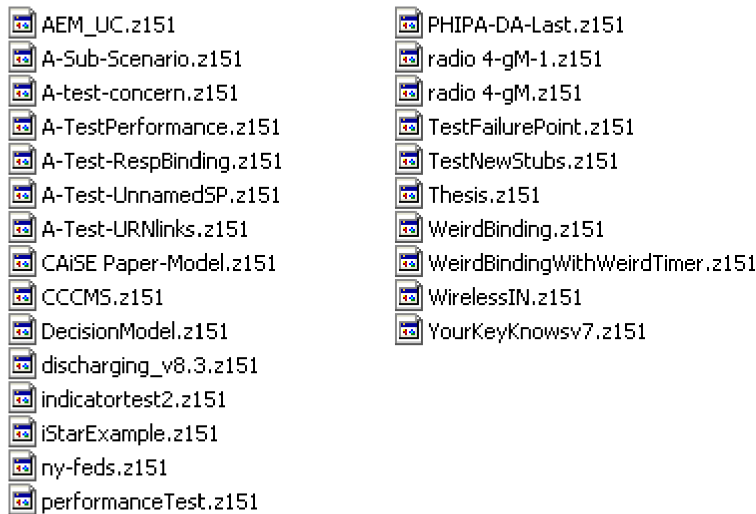

Figure 24 Test Models

5.2. Approach

We used the following approach: Take a jUCMNav URN model, export it to a Z.151 XML file, and then import the exported Z.151 XML file back. We compared the imported jUCMNav URN model with the original jUCMNav URN model using Eclipse's comparison tool. Then we export each imported jUCMNav URN model to another Z.151 XML file, and compare these two Z.151 XML files. Comparing two jUCMNav files helps identify URN model elements that have gone missing during import/export. Comparing two Z.151 files does the same thing, but we write JUnit test case for regression.

5.3. Test Cases

Class `Z151importexportTest` is a JUnit class to test this feature. Directory `'expect'` has all expected Z.151 XML files and directory `'actual'` holds actual Z.151 XML files generated by JUnit test cases (see Figure 25).

Class `Z151importexportTest` has two mock methods: `mockImport` and `mockExport`. These two methods are very similar to the implemented import and export methods, but for testing purpose. The method `compareTwoZ151File` implements the main testing process. It takes a string (a path) for the expected Z.151 XML file, imports the expected file to a `urn.URNspec` object and exports the `urn.URNspec` object to an-

other Z.151 XML file (actual), and then compare contents of two Z.151 XML files as two strings.

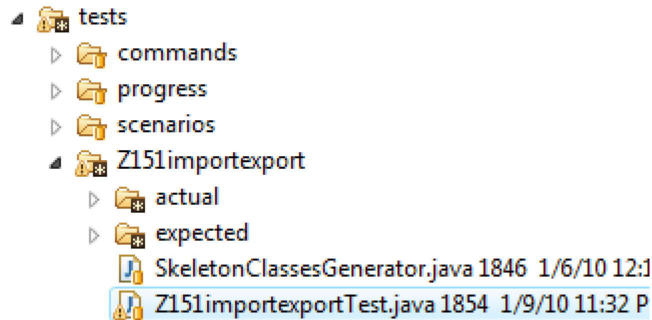

Figure 25 Z151importexport Test Directory

It is simple to create a test case in class `Z151importexportTest`: Initializing a string with the path to the expected Z.151 XML file and then call method compareTwoZ151File. The below is an example for a test case.

```
public void testActor(){
      String expected = "actor.z151";
      compareTwoZ151File(expected);
}
```

Please note that we need to run test cases in this class as JUnit Plug-in Test: Right click on the file in eclipse, select 'Run As > JUnit Plug-In Test'. Since some input jUCMNav files are designed for customers, we should not publish test cases based on these jUCMNav files in jUCMNav project's source code for confidentiality reasons.

Chapter 6. Conclusions

This report is about the implementation of Eclipse plug-ins for importing/exporting URN models in Z.151 XML format with jUCMNav.

6.1. Contributions

This project led to the following contributions

- We have implemented a new functionality for the jUCMNav application: the import/export of URN models in the standard Z.151 format.
- We have supported a bi-directional transformation between the standard Z.151 metamodel (November 2008) and the jUCMNav metamodel version 0.23 (December 21, 2009).
- We have tested the import and export with both a set of simple URN models and a comprehensive set of existing URN models provided by Prof. Amyot's team.
- We have found six errors in the Z.151 XML schema, which will be sent to ITU-T as a contribution to improve the Z.151 standard.

With the above contributions, jUCMNav becomes the first tool to support the standard Z.151 file format.

6.2. Future work

jUCMNav's metamodel keeps evolving possibly a few times per year and the metamodel defined in URN standard will evolve possibly once per two years. Maintenance of the code will be required whenever one of these metamodels evolves.

- Upon receiving changes in the standard Z.151 metamodel, we need to recompile the most updated Z.151 schema and save the generated Java files to the directory seg.jUCMNav.importexport.z151.generated package, and rebuild the

jUCMNav project. Then, we need to update the handler classes to be aligned with the updates in corresponding classes.

- If a new class is introduced into the standard metamodel or jUCMNav's metamodel and if we decide to support this class, we need to create two handler classes: one for MHandler and one for UMHandler. We also need to add an entry with a pair composed of the class and the handler class into HashMap ourClass2Conv in class MHandler and EObjectImplUMHandler. If the new class is a subclass of an existing class, only the attributes defined in this class should be handled in the new MHandler and UMHandler.
- Make sure to test the newly introduced changes thoroughly.

In addition, remaining compatibilities issues and limitations identified in Chapter 4 should be addressed. In particular, two issues were stored in jUCMNav's Bugzilla database: improvements to the handling of Concern associations (bug 750), and the handling of differences between the jUCMNav and Z.151 data models (bug 752).

References

[1] Allilaire, F. And Jouault, F. (2007) "Families to Persons" - A simple illustration of model-to-model transformation. http://www.eclipse.org/m2m/atl/doc/ATLUseCase_Families2Persons.ppt Accessed December 2009.

[2] Chen, P. (2007) Goal-Oriented Business Process Monitoring: An Approach based on User Requirement Notation combined with Business Intelligence and Web Services. MSc thesis, Carleton University, Dec. 2007. http://jucmnav.softwareengineering.ca/ucm/pub/UCM/VirLibPChenThesis07/Pengfei_MCS_Thesis.pdf Accessed December 2009.

[3] Eclipse, M2M/Atlas Transformation Language (ATL). http://wiki.eclipse.org/M2M/Atlas_Transformation_Language_%28ATL%29 Accessed December 2009.

[4] ITU-T – International Telecommunications Union: *Recommendation Z.151 (11/08), User Requirements Notation (URN)*. Geneva, Switzerland, November 2008.

[5] Jaxb: JAXB Reference Implementation. https://jaxb.dev.java.net/ Accessed December 2009.

[6] Jouault, F, and Bézivin, J (2006). In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037, Bologna, Italy, pages 171-185.

[7] jUCMNav 4.1, University of Ottawa, November 2009. http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome. Accessed December 2009.

[8] jUCMNav metamodel, http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/URNMetaModel . Accessed December 2009.

[9] Krause, C.(2009). Model Transformation in Eclipse with ATL and EMF Henshin, http://homepages.cwi.nl/~koehler/talks/model-transformation.pdf Accessed January 2009

[10] Laun, W. (2009) A JAXB Tutorial, https://jaxb.dev.java.net/tutorial/ Accessed December 2009.

[11] OMG, MOF 2.0 / XMI Mapping Specification, v2.1.1. http://www.omg.org/technology/documents/formal/xmi.htm Accessed December 2009.

[12] Sun Microsystems, Java Architecture for XML Binding - Binding Compiler (xjc). http://java.sun.com/webservices/docs/1.6/jaxb/xjc.html Accessed December 2009.

Appendix A: Sample Generated Skeleton Class

The code below is a sample skeleton class for EnumerationTypeUMHandler:

```
package seg.jUCMNav.importexport.z151.unmarshal;

// <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
// <!--  EnumerationType  -->
// <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
// <xsd:complexType name="EnumerationType">
//    <xsd:complexContent>
//       <xsd:extension base="UCMmodelElement">
//          <xsd:sequence>
//             <xsd:element name="values" type="xsd:string" />
//             <xsd:element maxOccurs="unbounded" minOccurs="0"
//                name="instances" type="xsd:IDREF" /> <!-- Variable -->
//          </xsd:sequence>
//       </xsd:extension>
//    </xsd:complexContent>
//  </xsd:complexType>

import org.eclipse.emf.common.util.EList;
import seg.jUCMNav.importexport.z151.generated.EnumerationType;
import seg.jUCMNav.model.ModelCreationFactory;

public class EnumerationTypeUMHandler extends UCMmodelElementImplUMHandler {
   public Object handle(Object o, Object target, boolean isFullConstruction) {
      EnumerationType elemZ = (EnumerationType) o;
      String objId = elemZ.getId();
      ucm.scenario.EnumerationType elem = (ucm.scenario.EnumerationType) getOb-
ject(objId, target, ucm.scenario.EnumerationType.class);
      if (isFullConstruction) {
         elem = (ucm.scenario.EnumerationType) super.handle(elemZ, elem, is-
FullConstruction);
         elem.setUcmspec();
         elem.setValues();
         elem.setId();
         elem.setInconcern();
         elem.setName();
         elem.setDescription();

         elem.getUcmspec();
         elem.getInstances();
         elem.getValues();
         elem.getFromLinks();
         elem.getToLinks();
         elem.getMetadata();
         elem.getInconcern();
         elem.getName();
         elem.getId();
         elem.getDescription();
         elem.getClass();
      }
      return elem;
   }
}
```