Reuse-Oriented Modelling and Transformation Language Definition

Deliverable D3.2.1, version 1.0, 11.06.2007

IST-2006-033596
ReDSeeDS
Requirements Driven
Software Development System
www.redseeds.eu

Infovide-Matrix S.A., Poland
Warsaw University of Technology, Poland
Hamburger Informatik Technologie Center e.V., Germany
University of Koblenz-Landau, Germany
University of Latvia, Latvia
Vienna University of Technology, Austria
Fraunhofer IESE, Germany
Algoritmu sistemos, UAB, Lithuania
Cybersoft IT Ltd., Turkey
PRO DV Software AG, Germany
Heriot-Watt University, United Kingdom

Reuse-Oriented Modelling and Transformation Language Definition

Workpackage	WP3
Task	T3.2
Document number	D3.2.1
Document type	Deliverable
Title	Reuse-Oriented Modelling and Transformation Language Definition
Subtitle	
Author(s)	Audris Kalnins, Agris Sostaks, Edgars Celms, Elina Kalnina, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Volker Riediger, Hannes Schwarz, Daniel Bildhauer, Sevan Kavaldjian, Roman Popp, Jürgen Falb
Internal Reviewer(s)	Michał Śmiałek, Hermann Kaindl, Davor Svetinovic, John Paul Brogan, Thorsten Krebs
Internal Acceptance	Project Board
Location	https://svn.redseeds.eu/svn/redseeds/1_DeliverablesSpace/WP3_Reusable-case_specification_language/D3.2.01/ReDSeeDS_D3.2_Reuse-Oriented_Modelling_and_Transformation_Language_Definition.pdf
Version	1.0
Status	Final
Distribution	Public

11.06.2007

History of changes

Date	Ver.	Author(s)	Change description
19.04.2007	0.01	Audris Kalnins	Proposition of ToC
02.05.2007	0.02	Audris Kalnins	Added section 5.1
07.05.2007	0.03	The WUT team	Added initial content for sections 6.1 and 6.2
07.05.2007	0.04	Daniel Bildhauer	Added chapter 3
09.05.2007	0.05	Agris Sostaks	Added initial content for section 5.2
21.05.2007	0.06	Daniel Bildhauer	Updates to chapter 3
21.05.2007	0.07	Daniel Bildhauer	Added section 2.1 and 2.3
24.05.2007	0.08	Albert Ambroziewicz	Updated section 6.2
25.05.2007	0.09	Albert Ambroziewicz, Wiktor Nowakowski	Added content for section 6.3
25.05.2007	0.10	Hannes Schwarz	Added chapter 8
29.05.2007	0.11	Albert Ambroziewicz	Updated sections 6.1, 6.2.3 and 6.2.4
29.05.2007	0.11	Tomasz Straszak	Added content for section 4.2.2
30.05.2007	0.12	The UL team	Finalized chapter 5
31.05.2007	0.13	Volker Riediger	Finalized chapter 7
01.06.2007	0.14	The TUW team	Added refined UI transformation example
04.06.2007	0.15	Audris Kalnins	Added content for 4.1, 4.2, 4.2.1
04.06.2007	0.16	Jacek Bojarski, Tomasz Straszak	Added content for 4.2.3
05.06.2007	0.17	The UL team	Added initial content for section 6.5
06.06.2007	0.18	The UL team	Finalized section 6.5
07.06.2007	0.19	Audris Kalnins	Added executive summary
08.06.2007	0.20	Hermann Kaindl, Audris Kalnins, Hannes Schwarz, Daniel Bildhauer	Minor fixes
08.06.2007	0.21	Daniel Bildhauer	Final build
08.06.2007	1.00	John Paul Brogan HWU	Document correction and final build

Summary

The ReDSeeDS project is primarily about reusing software cases. Before being reused, however, these software cases must be created. The artefacts constituting such a software case — requirements, models, code — are to be specified using the *software case language* (SCL). SCL is actually a union of several languages. One component of SCL — the Requirements Specification Language (RSL) — is already defined in previous deliverables. In this deliverable, we define the other parts of SCL for specifying software cases in the ReDSeeDS approach.

For creating software cases, we focus on *model-driven* software development. This approach requires one or more models to be built in a modelling language before code development can be started. In ReDSeeDS, two such models should be built as artefacts within a software case — an *architecture* model and a *detailed design* model. Hence, an appropriate *modelling* language must be part of SCL. There are some reuse-related issues common for all parts of SCL. Firstly, reuse must be supported by *traceability* links between all artefacts constituting one software case. Secondly, as far as possible, automatic *transformations* should be provided for generating an initial version of the next artefact in the chain from the previous one (which is then completed manually). Therefore, a model transformation language for defining these transformations is needed in SCL as well. All these issues are discussed in this document to a certain degree, but the main emphasis is on two of them — the definition of an appropriate *modelling language* for specifying architecture and detailed design models, and the definition of an appropriate model *transformation language*.

In contrast to RSL, which is an original development within the ReDSeeDS project, there has been no intention to develop a completely new modelling language for specifying architecture and detailed design models. Instead, we developed principles for how to select an appropriate subset of an existing language, together with some usage guidelines. Due to the wide use of the Universal Modelling Language (UML) in software design practice, especially the use of UML for SCL is discussed in detail. The main issue with UML is that it is too large and too universal to be efficiently used by software case developers without guidelines. Therefore, we propose appropriate subsets for specifying architecture and design models. However, another approach to building these models and another subsets of UML could be used as well within the general framework of SCL.

The situation with model transformation languages is different. Though there is a growing interest in such languages, no widely accepted candidate exists. The efforts by the OMG to propose a standard in this area have not succeeded yet. Therefore, we decided to use the transformation language *MOLA*, being developed by one of the ReDSeeDS partners — University of Latvia IMCS. This document contains the complete description of this language.

The possibility of applying transformations for automatic generation of models for software cases is discussed and illustrated through examples, especially for arriving at an architecture model from requirements. We sketch related transformation rules, some of which are also formalized as MOLA transformation examples. As a special innovation, the possibility of applying transformations is sketched for generating a user interface as well.

In addition, we propose a consistent approach to traceability among all parts of SCL. The technical evolution of RSL (how to make its metamodel tool-ready), and how a programming language can smoothly be incorporated into SCL are discussed as well. The latter issue is dealt with on the basis of Java 5, by analyzing its metamodel.

Overall, we propose a comprehensive solution for the complete SCL for specifying software cases to be reused.

Table of contents

History of changes III

Summary V

Table of contents VII

List of figures X

1 Scope, conventions and guidelines 1

1.1 Document scope . . . 1
1.2 Conventions . . . 2
1.3 Related work and relations to other documents . . . 2
1.4 Structure of this document . . . 3
1.5 Usage guidelines . . . 3

2 Introduction 5

2.1 General principles of SCL . . . 5
2.2 Components of SCL . . . 7
2.3 SCL and the ReDSeeDS software case repository . . . 8

3 Adaptation of RSL to SCL 10

3.1 General small changes . . . 10
3.1.1 Specialisation of SCLElement and SCLRelationship . . . 11
3.1.2 Role names . . . 11
3.1.3 Relationships . . . 12
3.1.4 Abstract classes . . . 12
3.1.5 Multiplicities . . . 13
3.1.6 Navigable associations . . . 13
3.1.7 Redefinitions . . . 13
3.1.8 Compositions and aggregations . . . 14
3.2 Changes to single packages . . . 14
3.2.1 Attributes . . . 14
3.2.2 DescriptiveRequirementRepresentation . . . 14

3.2.3 NaturalLanguageRepresentation 16
3.2.4 ConstrainedLanguageSentence 16
3.2.5 ActivityRepresentation 18
3.2.6 InteractionRepresentation 18
3.2.7 Phrases 20
3.2.8 Terms 21
3.2.9 DomainElementRepresentation 27
3.2.10 Notions 28
3.3 Conclusion 28

4 Software Development Specification Language **30**
4.1 General principles of SDSL 30
4.2 SDSL based on UML 31
4.2.1 Principles of UML usage for SDSL 31
4.2.2 UML facilities for Architecture model 33
4.2.3 UML facilities for Detailed Design model 41
4.3 Other possible notations for SDSL 45
4.3.1 Decision tables as example 45
4.3.2 Decision-table metamodel 46
4.3.3 Approach for other languages 46

5 Transformation Language MOLA **48**
5.1 MOLA overview 48
5.1.1 Basic elements of MOLA 48
5.1.2 Simple MOLA example 51
5.2 MOLA reference 58
5.2.1 Structure of the MOLA reference 58
5.2.2 Naming conventions 60
5.2.3 Kernel package - metamodel definition facilities in MOLA 60
5.2.4 MOLA package - transformation definition facilities in MOLA 75
5.2.5 Expressions 118
5.2.6 Additional remarks on syntax and semantics 126

6 Possibilities of automatic transformations between models **132**
6.1 Role of automatic transformations 132
6.2 Transformation rules for precise requirements model defined in RSL 133
6.2.1 Generating multi-layer architecture from requirements structure 133
6.2.2 Generating architectural details 134
6.2.3 Issue of traceability between generated model and model edited by an architect 136

6.2.4 Other issues concerning automatic architectural model generation . . . 137
6.3 Examples of possible model transformations in SCL 142
6.4 UI model-to-model transformation example 151
6.4.1 Conceptual GUI meta-model 151
6.4.2 GUI transformation rules 152
6.4.3 Transformation example 154
6.5 Example of model transformation in MOLA 159
6.5.1 Metamodel definitions in MOLA 161
6.5.2 Transformation procedures in MOLA 170

7 Transition to code in SCL 196
7.1 A metamodel for the Java 5 programming language 198
7.1.1 Overview over the Java 5 metamodel 199
7.1.2 Representaion of Java 5 programs 201
7.2 Rules for the integration of source code models into SCL 204

8 Traceability facilities in SCL 206
8.1 Traceability as enabler of reuse 206
8.2 Integration of traceability links in SCL 207
8.2.1 RSL traceability links 208
8.2.2 UML traceability links 212
8.2.3 Inter-language traceability links 213
8.3 Possible visualisation of traceability links 214

Bibliography 216

List of figures

2.1 SCL and software case structure 6
2.2 Relation of RSL, UML, Java and SCL 7

3.1 Changes to attributes package 15
3.2 Changes to descriptive requirement representation 16
3.3 Changes to natural language sentence 17
3.4 Changes to constrained language sentence 17
3.5 Changes to activity representation 19
3.6 Changes to predicate message in interaction representation 21
3.7 Changes to messages in interaction representation 22
3.8 Changes to interaction lifelines 23
3.9 New rejoin message in interaction representation 23
3.10 Noun phrases package 24
3.11 Verb phrases package 24
3.12 Changes to terminologie 25
3.13 Changes to terms inflections 26
3.14 Changes to terms synonyms 27
3.15 Changes to representation of domain elements 27
3.16 Changes to notions 28

4.1 4-layer architecture model example 38
4.2 Example of UML packages for components 39
4.3 Example of UML Packages for interfaces 39
4.4 Example of sequence diagram for 4-layer architecture 40
4.5 Class diagram - example of detailed design 43
4.6 Class diagram - example of detailed design 43
4.7 Sequence diagram - example of detailed design 45
4.8 Example for a decision table 46
4.9 Example metamodel for a decision table 47
4.10 Package structure of SCL extended by decision tables 47

5.1 The metamodel of the example 53
5.2 The MOLA procedure Main 54

5.3 The subprocedure ProcessAttribute 57
5.4 Kernel package 61
5.5 Comment and enumeration notation 62
5.6 Class and attribute notation 70
5.7 Generalization notation 71
5.8 Association notation 75
5.9 MOLA package 76
5.10 Procedures and packages in MOLA 77
5.11 Control flows in MOLA 83
5.12 Control flow notation 87
5.13 Statements in MOLA 88
5.14 Start symbol notation 89
5.15 End symbol notation 90
5.16 Call statement notation a) to a MOLA procedure; b) to an external procedure . . 91
5.17 Rule notation 95
5.18 Text statement notation 97
5.19 Fixed foreach loop notation 100
5.20 Not-fixed foreach loop notation 100
5.21 While-loop notation 102
5.22 Referencable elements 103
5.23 Parameter notation a) in b) in-out 106
5.24 Variable notation 107
5.25 Pattern definition elements 109
5.26 Class element notation a) normal b) create c) delete d) loop variable 112
5.27 Association link notation a) normal b) create c) delete 116
5.28 Assignments in MOLA 117
5.29 The metamodel used in examples 127
5.30 Pattern example 127
5.31 Pattern example 128
5.32 Pattern example 128
5.33 Pattern example 128
5.34 Pattern example 129
5.35 Pattern example 129
5.36 Pattern example 129

6.1 4-layer architecture generation overview 138
6.2 4-layer architecture generation - interfaces 139
6.3 4-layer architecture generation - sequence diagrams 140
6.4 4-layer architecture generation - methods of interfaces 141

6.5 4-layer architecture model example - actors 142
6.6 4-layer architecture model example - Presentation layer components with interfaces 143
6.7 4-layer architecture model example - Application Logic layer components with interfaces 143
6.8 4-layer architecture model example - Business Logic layer components with interfaces 143
6.9 4-layer architecture model example - Data Access layer components with interfaces 144
6.10 4-layer architecture model example - sequence diagram for basic path scenario of Enter Facility Use Case 145
6.11 4-layer architecture model example - sequence diagram for 1st alternate path scenario of Enter Facility Use Case 146
6.12 4-layer architecture model example - sequence diagram for 2nd alternate path scenario of Enter Facility Use Case 147
6.13 4-layer architecture model example - sequence diagram for basic path scenario of Browse An Offer And Reserve Use Case 148
6.14 4-layer architecture model example - sequence diagram for 1st alternate path scenario of Browse An Offer And Reserve Use Case 149
6.15 4-layer architecture model example - sequence diagram for 2nd alternate path scenario of Browse An Offer And Reserve Use Case 150
6.16 A conceptual GUI meta-model 152
6.17 Constrained language representation of scenario “Making facility reservation” . 155
6.18 UIBehaviourRepresentations::UIStoryboard for “Making facility reservation” Scenario 155
6.19 Object diagram for Facility Reservation 156
6.20 Object diagram for customer identification 157
6.21 Object diagram for reservation status 157
6.22 Object diagram for facility status 158
6.23 Object diagram for fitness club system 159
6.24 State Chart defining UI behaviour 160
6.25 Top RSL metaclasses in MOLA-ready metamodel 163
6.26 Constrained language sentences for activity scenario elements 164
6.27 Phrases and terms used in MOLA-ready metamodel 165
6.28 Notions and their relationships 166
6.29 The used fragment of UML Kernel package 168
6.30 The simplified Interactions package 169
6.31 Traceability elements used for transformations 169
6.32 Main procedure of the transformation 170

6.33 Main procedure for building the static structure 171
6.34 The “fixed infrastructure” of architecture model to be generated 172
6.35 Static package creation 173
6.36 Procedure building the Application logic elements 175
6.37 Procedure building the data transfer objects 176
6.38 Procedure building UML actors 177
6.39 Procedure building data access objects 178
6.40 Procedure building business logic components 179
6.41 Procedure managing the behaviour 180
6.42 Procedure building interactions 181
6.43 Procedure building lifelines 182
6.44 Procedure building invocation lifelines 183
6.45 Procedure traversing the activity graph 184
6.46 Procedure traversing one edge 186
6.47 Procedure distinguishing message kinds 187
6.48 Procedure generating actor-to-system messages 188
6.49 Procedure generating system-to-actor messages 189
6.50 Procedure generating system-to-system messages 191
6.51 Procedure generating system (“business”) lifeline 192
6.52 Utility procedure providing verb phrase text 193
6.53 Utility procedure providing operation name 194
6.54 Utility procedure converting string to UpperCamelCase 195

7.1 Package structure of the Java 5 metamodel 200
7.2 Representation of programs in the Java 5 metamodel 202
7.3 Example model for the ‘Hello world’ program 204

8.1 TraceabilityLink as a subclass of SCLRelationship 208
8.2 RSL traceability links 209
8.3 UML traceability links 212
8.4 Inter-language traceability links 213
8.5 Traceability link visualisation example 215

Chapter 1

Scope, conventions and guidelines

1.1 Document scope

This document provides an overview of a reuse-oriented modelling and transformation language definition that is used in the ReDSeeDS project. The modelling and transformation language in ReDSeeDS is part of the *software case language* (SCL), which is used to describe all artifacts in a software case. The ReDSeeDS Requirements Specification Language (RSL), which is also part of SCL, has already been described in the previous ReDSeeDS deliverables (though some technical improvements to RSL are described also in this document). The modelling language in SCL is used to define two more basic artifacts of a software case - architectural models and design models. In contrast to RSL, which is a completely new language created within the ReDSeeDS project, the modelling language in ReDSeeDS, which is named also Software Development Specification Language (SDSL), is not a new language. It is just an appropriate subset and guidelines for usage of some modelling language used in model-based software development practice. The given document mainly describes appropriate subsets of *Unified Modelling Language* UML, which is currently the most popular language for software design. The reuse aspect of a software case to a great degree is assisted by the fact that the next artefact in a software case is partially obtained from the previous one by means of automatic transformations, which in turn require a language to be specified. Therefore the description of the model transformation language MOLA is included in this document. The MOLA language is being developed by the University of Latvia IMCS team. The MOLA language integrates well with other components of SCL. This document contains also examples of model transformations applicable in ReDSeeDS - both informal ones and specifications in MOLA. Some aspects of programming languages, namely Java, in the context of software case and SCL are also discussed, as well as traceability in SCL.

1.2 Conventions

Whenever possible, the language descriptions in this document are based on the corresponding metamodels. The metamodels, in turn, have been developed in accordance with the metamodelling guidelines in the deliverable 3.1.

Lowest level package descriptions use the following notation conventions:

- sans-serif font is used for names of classes, attributes and associations, e.g. Requirement
- if a class name is used in description of package other than the one it is included in, it is preceded with package name and a double colon ("::"), e.g. RequirementsSpecifications::Requirement
- ***bold/italics font*** is used for emphasised text, e.g. ***Abstract syntax***

Class colours used on the diagrams indicate membership of the packages. Introduction of colours is intended to enhance readability of diagrams which contain classes from different packages (e.g. blue colour denotes that classes are from Requirement packages, yellow are from RequirementRepresentation package and green are from DomainElement package). Refer to deliverable 2.4.1 for more information about RSL packages and colours. Appropriate colour coding is used also for MOLA metamodels.

1.3 Related work and relations to other documents

Model driven development of software has become the de-facto standard for software development in practice, though actually only a few books (see e.g.,[MM03, KWW03, MSUW04]) provide a theoretical background for it. Therefore the systematic approach in ReDSeeDS to building software cases is quite a pioneering work in this area. The main modelling language for building software models, no doubt, is UML [Obj05b], but again, only few sources [BRJ05, Ś05] provide some methodological guidelines how this should be done. The approach used in ReDSeeDS is to a great degree original. And so is the whole idea of using SCL as a unifying principle for building reuse-oriented software cases. The model driven development to a great degree is impossible without model transformations which should be built in an appropriate language. The standardizing efforts by OMG, which resulted in the draft for the MOF QVT language [Obj05a], have stalled to a great extent. Therefore ReDSeeDS uses the original model transformation language MOLA, being developed by University of Latvia IMCS ReDSeeDS team [KBC04]. Another aspect to be incorporated in SCL is traceability. In ReDSeeDS, traceability plays a major role in facilitating reuse. This field of application is considered by Antoniol et al. in [ACC$^+$02] as well as Dahlstedt and Persson in [DP03]. Meta-models for traceability, including various traceability link types, are presented by Ramesh and Jarke in [RJ99], Ebner and Kaindl in [EK02], and Letelier in [Let02]. Furthermore, traceability facilities are also provided by the UML (see [Obj07]). Since all languages constituting SCL are defined by means of metamodels, this deliverable significantly relies on the previous deliverable 3.1, where the metamodelling principles for ReDSeeDS were defined. The technical improvement of RSL metamodel relies on the previous version defined in the deliverable 2.4.1.

1.4 Structure of this document

Chapter 2 is introduction to the topic of this work. It describes the general principles of SCL and main components of it. Chapter 3 describes the main changes that are needed to transform the RSL metamodel of deliverable 2.4.1 into a technical, tool-ready one. Chapter 4 describes the software development specification language. The principles of SDSL and the role of UML for it are described. Example UML subsets for architecture and detailed design models are provided. Other possibilities for SDSL are discussed briefly. Chapter 5 describes in detail the transformation language MOLA. Chapter 6 describes the possibilities of automatic transformations between models. There examples of transformations from requirements to architecture models are provided, both informally and as an example of transformations in MOLA. Chapter 7 describes transition to code in SCL. This problem is discussed on the basis of Java 5 language, including the relevant fragments of Java 5 metamodel. Finally, Chapter 8 describes traceability facilities in SCL.

1.5 Usage guidelines

The reuse-oriented modelling and transformation language definition document should be used as a book that guides the reader through the basic technologies, ideas and concepts of reusable case repositories and their meta-modelling. It provides the concrete implementation of the meta-modelling approach for reusable cases, which is defined in deliverable 3.1. It should be used mainly by ReDSeeDS users who wish to understand the background and main concepts of SCL and its extensibility. Further, ReDSeeDS users with some experience in meta-modelling could use it as reference and manual for extending the SCL with their preferred language, for instance decision tables.

Users of the this document are expected to know the basics of metamodelling and MOF (Meta Object Facility) specification [Obj06]. Basic knowledge of graph theory and model transformation theory may be helpful to understand the background of some sections in this document.

Chapter 2

Introduction

2.1 General principles of SCL

The *software case language* of ReDSeeDS is the language, which is used to model all artifacts in a software case. These artifacts may be requirements, architectural models, design models, source code and model transformations between these artifacts. Thus, SCL must be capable to represent all these artifacts and relationship mappings.

The languages used to model those artifacts are described in section 2.2. For every kind of artifact, one language is picked as an example in the ReDSeeDS project. But, as mentioned already in Deliverable 3.1, different software developers use different languages during the software lifecycle. One of the main goals of SCL is not to restrict those engineers into utilising a language they do not use in everyday practice. Instead of this, engineers should be able to extend SCL to incorporate every language they need. This is supported by a metamodel-driven approach where every language which is defined by a metamodel according to the styleguide in deliverable 3.1 can be integrated into SCL.

The general idea of SCL and the coarse structure of software cases are described in figure 2.1. Software cases consist of software artifacts. The parts of software artifacts are SCLElements and SCLRelationship. Both are abstract superclasses for the elements and relations of the languages that are used to model the several artifacts. This leads to the fact, that every class contains the attribute uid, which can be used as an *unique identifier* in the fact repository and the transformation engine even if the model changes over time. The next section explains these languages and their relation to SCL.

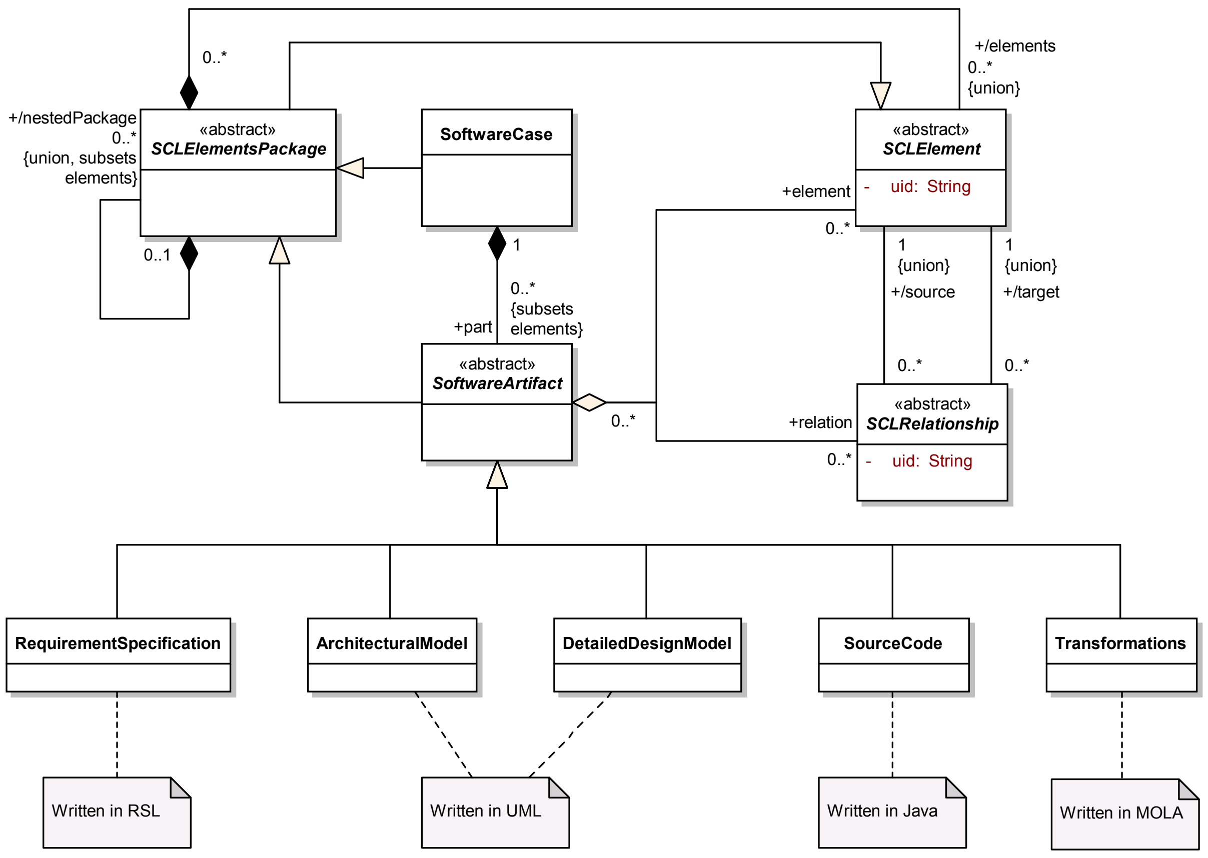


Figure 2.1: SCL and software case structure

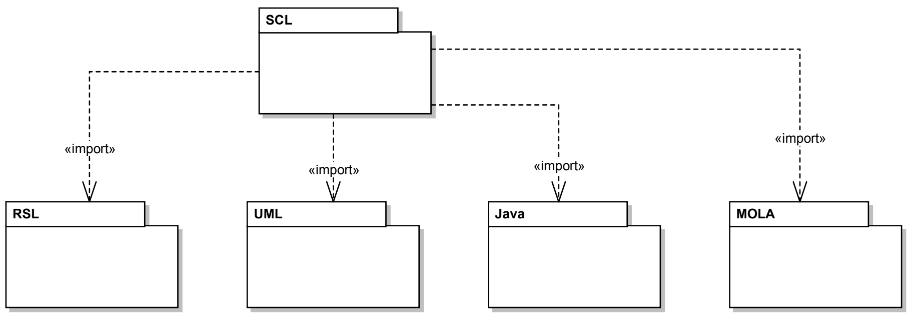


Figure 2.2: Relation of RSL, UML, Java and SCL

2.2 Components of SCL

The former section described the coarse structure of a software case and the software case language SCL. This section sketches the several parts of SCL and their relationships. Figure 2.2 shows the relations between SCL and the other languages that are depicted in this section.

The ReDSeeDS intention is to reuse software cases on the basis of their requirements. Hence, the requirements language is probably the most important part of SCL. This language is RSL, which was developed in workpackage two (WP2) of the ReDSeeDS project. Chapter 3 describes several changes to the RSL metamodel so it is ready to be used as a basis for tools. The concrete syntax and conceptual intention are left unchanged, only technical issues are clarified.

The second part of a software case is the architectural one. So, SCL has to contain some language for modelling of architecture. The *Unified Modelling Language* UML is currently, together with some UML profiles, the most popular language for modelling architecture. Hence, UML is used as modelling language for the architectural part of SCL. Section 4.2 gives some more detailed explanation how special UML facilities are used in SCL.

Beside architecture, UML is also used for detailed design models all over the world and is one of the standard languages in this area. Thus, UML is also used for detailed design models in SCL. Detailed explanation is presented in chapter 4.2, too.

The fourth part of a software case is the source code. In the ReDSeeDS project, Java is used as a sample language for source code, but every other language can be used with with not much extra effort.

Chapter 7 depicts the usage of Java in SCL in a greater level of detail.

As the last part of a software case, model transformations are used to convert requirements, architectural and detailed design model, and code into each other. Of course, not every transformation is possible. For example, it is not possible to transform Requirements directly into code. A detailed explanation of the transformation approach is given in chapter 6. The model transformation language MOLA is used to notate and execute those transformations. The general principles of MOLA are depicted in chapter 5. The idea of transformations and some examples are described in chapter 6.

2.3 SCL and the ReDSeeDS software case repository

As SCL is the language for notation of software cases, this means requirements, architectural and design models, and even code and TGraphs are used as technologie for the ReDSeeDS software case repository in which abstract syntax graphs of all those artifacts are stored, this chapter explains the relation between the ReDSeeDS Software Case Language and the ReDSeeDS Software Case Repository.

The metamodel of the SCL is the schema for the TGraph repository. That means, every entity in the metamodel is treated as a vertex class in the repository whereas every relationship is treated as an edge class. Since the fact repository is used as a database for querying and reuse, it stores only abstract syntax graphs. Thus, all parts of SCL that describe only concrete syntax are not related to the fact repository.

Since the fact repository does not offer support for the concrete syntax of SCL, other tools have to be used for creation of requirements, models and code. For requirements this tool may be based on the RSL editor which was developed by University of Koblenz and operates directly on the abstract syntax graphs in the fact repository. For models, probably the UML tool Enterprise Architect will be used while for code some IDE like Eclipse is preferred. All these tools, except the RSL editor, work on their own kind of repository, Enterprise Architect on a Microsoft Jet Database and IDEs on plain text files. This leads to the need for a interface between the different repository technologies with the ability to import all artefacts into the TGraph fact repository.

During some software re-engineering projects the GraLab team at the University of Koblenz has gained a lot of experience in parsing such artifacts, especially program code, into abstract syntax graphs. For some programming languages, for instance C/C++ and Java, there are already parsers available that transform these languages into TGraphs, but these parsers work on older versions of GraLab. For the Enterprise Architect repository, a converter tool was developed in a student thesis. Since these interfaces are dynamic and extensible, there is also the possibility to use alternate interface technologies such as web services.

All experience with fact repositories in software re-engineering and experiments with the converter tool for Enterprise Architect show, that it is not practical to hold the fact repository consistent to the tool repository all the time, since the conversion between the repositories takes too much time. Instead of this, it is best practice to have some kind of “nightly build”, that means, the data of the tool repository is extracted to the fact repository periodical.

As stated above, the fact repository contains abstract syntax graphs based on a graph schema. This leads to two advantages, the first one is that the query and reuse mechanism is independent of the tools used as long as there exists an export/import interface or some converter tool. The second advantage is that it is relatively simple to extend SCL by new representation or diagram types, such as decision tables. To do this, it is sufficient to implement a converter tool for a tool supporting these representation or diagram languages and to extend the SCL metamodel for the fact repository by the appropriate desired representation.

Chapter 3

Adaptation of RSL to SCL

Several experiments with and discussion about the ReDSeeDS Requirements Specification Language (RSL) show that there is a need for a technical version of the RSL metamodel. This section describes the main changes that are needed to transform the RSL metamodel of deliverable 2.4.1 into a technical, tool-ready one.

The main goal of the technical metamodel is the restriction of the modelling concepts used to EMOF. This is necessary since most modelling tools available support only EMOF but not full CMOF.

One of these tools is MOLA, the model transformation language used in ReDSeeDS. Another one is the fact repository tool JGraLab, which supports the EMOF features used in RSL and some of the used CMOF features, for instance redefinition and subsetting of rolenames.

Another goal of the technical metamodel is to provide a more convenient metamodel for tool developers. To reach this goal, the RSL metamodel has to be simplified at some points and some hurts of the principle called separation of concerns have to be cured. The next sections describe the changes in detail. First the general and overall changes are described, then changes which are special to a single package are explained.

3.1 General small changes

This section explains the changes that are not special to a single part of the RSL metamodel but affect classes in many or even all packages.

3.1.1 Specialisation of SCLElement and SCLRelationship

As mentioned in section 2.1, the classes SCLElement and SCLRelationship are abstract superclasses of the classes of the several modelling languages. This needs to be reflected in the RSL, so all classes that are part of RSL are made direct or indirect subclasses of one of these two classes. Every entity in the metamodel is a subclass of SCLElement while every relationship in the metamodel, for example the classes Hyperlink and DomainElementRelationship, has SCLRelationship as a direct or indirect superclass.

3.1.2 Role names

The next changes affect the role names. At many associations, especially in the user interface part, there are neither association names (which are rarely used in MOF) nor role names (which should be used). To give the associations more semantics than "has to do with", it is necessary to have a role name at least at one association end. Another point why role names are needed is, that in every implementation of the metamodel, the implementation of that association must have an identifier. In Java, role names at far ends of associations will be used as attribute names in the local classes. In other implementations like the TGraph laboratory JGraLab, the role names will be transformed into EdgeClass names. Without a role name at least at one association end of each association, no implementation of these associations is possible.

In some places in the RSL metamodel the visual representation of the role names and the model of these role names in Enterprise Architect differs. To give an example, there are two classes A and B, and an association between them. In the diagram, the role name of A seems to be a and the role name of B seems to be b. But in the Enterprise Architect model, the role name of A is b and the role name of B is a, and in the diagram the role names are placed at the wrong association ends. Probably, the role name was just moved as graphic object but not changed in the association definition. Errors like this one occur several times in the RSL model and have to be corrected to use the model as basis for a tool.

Another change at the role names affects the role names that redefines target or source at the far end of associations connected to a subclass of UML::DirectedRelationship. These are for instance linkedElement at the association from Hyperlink to Element and linkedTerm at the association from TermHyperlink to Term). To get the RSL metamodel tool-ready, the role names target and source which are defined at the class UML::DirectedRelationship should be retained. Otherwise, handling relationship classes would be unnecessary complicated.

Further, number of all role names have to be changed to singular, even if the multiplicity at these association end is greater than one. In the RSL metamodel, role names in plural form are used several times.

3.1.3 Relationships

Beside the change to role names at relationships described above there are two more changes at the relationship classes necessary. First one is to check all classes in the RSL metamodel that are thought of as relations or links between other entities. These are for example the classes Hyperlink with all its specialisations. To get a clear distinction between entities and relations in the metamodel and thus vertices and edges in the TGraph-based fact repository, it is necessary to have UML::DirectedRelationship as a superclass of all these classes. Thus, instances of these classes will be treated as edges in the fact repository graph which leads to smaller models and better understandable algorithms and queries.

The second change to relationships affects the usage of compositions between the relations' sources and the relations themselves. The intention of this composition is to express that a relation may not exist without a source and thus if the source is deleted the relation should also be deleted. In fact, these hold also for the target, a binary relation, as UML::DirectedRelationship is, may not exists without a valid target. But in contrast to the source, the target is connected to the relation by a regular directed association and not a composition. One of the main principles of software-technology states, that **same** things have to be treated in the **same** way. So either both or none of these associations should be a composition. In UML both associations are simple associations and not compositions, so the same modelling style should be used in the RSL metamodel. For usage of the RSL metamodel as a schema for the fact repository JGraLab it does not matter if compositions or regular associations are used at this point, JGraLab will treat every subclass of UML:DirectedRelationship as an EdgeClass and thus every instance as an edge - which may not exists without a valid start and target vertex. So the semantics of the compositions are not lost.

3.1.4 Abstract classes

There is a small amount of classes spread over the RSL metamodel that are thought of but not marked as abstract. To ensure that no instances of these classes may be created, these classes have to be marked as abstract. Beside this, all abstract classes are additionally marked with the stereotype «abstract», since experience with modelling shows, that the default marking just writing the class name in italics is not sufficient for humans.

3.1.5 Multiplicities

Similar to the missing role names there are missing multiplicities spread all over the RSL metamodel. This is no real problem in most cases since MOF and UML have default multiplicities, but since there is some controversy about the "right" defaults, multiplicities should be put at all association ends.

3.1.6 Navigable associations

At some associations, especially at associations starting at Hyperlink and its subclasses, there are navigation markings used. These navigation marks are quite useful in some cases, but an unnecessary restriction in this concrete case. One example is the reflexive hasSynonym-relation at the class Term. It is navigable in only one direction, so if the words TFT and LCD are part of the terminologie and there is a hasSynonym relation from TFT to LCD leads to problems when searching for all synonyms of LCD, since the relation from TFT to LCD is not navigable from LCD. Beside this, the fact repository JGraLab always provides relations that are navigable in both directions.

3.1.7 Redefinitions

The constraint redefines is used in the metamodel quite a lot. In most cases, an abstract association is redefined using this constraint. To make the conceptual distinction between abstract and concrete association explicit in the metamodel, abstract associations are marked as derived union. This is done using a / as prefix of the role name and adding the constraint union. The concrete associations are then no longer redefinitions of the abstract ones but subsets. Thus, the constraint redefines is replaced by subsets. This leads to a better understandable model which is more compliant to CMOF.

3.1.8 Compositions and aggregations

Compositions and aggregations are used in the RSL metamodel without a clear distinction. In some places an aggregation is used even if the semantics of a composition is needed, in other places the composition is used even if the semantics of the composition are not needed. So there are several places where aggregations have to be replaced by compositions and vice versa. Since some classes, especially HyperlinkedSentence and its subclasses, participate at more than one composition, it is necessary to enforce that every entity can be of at most one composition at a time. This can be done using some {xor}-constraints at the concurrent compositions.

3.2 Changes to single packages

While the previous section depicts the changes that are necessary in several or even all parts of the RSL metamodel, the following subsections explain changes that are restricted to a single package. Every package that has to be changed is explained in a separate subsection.

3.2.1 Attributes

In the package RSLSyntax::Kernel::Attributes two classes have to be changed, the diagram in figure 3.1 shows the updated model. The first one is the class AttributeValue, which models the value of an individual Attribute of an object. This class is replaced by an attribute data at the class Attribute, which is the second class to change. This change is necessary since otherwise every instance of Attribute may have only objects as values which are subclasses of AttributeValue, so neither String nor Integer which are predefined in Java and thus available in MOLA and JGraLab can be used.

3.2.2 DescriptiveRequirementRepresentation

Experiments with development and usage of a first RSL-Tool show, that there are some changes necessary to the package DescriptiveRequirementRepresentations to make tool building and tool usage as comfortable as possible. The diagram in figure 3.2 shown the updates metamodel, the changes are described in detail in this subsection. In deliverable 2.4.1 there are three non-abstract subclasses of DescriptiveRequirementRepresentation, namely NaturalLanguageHypertext, ConstrainedLanguageScenario and ConstrainedLanguageStatement. While Constrained-

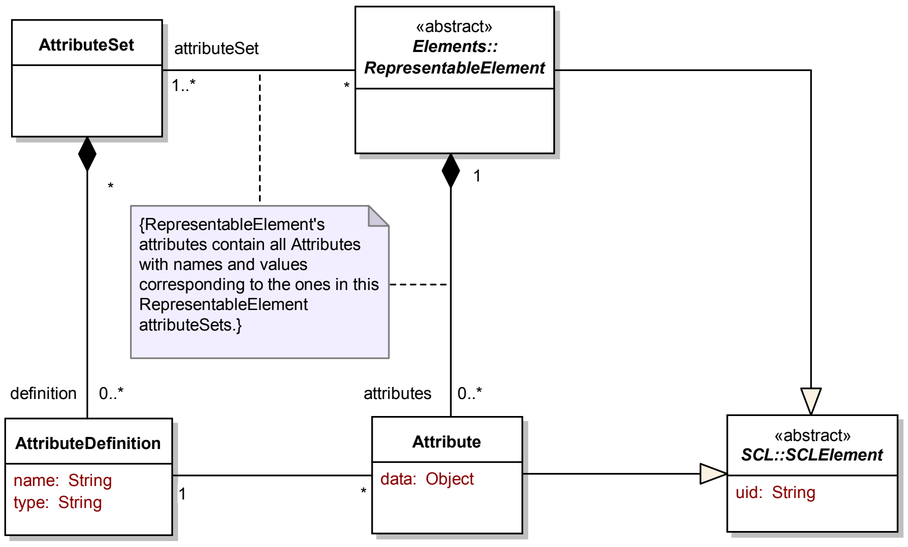


Figure 3.1: Changes to attributes package

LanguageScenario offers the ability to model a scenario with constrained language sentences in pure textual form, the other two classes have some features in common. Both intend to provide the ability of modelling requirements that are not scenarios. But while NaturalLanguageHypertext supports any number of NaturalLanguageSentences, ConstrainedLanguageStatement supports only one ModalSVOSentence. So the first adaption is to change the multiplicity and the name of ConstrainedLanguageStatement to ConstrainedLanguageHypertext.

Beside this, the test of the first RSL-Tool shows, that a user expects also to add sentences in natural language to an already existing list of constrained language sentences. If this should be supported by the RSL tool, it results in adding a metamodel class AnyLanguageHypertext, which may contain ConstrainedLanguageSentences as well as NaturalLanguageSentences. If a user adds a NaturalLanguageSentence to an already existing ConstrainedLanguageHypertext, these ConstrainedLanguageHypertext elements must be transformed into a AnyLanguageHypertext with great effort. So, from perspective of tool builders, it is more convenient to have only one class called SentenceList, which may contain ConstrainedLanguageSentences as well as NaturalLanguageSentence and thus replaces NaturalLanguageHypertext and ConstrainedLanguageStatement. Since some subclasses of ConstrainedLanguageSentence, for instance PreconditionSentence, should not be used in a SentenceList, a constraint is added to the aggregation from SentenceList to HypertextSentence.

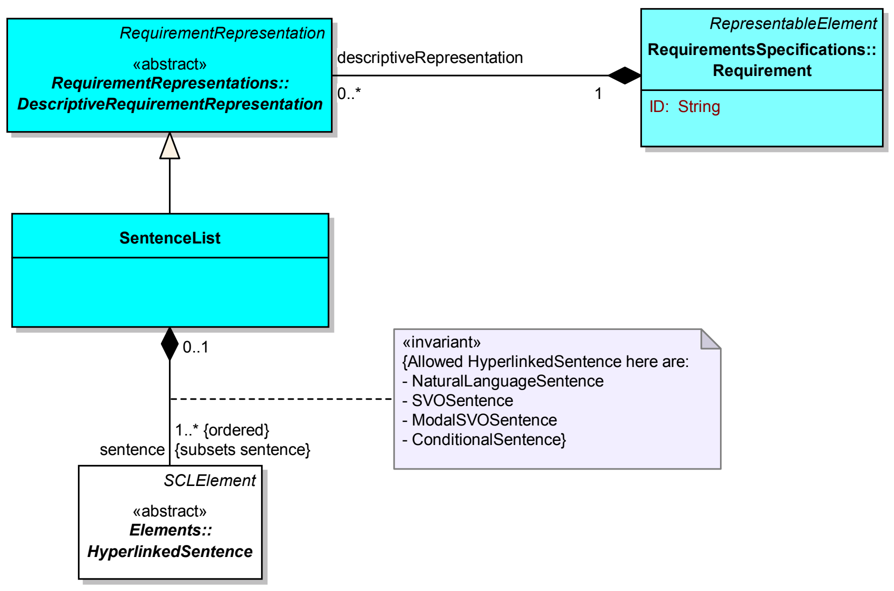


Figure 3.2: Changes to descriptive requirement representation

3.2.3 NaturalLanguageRepresentation

In the metamodel of Deliverable 2.4.1, a NaturalLanguageSentence may contain any kind of Hyperlinks. To make the calculation of similarity measures between different sentences as precise as possible, these containment of Hyperlinks in NaturalLanguageSentences has to be restricted. In the toolready metamodel, a NaturalLanguageSentence may contain only PhraseHyperlinks that are the name of a DomainStatement. The changed diagram is shown in figure 3.3

3.2.4 ConstrainedLanguageSentence

During the development of the tool ready RSL model, the original RSL model was extended by the class RejoinSentence. This class can be used to model loops in scenarios. A RejoinSentence is a subclass of ConstrainedLanguageSentence. This class is also part of the tool ready RSL version. Figure 3.4 shows the changed metamodel.

In a ConstrainedLanguageScenario, a RejoinSentence is represented by the prefix ==>*rejoin:* followed by the sentence where the loop begins. This sentence occurs only as the last sentence in scenario. Inside the loop, a ConditionSentence should be used to break the loop.

The updates in the metamodels and concrete syntax of the other scenario representations, namely interaction and activity representation, are described in the appropriate sections 3.2.5 and 3.2.6

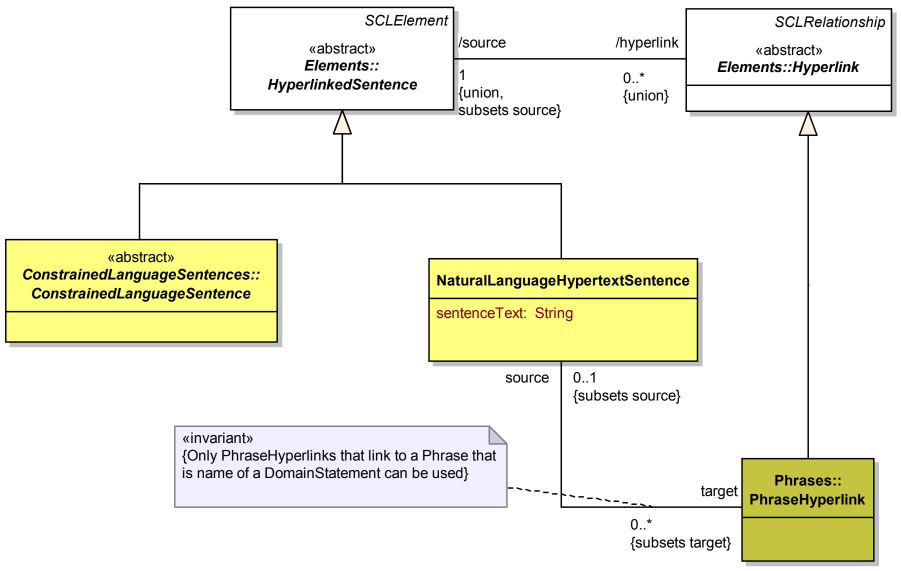


Figure 3.3: Changes to natural language sentence

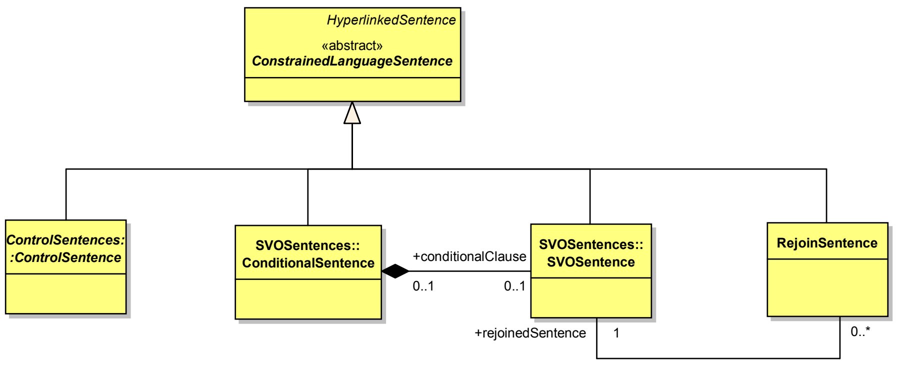


Figure 3.4: Changes to constrained language sentence

3.2.5 ActivityRepresentation

The metamodel for the representation of a scenario using an activity diagram is quite complex and, as some experiments show, not really suitable for building a RSL tool. Because of this, several changes are needed to the activity representations. Figure 3.5 shows the updated model. The most important change, which in fact leads to the other changes, concerns the class ActivitySentence and all subclasses. These classes hurt the principle called "separation of concerns". For example, the ordinary SVOSentence and the ActivitySVOSentence have the same grammar, only their occurrence differs. The same holds for all other activity sentences. So, any occurrence of these special activity sentences is replaced by the appropriate classes like SVOSentence, whereas ActivitySentence and all subclasses will be removed.

These changes lead to the next adaptation. As soon as the special activity sentences are removed, also the artificial classes ActivitySubject and ActivityPredicate, which are not more expressive than their superclasses Subject and Predicate are, can be removed.

In the RSL metamodel of deliverable 2.4.1, the nodes of the activity diagrams were activity sentences itself. In the tool ready version, this is modelled with aggregation instead of generalisation. Two new classes, namely RSLActivityNode and RSLActivityEdge are introduced. RSLActivityNode contains a ConstrainedLanguageSentence, which may be a ControlSentence or a SVOSentence. All other subclasses of ConstrainedLanguageSentence are not allowed to be part of RSLActivityNode. Every single activity in a diagram consists of the node, which is represented by RSLActivityNode and the sentence describing this activity, represented by a ConstrainedLanguageSentence. The class RSLActivityEdge models the control flow between two nodes. Every RSLActivityEdge may contain up to one ConditionSentence, which restricts the control flow. Further, due to the fact that the class RejoinSentence was introduced as described in section 3.2.4, an RSLActivityEdge may contain a RejoinSentence. In an ActvityRepresentation, a RejoinSentence is represented by a control flow edge with the guard *rejoin*. This edge points to an action occurring earlier in the scenario.

3.2.6 InteractionRepresentation

The problems mentioned above for ActivityRepresentation hold also for InteractionRepresentation. Thus, some cleanups are needed to make the tool building as easy as possible without loosing expressiveness. The changes to InteractionRepresentation are similar to the changes to ActivityRepresentation and shown in figures 3.6 and 3.7. The first adaption concerns again the specialised sentence classes. In this case they are InteractionSentence and all subclasses. These

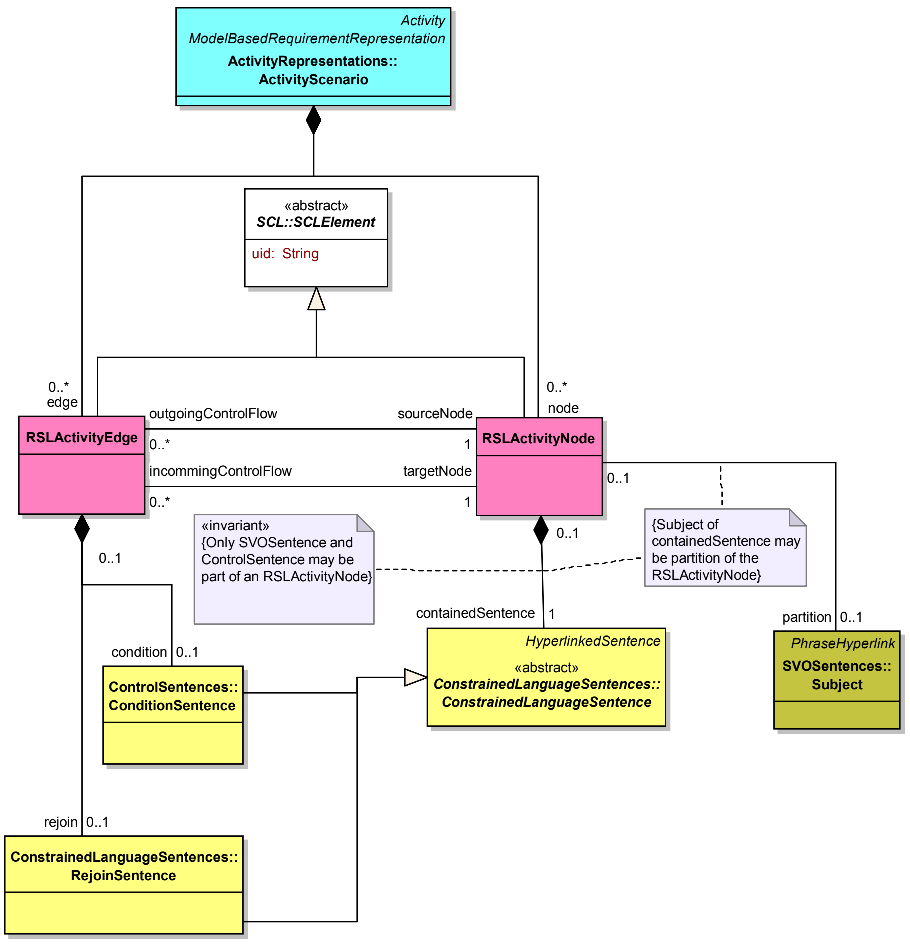


Figure 3.5: Changes to activity representation

classes are removed and replaced by the regular ConstrainedLanguageSentence-subclasses, as they hurt the principle "separation of concerns".

Further, some of the interaction sentences were used as messages. This is not a bad idea indeed, but for separation of concerns, it is better to have messages which contain those sentence. For example, occurrences of InteractionPreconditionSentence can for instance be replaced by PreconditionMessage which contains a PreconditionSentence. So, parsing of the sentence can be done by the same parser as parsing of any other PreconditionSentence. The updated model is shown in figure 3.7. The same holds for the SubjectLifeline, which is a subclass of Subject in the RSL version of Deliverable 2.4.1. To make tool building more efficient, the specialisation between those two classes can be replaced by a composition. Instead of specialising Subject, the class SubjectLifeline may contain a NounPhrase. Further, the class SubjectLifeline is renamed to NounPhraseLifeline. Figure 3.8 shows this change.

As the special interaction sentences were removed, also the class PredicateMessage has to be changed. Instead of specialising PredicateMessage from Predicate and ScenarioMessage, ScenarioMessage is the only generalisation of PredicateMessage. The generalisation to Predicate is replaced by a composition to VerbPhrase. Figure 3.6 shows the updated diagram.

Due to the fact, that the class RejoinSentence was introduced as described in section 3.2.4, a new class RejoinMessage was introduced. The diagram in figure 3.9 shows the message. The metamodel is very similar to the one of PredicateMessage, so a detailed explanation is not necessary. In an InteractionRepresentation, a RejoinSentence is represented by a message with the guard *rejoin*. The message representing a RejoinSentence repeats the message which occurs earlier in the sequence of messages. All messages between these two messages are part of the loop.

3.2.7 Phrases

A lot of the classes in Phrases package lead to problems since the names of these classes are the same as the names of classes in the Terms package. Classes in Phrases are subclasses of Hyperlink and thus are treated as edges in the fact repository. Because of this, their names are changed from xy to xyLink whenever there exists a class xy in the Terms package. Further, the class Object was also renamed according to the methodology described above. This is done since Object is a reserved word in the programming language Java which is used as implementation language of the fact repository and probably the ReDSeeDS engine.

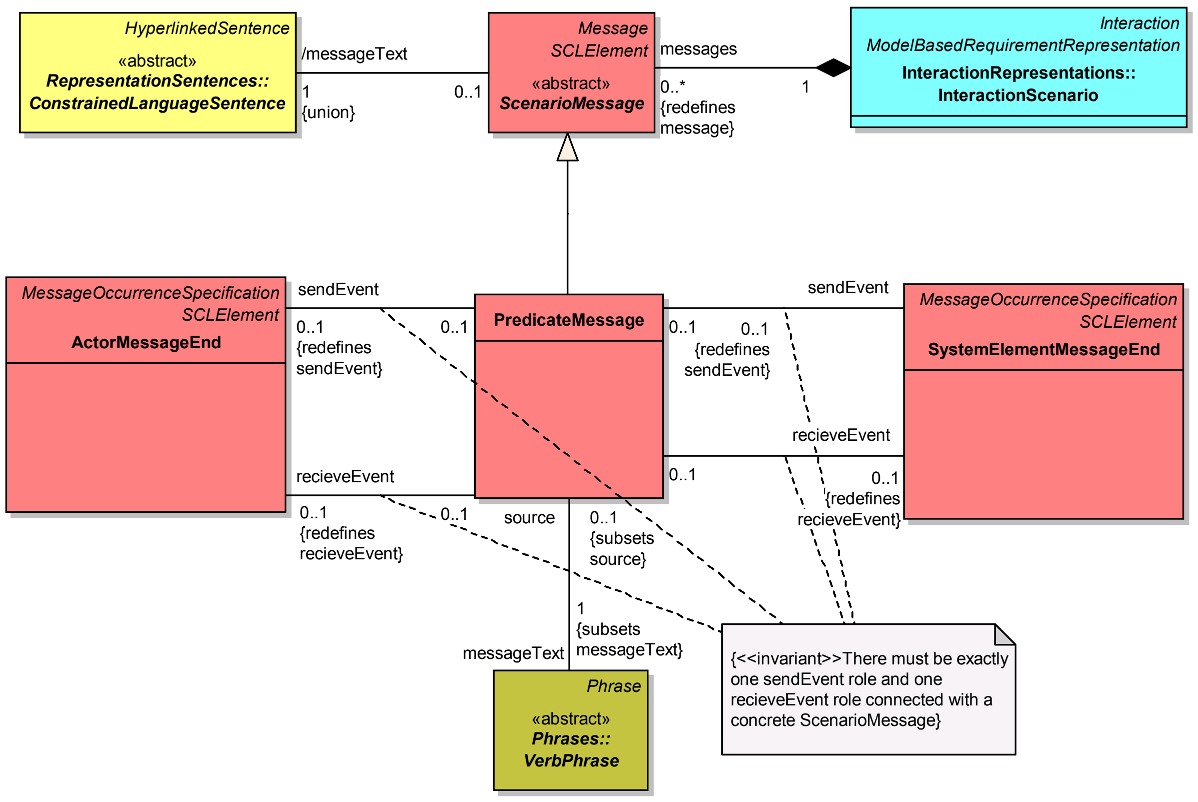


Figure 3.6: Changes to predicate message in interaction representation

Another change in the Phrases package concerns the classes Phrase, VerbPhrase and the subclasses ComplexVerbPhrase and SimpleVerbPhrase. In the RSL meta model of Deliverable 2.4.1 a Phrase consists of a Determiner, a Modifier and a Noun. It is used via an association with rolename name in several places spread all over the meta model. At many of these usages, a constraint “name must be Phrase and not one of its specialisations” is added to the association. To make the model more understandable, a new subclass of Phrase namely NounPhrase is introduced. Many occurrences of Phrase can then be replaced by NounPhrase. Further, the class VerbPhrase is remodelled, so a VerbPhrase contains a NounPhrase as object. This makes the tool building and understanding of the Phrases package more comfortable and offers the possibility to get rid of the constraints mentioned above.

The model with all these changes is presented in figures 3.10 and 3.11.

3.2.8 Terms

The Terms packages and especially the TermsInflections package, needs a general rework. As a first change, the class Term is made abstract, since there are no instances of Term but only of its subclasses. The changed model is depicted in figure 3.12. Next, the class InflectionType,

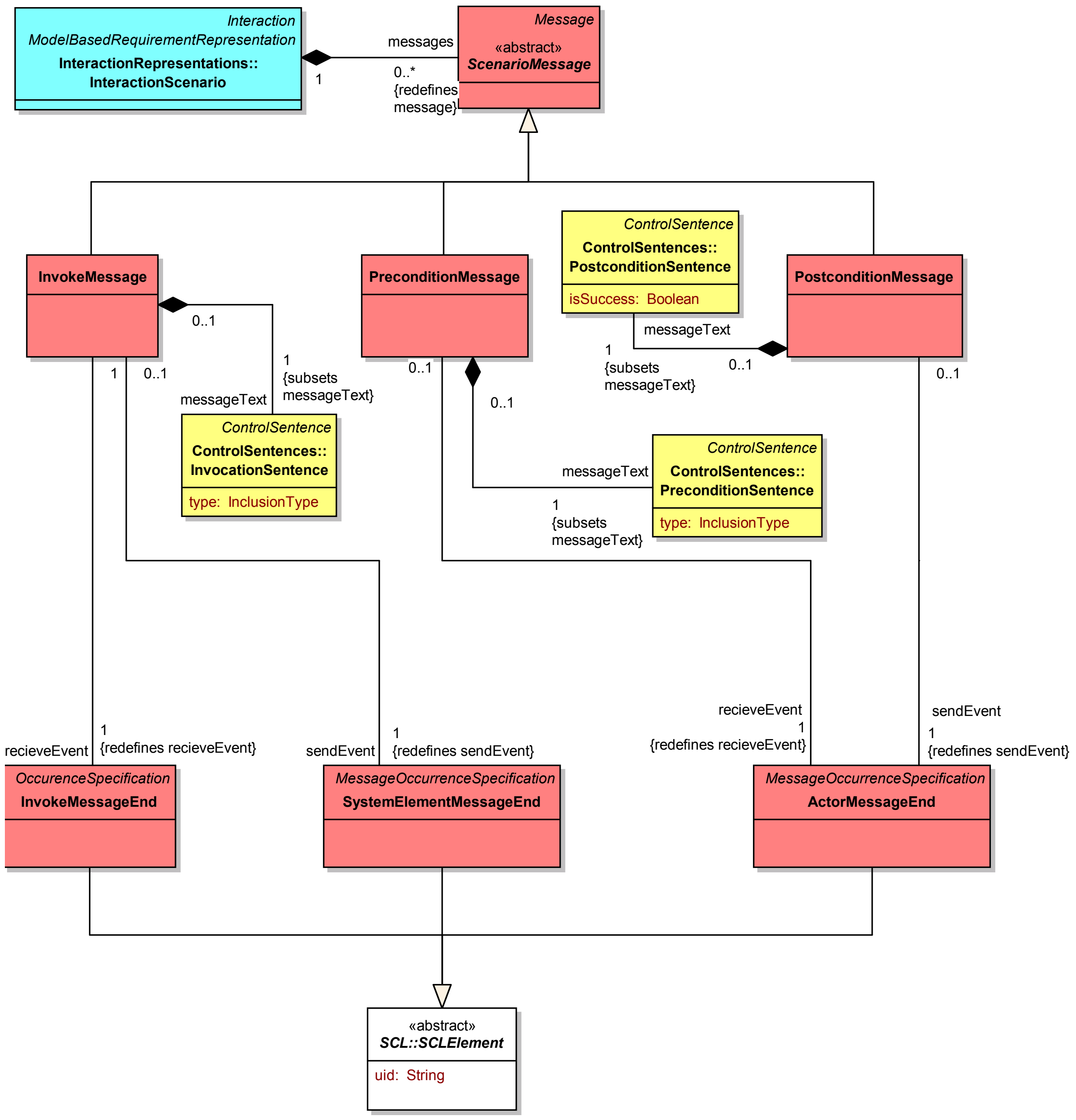


Figure 3.7: Changes to messages in interaction representation

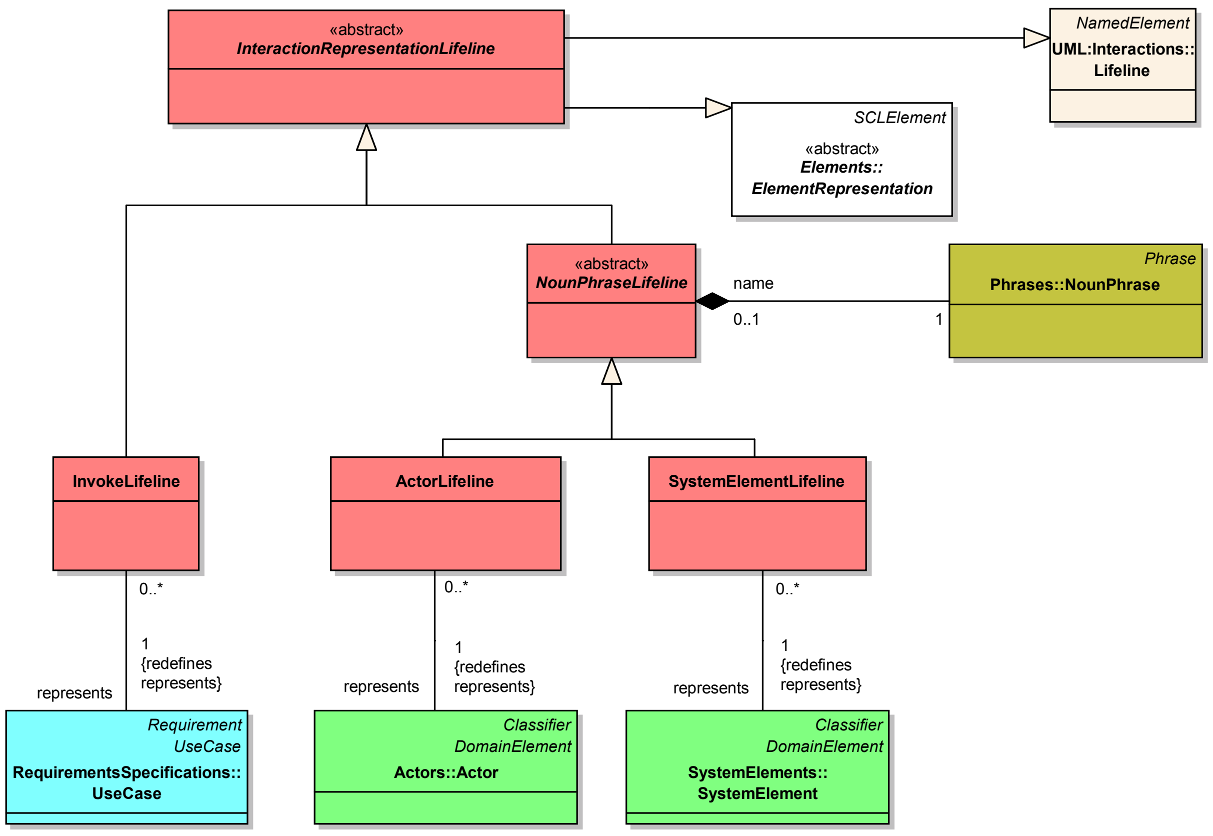


Figure 3.8: Changes to interaction lifelines

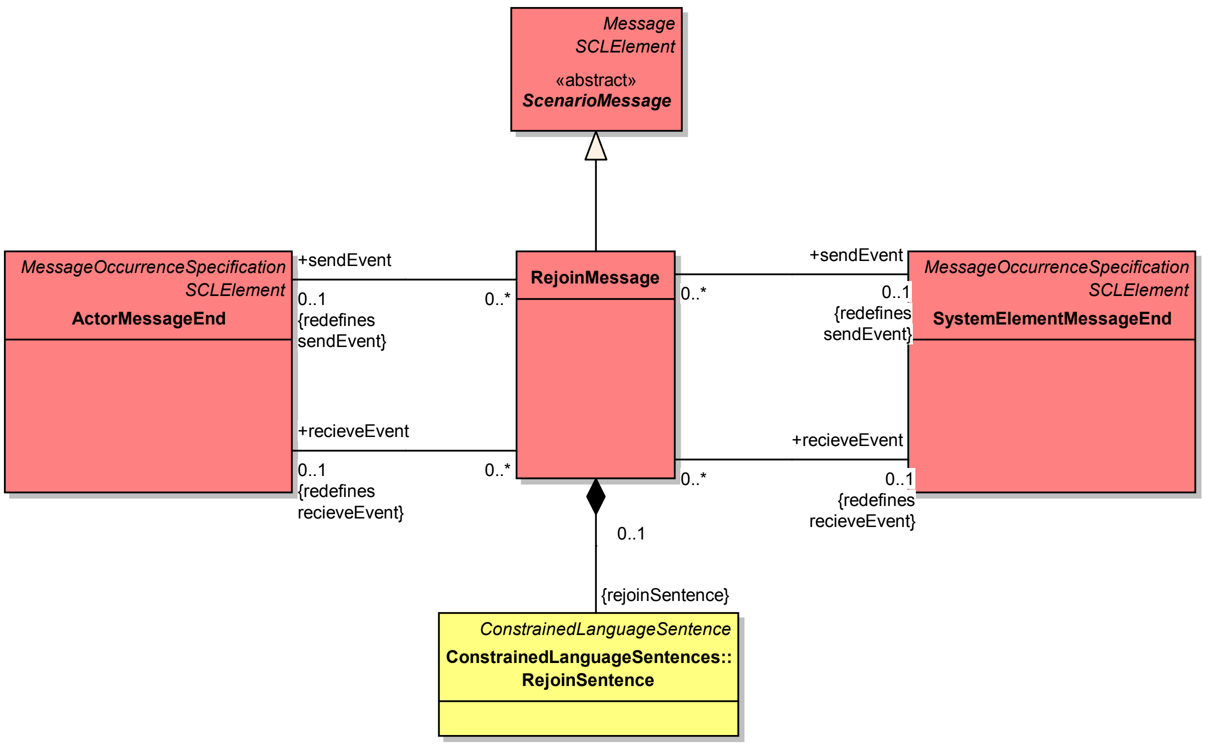


Figure 3.9: New rejoin message in interaction representation

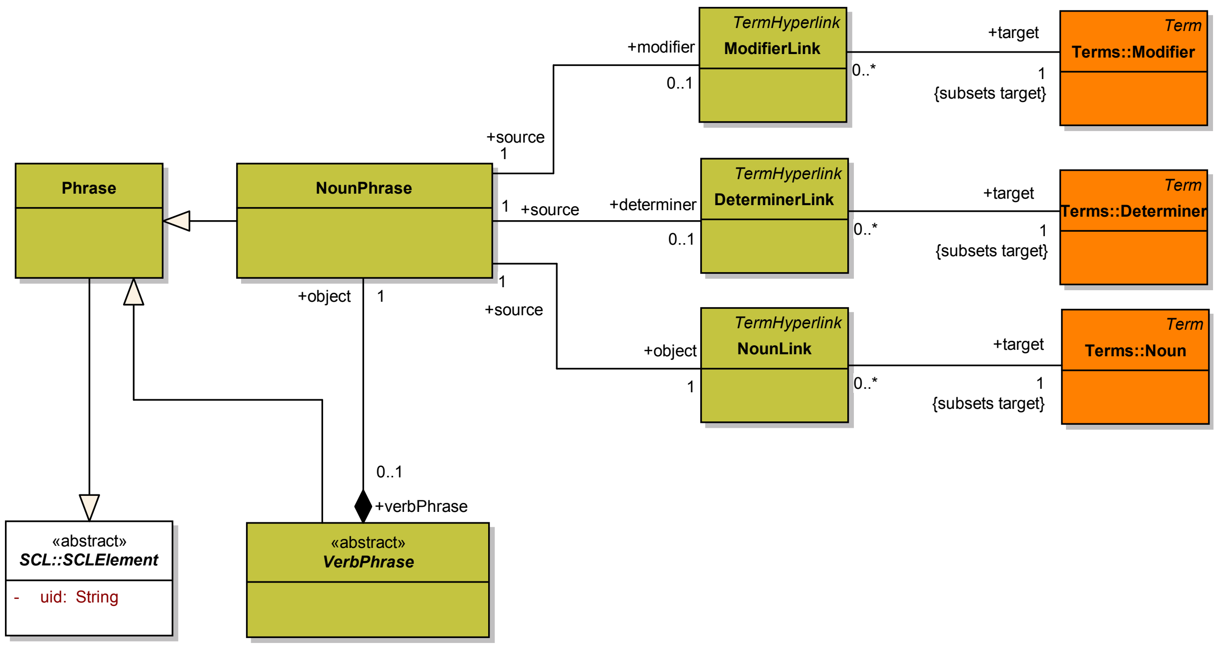


Figure 3.10: Noun phrases package

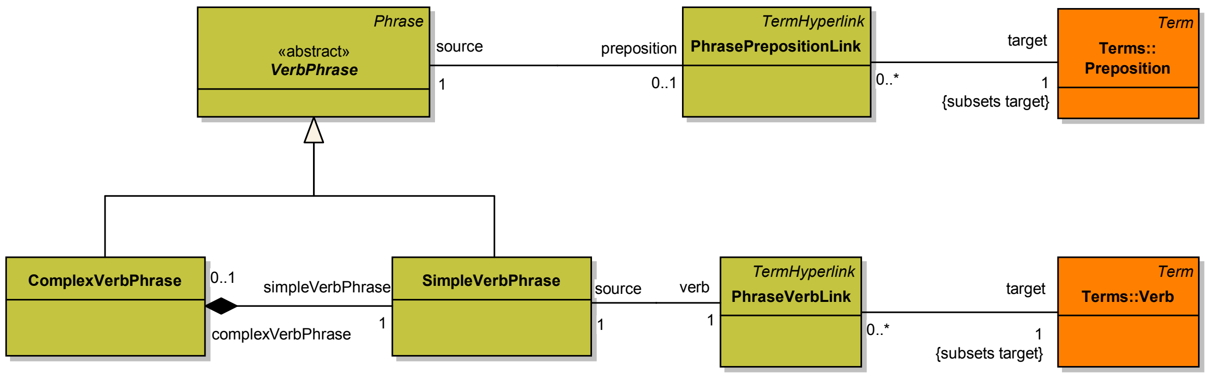


Figure 3.11: Verb phrases package

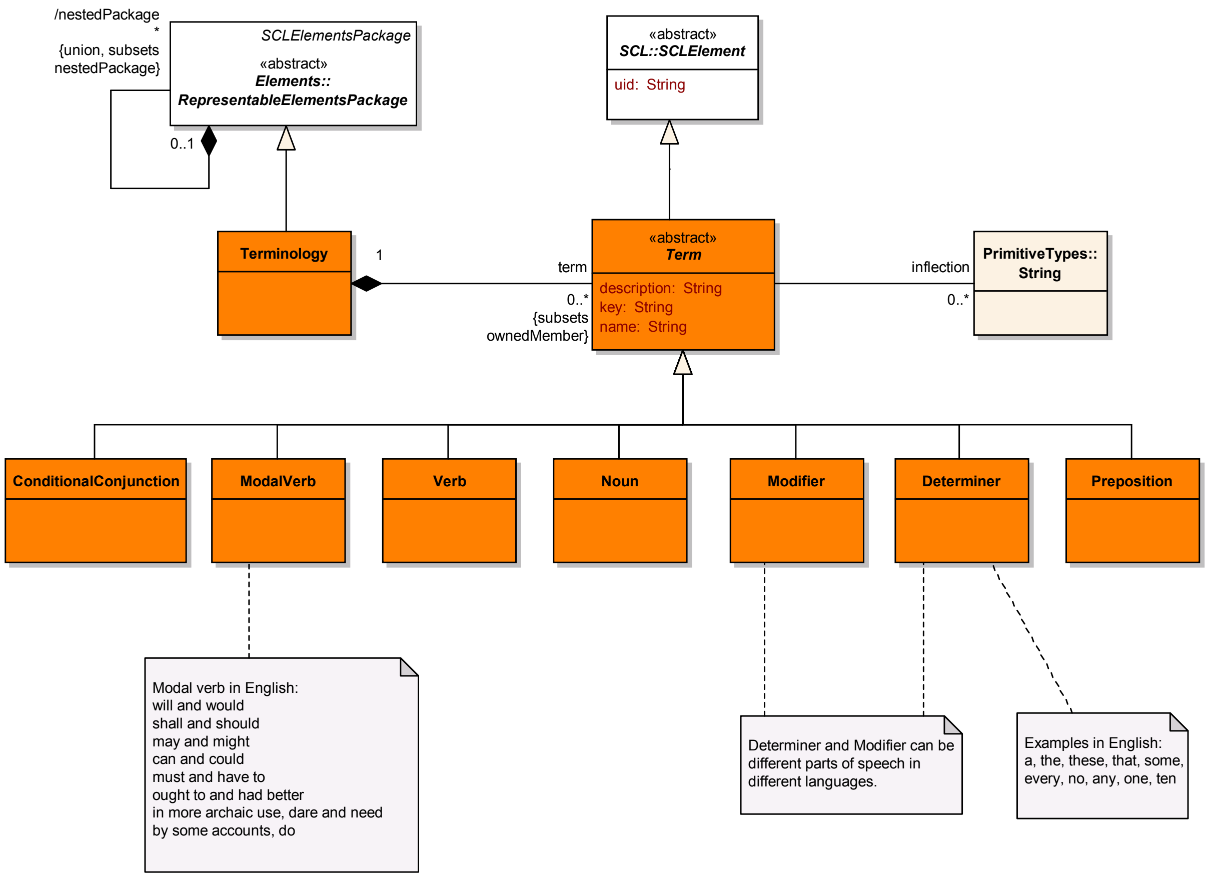


Figure 3.12: Changes to terminologie

which modells inflections of a term, is not expressive enough. For instance, it is not possible to model the inflection was of the verb to be since it is a change in the InflectionTypes GENDER and TENSE.

For this reason, the class TermHyperlink has to contain six attributes of type String, which model the inflection of an occurrence of a term more precise. The classes Lemma and Lexeme are then no longer needed and can be removed. Figure 3.13 shows the updated model.

Further, some adaptations are needed in the TermsSynonyms package. The relations HasSynonym and HasHomonym has to be changed so they reflect the nature of these relations of being symmetric. The changed diagram is shown in figure 3.14.

As last change at the terms, a link to the free dictionary WordNet[1] has to be introduced. Experiments with the RSL editor show, that it is not suitable for requirements engineers, to enter all words they need in their terminology manually. Instead of this, they should be able to use an already existing terminology wherever possible. For reasons of availability, the WordNet terminology was chosen to fill this gap in RSL. Figure 3.13 shows the updated model.

[1]wordnet.princeton.edu

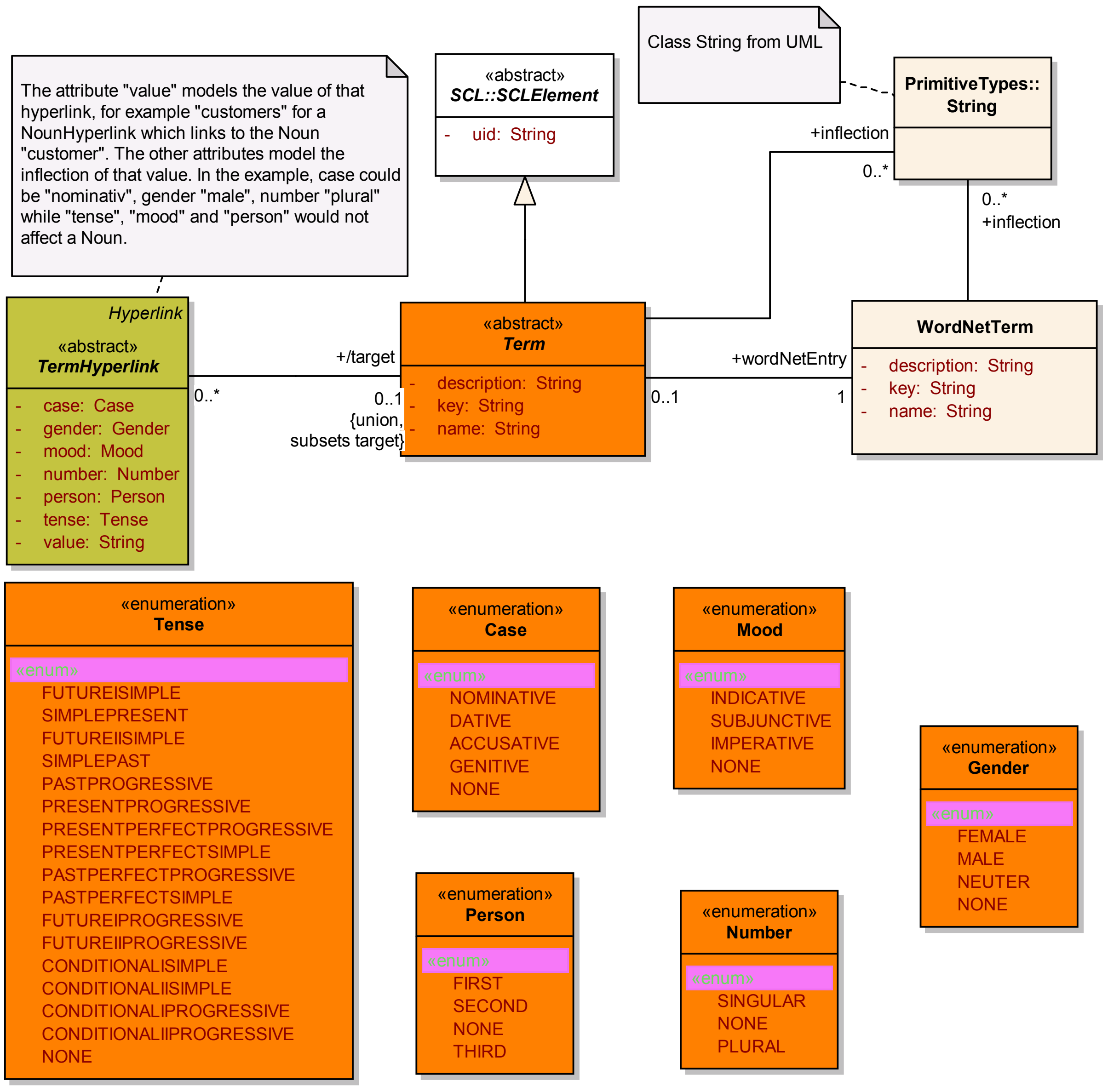


Figure 3.13: Changes to terms inflections

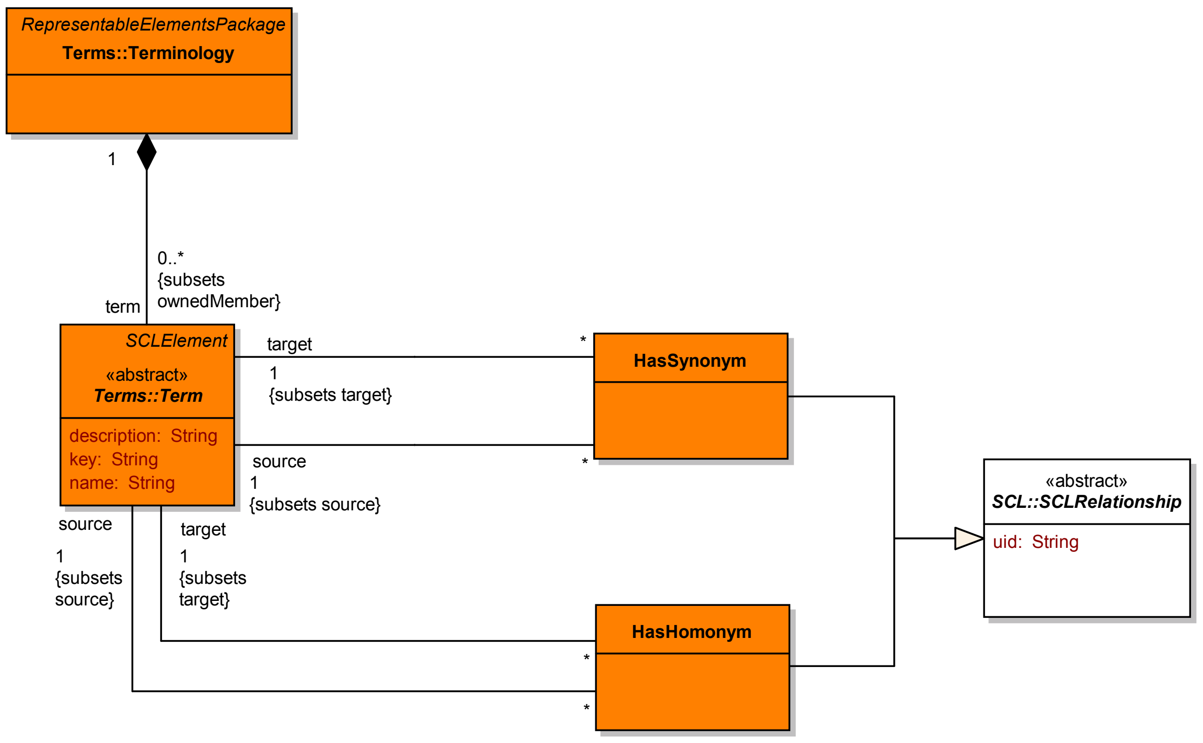


Figure 3.14: Changes to terms synonyms

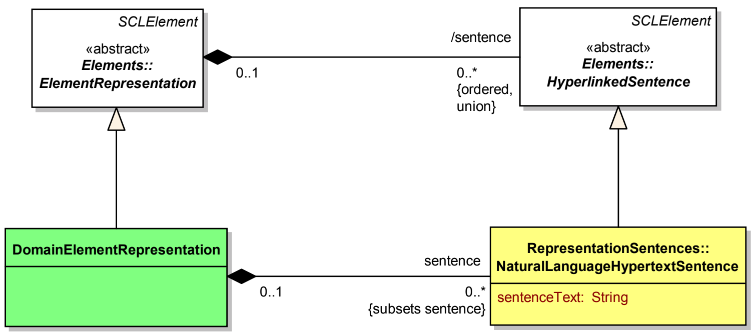


Figure 3.15: Changes to representation of domain elements

3.2.9 DomainElementRepresentation

In the package DomainElementRepresentation, the class DomainElementHyperlinkedSentence has to be replaced by NaturalLanguageSentence. A DomainElementHyperlinkedSentence contains natural language with links to phrases, which is in fact the same as NaturalLanguageSentence. For this reason DomainElementHyperlinkedSentence is completely removed from the model. Figure 3.15 shows the updated diagram.

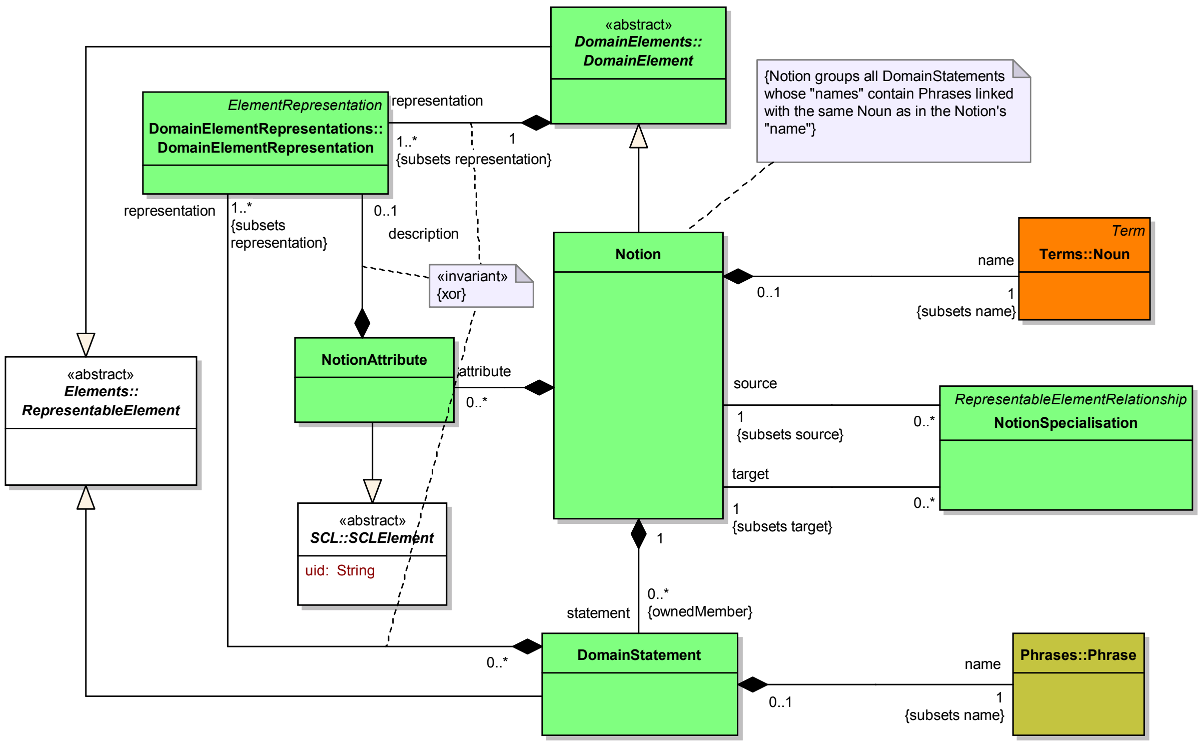


Figure 3.16: Changes to notions

3.2.10 Notions

According to the principle of analogy, the class NotionGeneralisation is renamed to NotionSpecialisation. In all other places in the meta model where a specialisation/generalisation relationship is modelled, it is named as specialisation. Figure 3.16 shows the update in package Notions.

3.3 Conclusion

This chapter presented some changes to the ReDSeeDS Requirements Specification Language that are necessary to make the development of an RSL tool as easy as possible and to make JGraLab and MOLA available to handle RSL documents.

All these changes described above affect only the abstract syntax of RSL. For RSL users, the concrete syntax remains as it is, only the mapping between concrete syntax and abstract syntax changes at some places. One of these places is the ActivityRepresentation, while in RSL of deliverable 2.4.1 an activity node itself was a sentence, it just contains a sentence in the tool ready RSL version. But the notation of the activity diagram does not change at all. The same holds for several other parts of the metamodel, the abstract syntax has changed but the concrete syntax is as it was before.

Chapter 4

Software Development Specification Language

4.1 General principles of SDSL

As it was pointed out in the introduction, the software case language (SCL) must provide facilities for description of the **architecture** and **detailed design** of a software case. In most cases, it will be some sort of modelling language, most typically UML, which should be used in a consistent way in order to facilitate reuse of a software case. Therefore it is reasonable in ReDSeeDS to say that a special **Software Development Specification Language** (**SDSL**) is used for the architecture and design steps. It does not mean that SDSL is a new language, most typically it will be the guidelines for using an existing modelling language. The appropriate language subset which in the best way fits the guidelines must be chosen, sometimes a domain-specific extension will also be needed.

The issues specific to ReDSeeDS are the necessity to be well harmonised with the formal requirements specified in RSL and the possibility to define in an easy way the traceability links to these requirements. Also the need to use automatic transformations (from requirements to architecture, from architecture to design) whenever it is possible in principle, poses some requirements to SDSL.

Thus SDSL must support the general principles of software case development, already in a very general way present in the software case metamodel in 2.1

The most typical SDSL in ReDSeeDS, no doubt, is expected to be UML. More precisely, it will be some subset of UML, including also the possibility of some extensions by means of profiles, together with some guidelines how to build the models with possible reuse in mind. The next section will sketch in more detail, how UML can play the role of SDSL in ReDSeeDS. An example of specific guidelines will also be provided.

However, another consistent design notation may play the role of SDSL as well. For example, it could be decision tables. But in any case, the chosen variant of SDSL must be able to describe architecture and detailed design models.

4.2 SDSL based on UML

As it was already pointed out, the most typical modelling language in the role of SDSL in ReDSeeDS will be UML. Therefore this deliverable will discuss mainly the UML. The section 4.2.1 will provide the general principles of usage of UML in this role. The next sections will describe in more detail the architecture and design models respectively. For each a specific example of UML-based design notation will be given.

4.2.1 Principles of UML usage for SDSL

UML is the most used modelling language for software development now. An adequate notation for software design has been the primary goal for the development of UML itself. UML is being standardised by the Object Management Group (OMG) - the world leading standardising organisation in the area of modelling and design.

The current version of UML is 2.1.1, but the most used version, supported also by most of the tools, still is 2.0. Except for some special cases (e.g., the maximal power of profile definition), the difference between these two versions is small, therefore in this document mainly UML 2.0 will be used. The main document describing the language is the superstructure document [Obj05b].

UML is defined by means of a metamodel, which is the base for its syntax and semantics. Namely the UML definition by means of a metamodel has initiated this language definition style, which is used also in ReDSeeDS. The UML metamodel is well standardised and available in its superstructure document. Therefore UML metamodel, especially due to its size, will not be repeated in this deliverable. When needed for some technical goals in ReDSeeDS, e.g., for defining transformations in MOLA, it will be taken directly from the OMG documentation. However, it should be noted, that UML tools internally not always strictly adhere to this standard metamodel. The base UML tool in ReDSeeDS - Enterprise Architect also uses a significantly simplified metamodel. This should be taken into account for data exchange (if the slow, but standard-preserving XMI exchange is not used).

UML is a very large modelling language. It now contains 13 kinds of graphic diagrams, some of which are used for very special purposes only. They cover all possible kinds of software development for all possible kinds of software or software/hardware systems - namely such has been the goal of UML creation. Therefore the primary concern in a successful application of UML for software design is the selection of a UML subset, which fits the chosen design methodology the best way.

The ReDSeeDS approach to software case design strictly specifies that two kinds of models - architecture and detailed design are used. This already significantly reduces the choice of UML facilities to be used. In particular, the most appropriate diagram kinds can be selected for both model kinds. However, each UML diagram can contain a large variety of elements, there also a strong methodological guidance is necessary. For a well defined methodology actually the total number of UML elements used is not very large and can be easily learned by developers. The restricted use of UML elements is also beneficiary from the transformation point of view - a greater part of the target model could be generated from the source model by means of transformations specified in MOLA.

Software models in UML typically consist of static and dynamic parts, the latter one describing the system behaviour at the selected level of abstraction. For the static part of the architecture model the most relevant are class and component diagrams (also deployment diagram if the physical structure is described too). Classes, components and interfaces are the most relevant UML elements for the static part. In UML model any kind of packages are used to structure the model itself. The dynamic part at the architecture level in a most appropriate way can be described by sequence diagrams. The semantically equivalent communication diagram has significantly lost its value in UML 2.0. The formal syntax of sequence diagrams is very complicated in UML 2.0, therefore mainly the basic elements - lifelines and messages are recommended for use. The value of fragments (conditional and loop ones) depends on the kind of system to be described. For simple cases the guard condition notation used in UML 1.4 sometimes is more readable (the chosen UML tool - Enterprise Architect supports both notations). The proposed in ReDSeeDS UML elements for the architecture model are described in greater detail in section 4.2.2. There a specific design methodology example is provided also, this kind of architecture notation in UML permits a high percent of the architecture model to be generated automatically from requirements in RSL.

For the design model the possible variations are fewer, nearly all design notations in UML are based on elaborate class diagrams. Accordingly, a lot of UML tools support language (Java, C++,C#,...) class skeleton generation from class diagrams by means of built-in transformations. Thus they support a small fragment of MDA approach. Various UML based design methodologies differ on what elements are really used in class diagrams. However, class diagrams support only the description of the static part of the software system. The behaviour of classes again are best described by sequence diagrams, where messages strictly conform to class operation invocations. Sequence diagrams could increase significantly the percent of code generated automatically, but unfortunately built-in transformations typically do not support them. For more specific purposes state diagrams (statecharts) can be used too. Section 4.2.3 describes the proposed in ReDSeeDS UML facilities for design in greater detail, again a transformation-ready notation example is provided.

However, UML needs not only to be restricted, sometimes it needs also to be extended. UML has built-in facilities for extension - **profiles**. The profile mechanism has been significantly redefined in version 2.0, and made even more powerful in UML 2.1.1. A profile in UML is a package consisting of stereotypes. The **stereotype** is the basic extension facility in UML, it allows to add new features to the selected UML element kind, thus actually extending the UML metamodel in a restricted way. For example, if a class is to represent an Enterprise Java Bean, features describing the bean kind can be added. In UML 2.1.1 even new links to other metamodel elements can be added. However, such a sophisticated use of stereotypes is not always so easy. Therefore frequently the stereotypes do not add new features to the element they extend. They have only the name and specify the way how the element is to be interpreted in the next development step. Thus actually an implicit UML element subclass is defined. Annotations based on such simple stereotypes are of high value for defining transformations. Most probably, namely such simple stereotypes will be defined in the ReDSeeDS profile. For example, components and interfaces in the architecture model could be annotated with stereotypes, thus specifying the way how they must be implemented in the design. Another candidate for a stereotype could be the DTO (data transfer object) stereotype for the class, saying how this class should be used in the design. The current document does not provide examples of stereotypes, they will appear in the next deliverables providing more examples of model transformations.

4.2.2 UML facilities for Architecture model

An architecture model (similarly to requirements model) should consists of both static and dynamic parts. The static part of an architecture should include all elements used in the architecture description with relationships between them. The dynamic part should describe a behaviour of these elements by including all interactions between static elements. The best notation for these two parts seems to be the UML. There is a great variety of ways of modeling architecture with the UML. To fulfill special needs of an architect certain subset of diagrams and elements should be used.

To adapt ReDSeeDS architecture to present trends in software engineering, this architecture will be modeled in the UML. As ReDSeeDS architecture is a part of the Software Case, architecture model should be simple enough to keep a coherency with RSL. At the same time architecture model should be sufficient to express typical architecture issues. Automatic transformations from Requirements model (created in RSL) to architecture model are also easier to define if only some subset of UML is used. To express elements of static part of architecture model a component diagram might be used. Suitable UML elements for a component diagram are:

- Component – “A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.” (UML Superstructure [Obj05b], paragraph 8.3.1, page 142)
  In ReDSeeDS architecture components are used for representing some part of a logic structure of the developed system. They can be used in each layer of architecture (for 4-layer architecture: UI, Application, Business, DataAccess - see Figure 4.1).

- Interface (provided and required) - “An interface declares a set of features and obligations that constitute a coherent service offered by a classifier.” (UML Superstructure [Obj05b], paragraph 7.3.24, page 82)
  In ReDSeeDS architecture provided interfaces are used for representing operations offered by concrete component, while required interfaces are used for signification of an other provided interface, which is needed. All interfaces are strongly connected with components (see Figure 4.1).

- Dependency – “A dependency signifies a supplier/client relationship between model elements where the modification of the supplier may impact the client model elements.” (UML Superstructure [Obj05b], paragraph 7.3.12, page 58)
  In ReDSeeDS architecture dependencies are used for signifying suplier/client relationships between two components through interfaces (see Figure 4.1).

- Class – “Class is a kind of classifier whose features are attributes and operations.” (UML Superstructure [Obj05b], paragraph 7.3.7, page 45)
  In ReDSeeDS architecture classes are used as parameters of operation offered by interfaces. They depict data transfer objects. Classes do not create any logic structure in architecture model 4.4).

- Package – “Package is used to group elements, and provides a namespace for the grouped elements.” (UML Superstructure [Obj05b], paragraph 7.3.37, page 103)
  In ReDSeeDS architecture package are used for grouping static elements of architecture - components and interfaces. Packages grouping components are only on the level of architecture layers, components are not packaged inside the layer packages (see Figure 4.2). Interfaces are grouped in packages, the same as components, on the level of the concrete layer. They should be also grouped in concrete layer in packages corresponding to components of this layer (see Figure 4.3). One package groups all classes used as data transfer objects.

Dynamics on the level of architecture might be presented as a kind of interaction diagram - sequence diagram. Suitable elements for sequence diagram are:

- Lifeline – “A lifeline represents an individual participant in the Interaction.” (UML Superstructure [Obj05b], paragraph 14.3.19, page 475)
  In ReDSeeDS architecture there are three types of lifeline:
  - actor lifeline – represents a lifeline of an actor, which is the main initiator of system actions
  - component lifeline - represents a lifeline for a component. It should be used for a user interface component as source of message from user lifeline.
  - interface lifeline – represents a lifeline for an interface. If interface is a part of user interface component, it is only a source of messages and if not, it can by a source and a target of messages (is in role of provided and required interface).

  Usage of above lifelines are presented in Figure (see Figure 4.4)
- Message – “A message defines a particular communication between Lifelines of an Interaction.” (UML Superstructure [Obj05b], paragraph 14.3.20, page 477)
  In ReDSeeDS architecture Message is the same as in the standard UML (see Figure 4.4).

The list of elements and their role presented above should not limit an architect in using the UML in specifying the architecture. Structure of architecture and its behaviour specified with another elements will be valuable, but will not be considered under the transformation and similarity measure rules.

Architecture model - an example

To illustrate usage of presented above elements of UML, some example diagrams will be shown.

As an typical example of Architecture model in SDSL, the model of 4-layer architecture will be used. It is the most common multi-layer architecture in today's business software, mostly because of possibility to distribute logical layers among several machines (servers). The model describes both the static structure - by component and class diagrams, and the system behaviour - by sequence diagrams.

Presentation Tier (UI) - one UI component with one interface is built for whole application. It mediates between a user and the system, "translating" user-system interaction to calls to the Application Logic tier. Although some logic can be implemented in UI (like some common sense validation of data entered by user), it is recommended to avoid realising functional requirements in this layer.

The Application Logic layer is responsible for realisation of rules described in functional requirements of the system. No business logic should be implemented at this tier, only the logic that is needed to control flow of functional requirements (e.g. Use Cases). The layer structure is designed using component and class diagrams. Application logic layer consists of components, which correspond to logically related groups of use cases. Each use case corresponds to an interface of the corresponding component. Interface contents (operations) are described by means of class diagrams. Components' creation is based on grouping logically related functional requirements. This layer uses Data Transfer Objects (DTO) for data exchange with Business Logic layer.

The Business Logic layer is a tier where all business rules of the system are implemented. The above layer, Application Logic, calls interfaces of this tier to retrieve and process data needed in flow of user-system interaction. The Business Logic layer calls the Data Access layer for basic objects operations used in performing high-level business data manipulation (for instance validation of sets of data against some aggregated data). Business logic layer consists of components, which correspond to related groups of domain concepts - notions. For each notion corresponding interface of this component is created if they are any business methods resulting from functional requirements concerning this notion. This interface is treated as a service providing business methods.

The Data Access layer is a direct access to data source (like database or flat file) of the system. Please note that there can be more than one data source for a given system. In this tier interfaces exposing basic CRUD (Create/Read/Update/Delete) operations on data access objects (DAOs) are implemented for every notion.

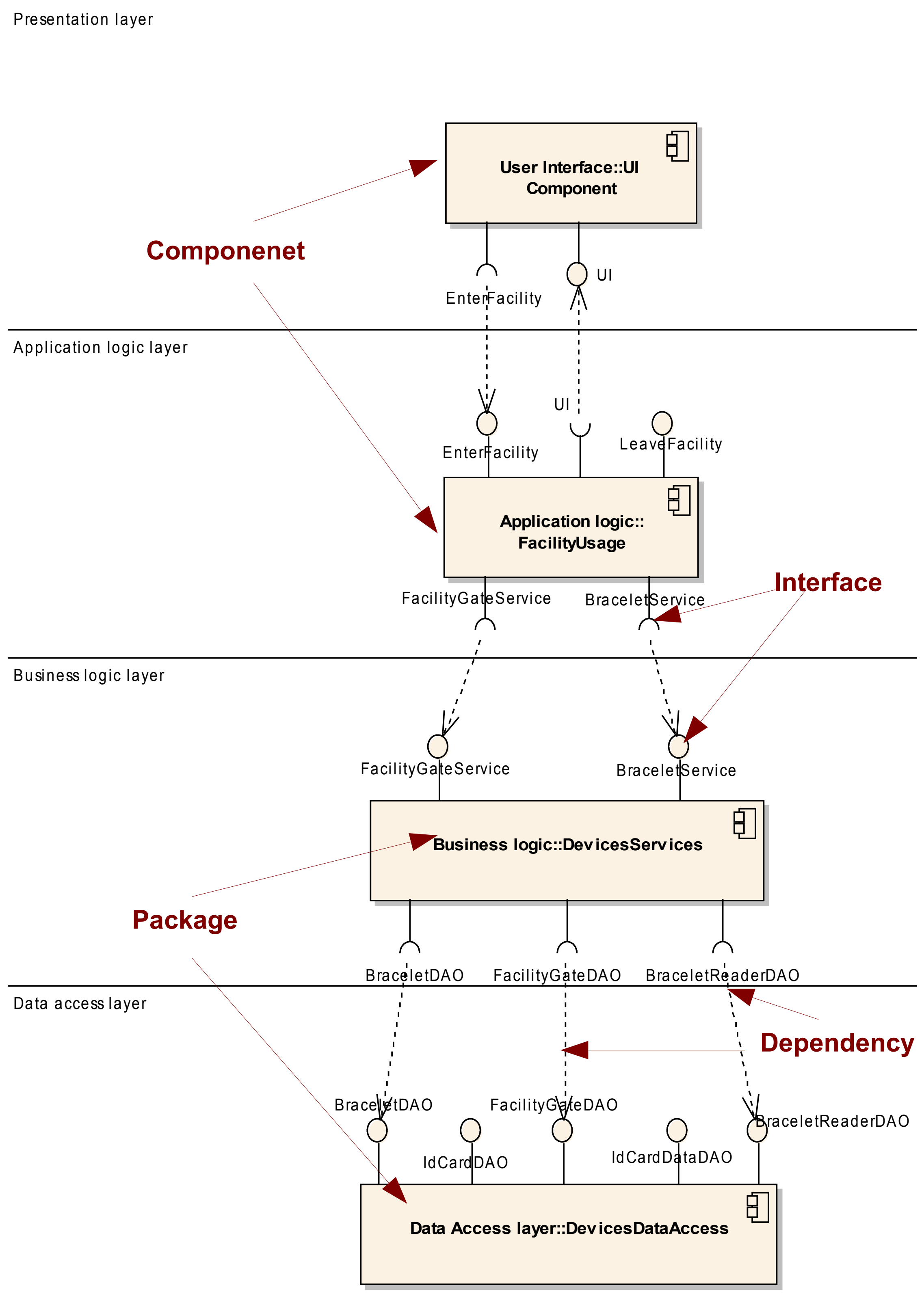


Figure 4.1: 4-layer architecture model example

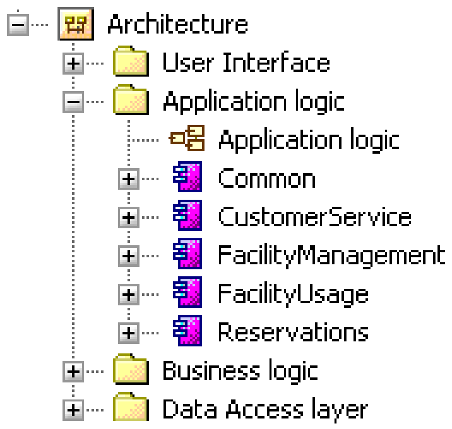


Figure 4.2: Example of UML packages for components

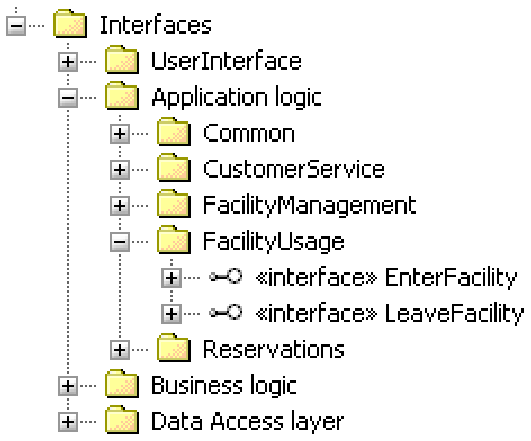


Figure 4.3: Example of UML Packages for interfaces

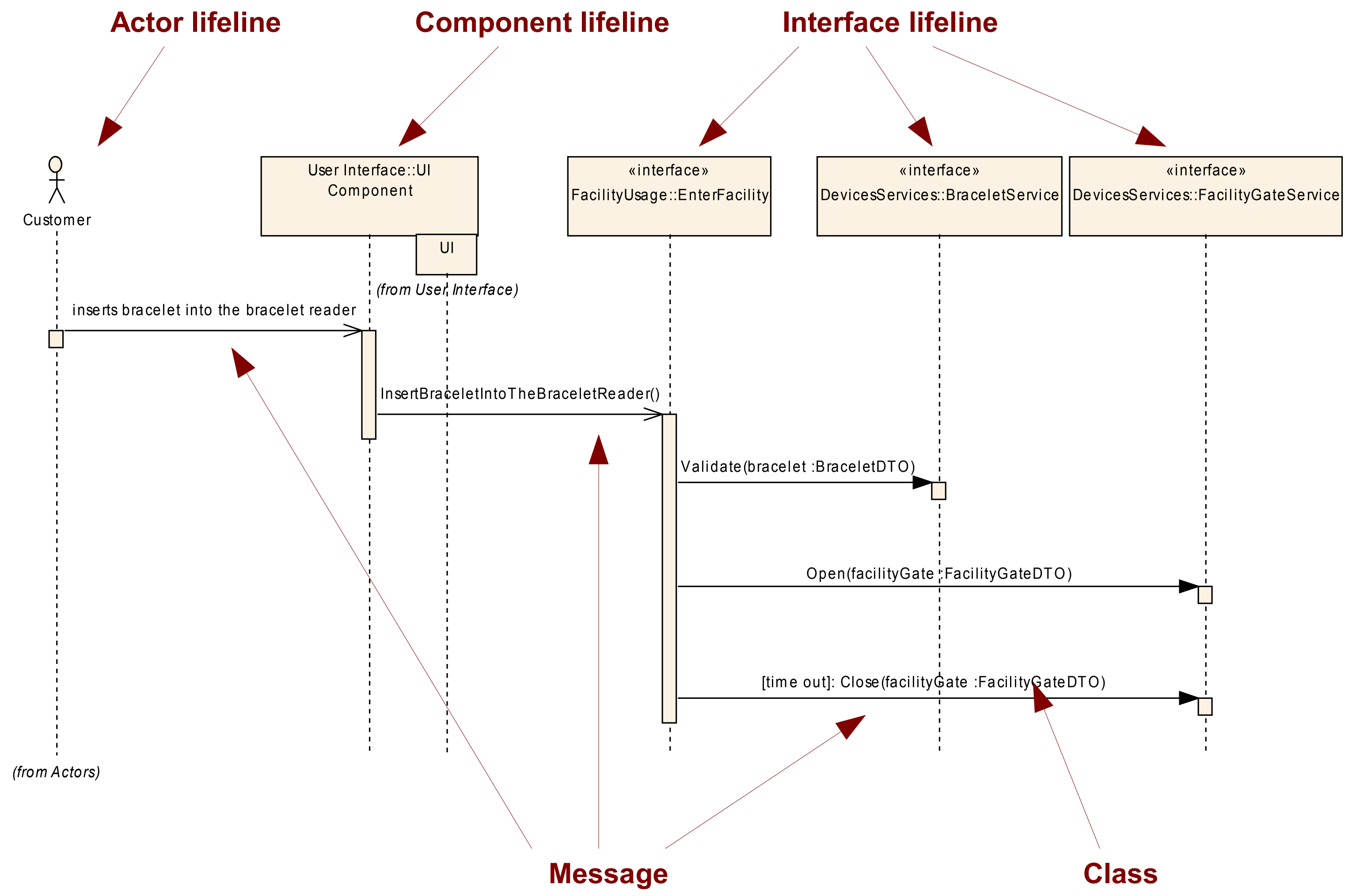

Actor lifeline
Component lifeline
Interface lifeline
Customer
User Interface::UI Component
UI
(from User Interface)
«interface» FacilityUsage::EnterFacility
«interface» DevicesServices::BraceletService
«interface» DevicesServices::FacilityGateService
inserts bracelet into the bracelet reader
InsertBraceletIntoTheBraceletReader()
Validate(bracelet :BraceletDTO)
Open(facilityGate :FacilityGateDTO)
[time out]: Close(facilityGate :FacilityGateDTO)
(from Actors)
Message
Class

Figure 4.4: Example of sequence diagram for 4-layer architecture

4.2.3 UML facilities for Detailed Design model

Detailed design of software system is the lowest level of project specification. It should contain all logic elements - classes and relations between them for each component in an architecture specification. Detailed design model is the basis of implementation in the concrete programming language (e.g. Java, C#). Such models in the UML are presented in class diagrams.
To give the full view of design model on the level of classes, behaviour between the elements in the component should be depicted. The best way to express this behaviour seems to be utilisation of sequence diagrams (subtype of interaction diagram).

Suitable UML elements for a class diagram are:

- Class – "The purpose of a class is to specify a classification of objects and to specify the features that characterize the structure and behavior of those objects". (UML Superstructure [Obj05b], paragraph 7.3.7, page 46)
  In ReDSeeDS detailed design model a class represents an entity of a given component that provides an encapsulated implementation of certain functionality of a given entity. These are exposed by the class to other classes as methods. Apart from business functionality, a class also has properties that reflect unique features of a class. For example, see Figure 4.6.
  Another role of a class is representing data transfer object - boxes containing data with access methods (see Figure 4.5).
- Interface – "An interface declares a set of public features and obligations that constitute a coherent service offered by a classifier." (UML Superstructure [Obj05b], paragraph 7.3.24, page 82)
  In ReDSeeDS detailed design model an interface is a variation of a class. While a class provides an encapsulated implementation of certain business functionality of a componenet, an interface provides only a definition of it. For example, see Figure 4.6.
- DataType – "A data type is a special kind of classifier, similar to a class. It differs from a class in that instances of a data type are identified only by their value." (UML Superstructure [Obj05b], paragraph 7.3.11, page 57). For example of usage of this element in ReDSeeDS detailed design model, see Figure 4.6.

Relations between above elements of class diagram:

- Association – "An association specifies a semantic relationship that can occur between typed instances. It has at least two ends represented by properties, each of which is con-

nected to the type of the end.” (UML Superstructure [Obj05b], paragraph 7.3.3, page 36)
In ReDSeeDS detailed design model associations are used for depicting relationship between two classes - usually implemented as an instance variable in one class. For example, see Figures 4.6 and 4.5. There are some useful types of simple association:

  – Multiplicity – defines the association with size of the collection of related class instances,
  – Directed Association – defines the direction of the association,
  – Reflexive Association – defines the association that points back at the same class.

- Aggregation – In ReDSeeDS detailed design model an aggregation relationship shows that an element contains other classes. For example, see Figures 4.6.

- Composition – “Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it.” (UML Superstructure [Obj05b], paragraph 7.3.7, page 38)
  In ReDSeeDS detailed design model a composition is used to depict a class which is made up of smaller classes. For example, see Figures 4.5.

- Generalisation – “A generalization relates a specific classifier to a more general classifier, and is owned by the specific classifier.” (UML Superstructure [Obj05b], paragraph 7.3.20, page 67)
  In ReDSeeDS detailed design model a generalization is the type of relationship used to define reusable classes. The child classes "inherit" the common functionality defined in the parent class. For example, see Figures 4.5.

- Realization – “Realization is a specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other represents an implementation of the latter (the client).” (UML Superstructure [Obj05b], paragraph 7.3.45, page 124)
  In ReDSeeDS detail design model a relization shows that one entity (an interface) defines a set of functionalities as a contract and the other entity (a class) "realizes" the contract by implementing the functionality defined in the contract. For example, see Figures 4.6.

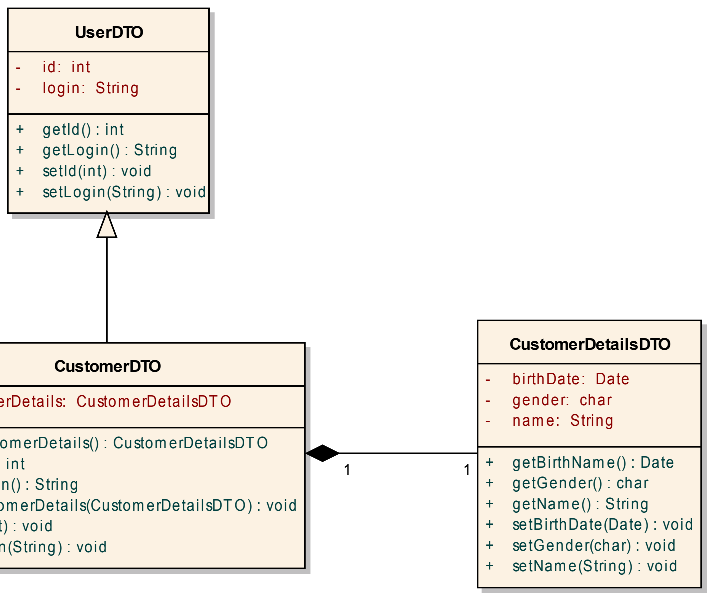


Figure 4.5: Class diagram - example of detailed design

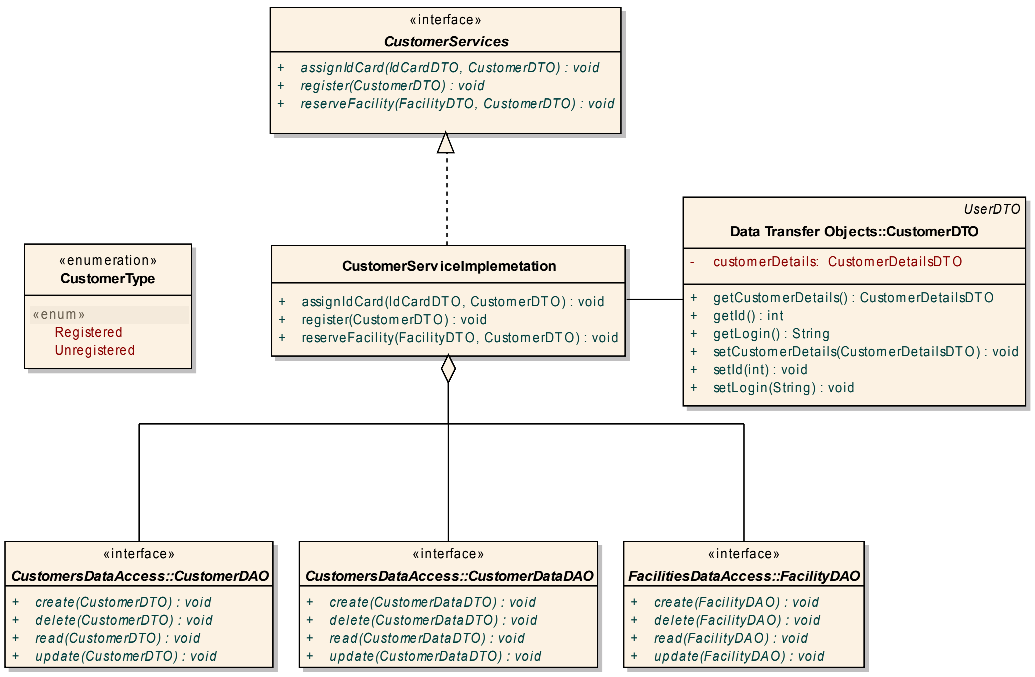

«interface»
CustomerServices
+ assignIdCard(IdCardDTO, CustomerDTO) : void
+ register(CustomerDTO) : void
+ reserveFacility(FacilityDTO, CustomerDTO) : void
«enumeration»
CustomerType
«enum»
Registered
Unregistered
CustomerServiceImplemetation
+ assignIdCard(IdCardDTO, CustomerDTO) : void
+ register(CustomerDTO) : void
+ reserveFacility(FacilityDTO, CustomerDTO) : void
UserDTO
Data Transfer Objects::CustomerDTO
- customerDetails: CustomerDetailsDTO
+ getCustomerDetails() : CustomerDetailsDTO
+ getId() : int
+ getLogin() : String
+ setCustomerDetails(CustomerDetailsDTO) : void
+ setId(int) : void
+ setLogin(String) : void
«interface»
CustomersDataAccess::CustomerDAO
+ create(CustomerDTO) : void
+ delete(CustomerDTO) : void
+ read(CustomerDTO) : void
+ update(CustomerDTO) : void
«interface»
CustomersDataAccess::CustomerDataDAO
+ create(CustomerDataDTO) : void
+ delete(CustomerDataDTO) : void
+ read(CustomerDataDTO) : void
+ update(CustomerDataDTO) : void
«interface»
FacilitiesDataAccess::FacilityDAO
+ create(FacilityDAO) : void
+ delete(FacilityDAO) : void
+ read(FacilityDAO) : void
+ update(FacilityDAO) : void

Figure 4.6: Class diagram - example of detailed design

Dynamics at the level of detail required might be presented as a kind of interaction diagram - sequence diagram. Suitable elements for sequence diagram are:

- Lifeline – "A lifeline represents an individual participant in the Interaction." (UML Superstructure [Obj05b], paragraph 14.3.19, page 475)
  In ReDSeeDS detailed design there are three types of lifelines:
  - actor lifeline – represents a lifeline of an actor which is the initiator of actions for UI component.
  - component lifeline – represents a lifeline of a component, which is the initiator of system actions. Considered system part is interacting with components (other parts of the system).
  - class lifeline - represents a lifeline of instances of classes from class model.
  - interface lifeline – represents a lifeline for an interface occuring in class model.
- Message – "A message defines a particular communication between Lifelines of an Interaction." (UML Superstructure [Obj05b], paragraph 14.3.20, page 477)
  In ReDSeeDS detailed design Message is the same as in the standard UML.
- Combined fragments – "A combined fragment defines an expression of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. Through the use of CombinedFragments the user will be able to describe a number of traces in a compact and concise manner." (UML Superstructure [Obj05b], paragraph 14.3.3, page 453)
  In ReDSeeDS detailed design Combined fragments ("loop" kind) are used for expressing loops in interactions between particular implementation classes.

Sequence diagrams on detailed design level should reflect interaction from architectural sequence diagrams, showing implementation details of components (interaction between realisation classes). Figure 4.7 shows simple example of sequence diagram on detailed design level.

The list of elements and their role presented above should not limit a designer in using the UML in specifying the detailed design. Structure of detailed design and its behaviour specified with another elements will be valuable, but will not be considered under the transformation and similarity measure rules.

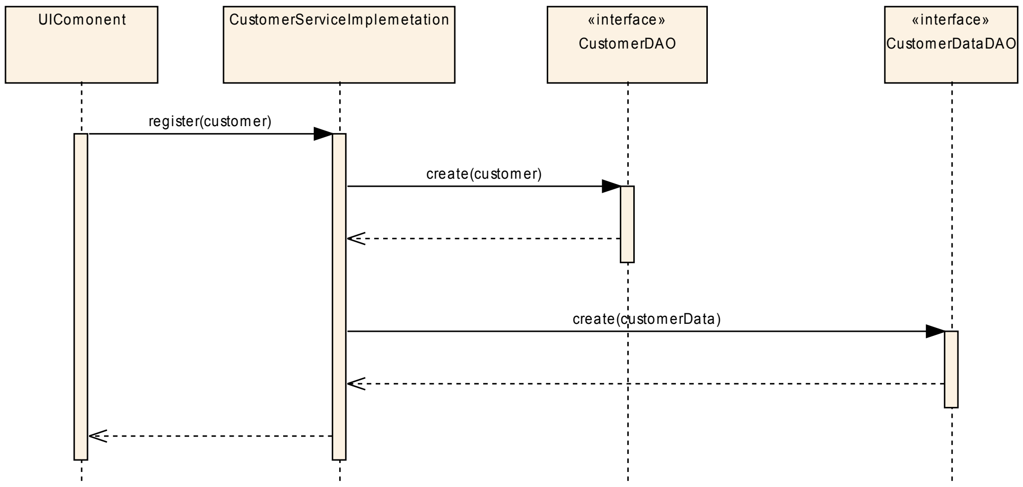


Figure 4.7: Sequence diagram - example of detailed design

4.3 Other possible notations for SDSL

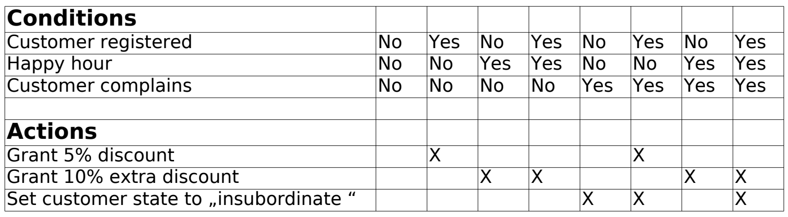
In section 4.1 it was already stated that SDSL is not a new language, but it consists of guidelines for using an existing modelling language. The previous section depicted the usage of UML as SDSL, this chapter gives information about the usage of other languages to describe architecture and detailed design models in SCL. Decision tables are used as a well-known and frequently used example language. For a detailed explanation of decision tables refer to [Pol72].

4.3.1 Decision tables as example

A decision table consists of the four parts conditions, condition alternatives, actions and action entries. The table below shows the structure of a decision table.

Conditions	Condition alternatives
Actions	Action entries

An example for such a condition table is presented in figure 4.3.1. This table describes a part of the policy of a possible fitness club owner on how to deal with customers. The conditions and their values are located in the upper part while the lower part contains the actions and wether they have to be executed. The first row on the left contains the conditions and actions, the other rows contain all combinations of the conditions and an “X” in for every action that should be executed. For example, the third row of these entries depicts, that a a non-registered customer gets 10% extra discount on the price he or she has to pay in the happy hour.



Conditions								
Customer registered	No	Yes	No	Yes	No	Yes	No	Yes
Happy hour	No	No	Yes	Yes	No	No	Yes	Yes
Customer complains	No	No	No	No	Yes	Yes	Yes	Yes
Actions								
Grant 5% discount		X				X		
Grant 10% extra discount			X	X			X	X
Set customer state to „insubordinate “					X	X		X

Figure 4.8: Example for a decision table

4.3.2 Decision-table metamodel

To use such decision tables as a notation for SDSL, a metamodel according to the style guide in deliverable 3.1 is needed. This metamodel could be as simple as the one depicted in figure 4.9 or even more sophisticated.

With this metamodel it is possible to use decision tables as an additional notation for SDSL. To do this, the SCL language must import the package that contains this metamodel. The technical details and the implementation of this step are part of workpackage five, which deals with the implementation of a first RSL tool. Figure 4.10 shows the package structure of SCL with support for decision tables.

To make the new language usable as a notation for SDSL, a tool supporting the language has to be chosen. There is no restriction to the tool except that there must be a way of storing the artifacts created with that tool in a format the ReDSeeDS tool can read. A detailed specification of these formats and the integration process is part of workpackage five (WP5).

4.3.3 Approach for other languages

To integrate any other language into SCL as notation for SDSL, the procedure is exactly the same as the one depicted in this section. A metamodel according to the style guide depicted in deliverable 3.1 is necessary as the basis. Further, a tool to use the new language should be chosen. This tool should be able to store the created artifacts in a language the ReDSeeDS tool can read. If these two steps are finished, the new language can be used as a notation for SDSL.

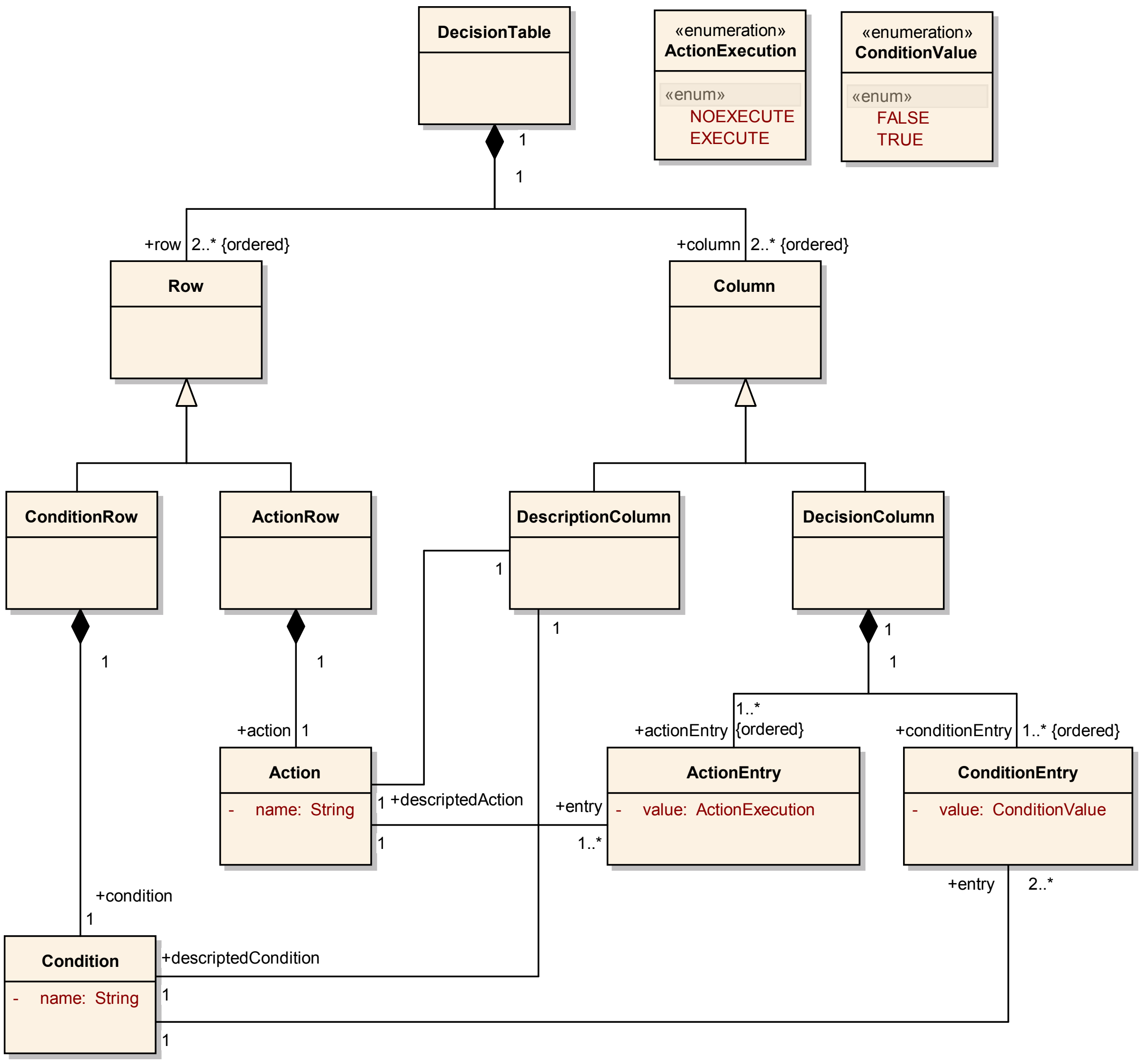


Figure 4.9: Example metamodel for a decision table

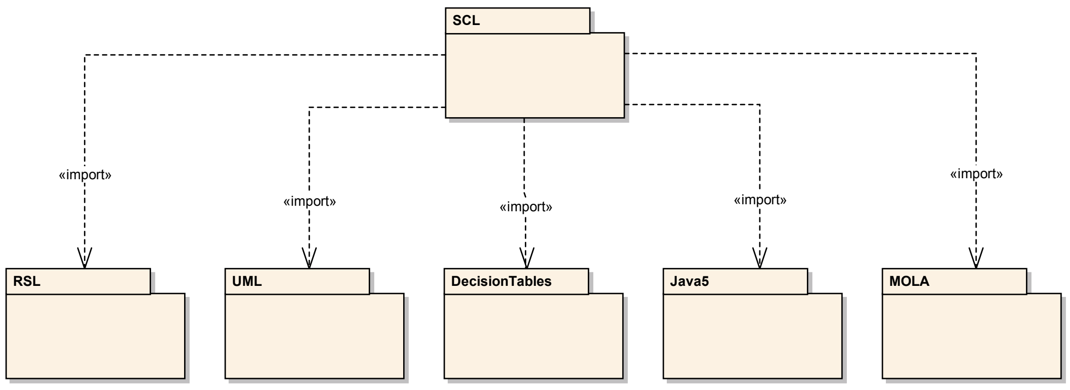


Figure 4.10: Package structure of SCL extended by decision tables

Chapter 5

Transformation Language MOLA

5.1 MOLA overview

In this section an informal introduction into MOLA transformation language is provided. First, the general description of basic MOLA elements is given, with a very informal semantics for the most typical situations. Then a simple MOLA transformation example (not from the ReDSeeDS domain) is described, where a typical usage of these basic elements is shown. Both the description and the example cover only the most typical constructs of MOLA language, the complete description is given in the reference part in the next section.

5.1.1 Basic elements of MOLA

MOLA is a graphical model transformation language, which is used for transforming an instance of a source metamodel (the **source model**) into an instance of the target metamodel (the **target model**). A **transformation** definition in MOLA consists of the **source** and **target metamodel** definitions and one or more MOLA **procedures**.

Source and target metamodels are jointly defined in the MOLA **metamodelling language**, which is quite close to the OMG EMOF specifications (the small differences have already been discussed in the deliverable 3.1, section 7.1). These metamodels are defined by means of one or more class diagrams, packages may be used in a standard way to group the metamodel classes. Actually, the division into the source and target parts of the metamodel is quite semantic, they are not separated syntactically (the complete metamodel may be used in transformation procedures in a uniform way). Typically, additional **mapping** (or traceability) associations link the corresponding classes from source and target metamodels; they facilitate the building of natural transformation procedures and document the performed transformations. The source and target metamodel may be the same - that is the case for in-place model update transformations. The MOLA metamodelling language is defined formally in the **Kernel** package of MOLA metamodel.

MOLA **procedures** form the executable part of a MOLA transformation. One of these procedures is the main one, which starts the whole transformation of the source model. MOLA procedure is built as a traditional structured program, but in a graphical form. Similarly to UML activity diagrams (and conventional flowcharts), **control flow** arrows determine the order of execution. **Call** statements are used to invoke sub-procedures. However, the basic language elements of MOLA procedures are specific to the model transformation domain - they are **rules** and **loops** based on rules. Rules embody the **pattern match** paradigm, which is typical to model transformation languages.

Each rule in MOLA has the **pattern** and **action** part. Both are defined by means of class elements and links. A **class element** is a metamodel class, prefixed by the element ("role") name (graphically shown in a way similar to UML instance). An **association link** connecting two class elements corresponds to an association linking the respective classes in the metamodel. A pattern is a set of class elements and links, which are compatible to the metamodel for this transformation. A pattern may simply be a metamodel fragment, but a more complicated situation is also possible - several class elements may reference the same metamodel class, certainly, their element names must differ (these elements play different roles in the pattern, e.g., the start and end node of an edge). A pattern element may contain also a **constraint** - a Boolean expression in a simplified subset of OCL.

The main semantics of a rule is in its pattern match - an instance set in the model must be found, where an instance of the appropriate class is allocated to each pattern element so that all required links are present in this set and all constraints evaluate to true. If such a match is found, the action part of the rule is executed. The action part also consists of class elements and links, but typically these are **create** actions - the relevant instances and links must be created. An end of a create-link may be attached to a pattern element too. **Assignments** in create-elements may be used to set the attribute values of the instances to be created. Elements may also be deleted and modified in the action part. Thus a rule in MOLA typically is used to locate some construct in the source model and build a required equivalent construct in the target model. If several instance sets in the model satisfy the rule pattern, the rule is, however, executed **only once** (on an arbitrarily chosen match). Such a situation should be addressed by another related construct in MOLA - the loop construct. In addition, the reference mechanism (a pattern element may be a **reference** to an already matched instance in a previous rule) is used to restrict the available match set. Thus, rules are typically used in MOLA in situations, where at most one match is possible. Certainly, there may be a situation when no match exists - then the rule is not executed at all. To distinguish this situation, a rule may have a special **ELSE-exit** (a control flow labelled ELSE), which is traversed namely in this situation. Thus a rule plays in MOLA also the role of an if-then-else construct.

Another essential construct in MOLA is the loop (more concretely, **foreach** loop). This loop is a rectangular frame, which contains one special rule - the **loop head**. The loop head is a rule, which contains one specially marked (by a bold border) element - the **loop variable**. The semantics of a foreach loop is that it is executed for all possible matches for the loop head, which differ by instances allocated to the loop variable (possible variations for other loop head elements are not taken into account). In other words, a foreach loop is an iterator, which iterates through all possible instances of the loop variable class, which satisfy the constraint imposed by the pattern in the loop head. With respect to other elements of the pattern in the loop head, the "existential semantics" is in use - there must be a match for these elements, but it does not matter, whether there is one or several such matches. Thus a foreach loop is the main MOLA construct, which is used to code a situation: "for each instance of ... in the source model which satisfies ... perform the following transformation...". Namely such situations in informal descriptions of model transformations are frequently called transformation rules, but in MOLA they must be formalised as foreach loops.

In addition to the loop head, a loop typically contains the **loop body** - other rules and nested loops, whose execution order is organised by control flows. The loop body is executed for each loop iteration. Since the loop head is a rule, it also may contain create actions, thus simple transformations of source model elements may be coded in MOLA by loops consisting of the loop head only. For nested lops the main organising feature is the possibility to reference the loop variable (and other pattern elements) of the main loop in the pattern of the nested loop head, thus specifying an iteration over all related instances (to the current instance in the main loop).

There is a semantic variation of the foreach loop in MOLA - the non-fixed loop, where the iteration set for the loop variable can be augmented during the loop body execution (in the standard case this set is fixed to that at the loop execution start). The while-loop is also available in MOLA, but it is used less frequently.

There are also other available constructs in MOLA procedures. Procedures may have **parameters** (of type of a metamodel class or a primitive type) and local **variables** (also of both kinds). These elements may be used in MOLA patterns, but in addition **text statements** (consisting of a constraint and assignments) may be used to process these elements more directly. For primitive-typed variables the text statement is the only option. A text statement containing a constraint (a Boolean expression) may also have an ELSE-exit and serve as an if-then-else construct (in addition to rule). Besides MOLA procedures, external (coded in an OOPL) procedures can also be invoked, this feature is used for low-level data processing (e.g., model data import). It should be noted that MOLA has no built-in UI support (MOLA is oriented towards behind-the-scenes transformations), therefore diagnostic messages and similar situations should be addressed via a library of external procedures. All MOLA procedure elements are defined formally in the **MOLA** package of the MOLA metamodel.

The execution of a MOLA transformation on a source model starts from the main procedure. A loop is executed while there are instances to iterate over, then the next construct according to the control flow is executed. If a rule without a valid match is to be executed, and this rule has no ELSE-exit, then the current procedure is terminated (if this occurs outside a loop) or the next iteration of the loop is started (within a loop body). When the main procedure reaches its end, the transformation is completed and the target model is built. If some traceability between these models is required, MOLA procedures simply have to explicitly build the required traceability elements - frequently these are the already mentioned mapping links.

5.1.2 Simple MOLA example

In order to illustrate the basic MOLA concepts, briefly listed in the previous section, a simple MOLA transformation example is provided. It is from an "abstract" MDA area, not from the ReDSeeDS domain, in order to keep it very simple.

Let us assume that we have to build an initial part of the database schema definition - tables and columns from a class diagram. The source model (simple class diagrams) is described by a significantly simplified fragment of Classes package in the UML 2 metamodel. Though only the very basic elements in this source metamodel are retained, still it has the feature that a class attribute is represented by the Property metamodel class, and so are the association ends. Therefore each Property has to be analysed, whether it really represents an attribute. All metamodel classes in this fragment are placed in the Kernel package. The Class metamodel class has one additional tag - the Boolean isPersistent, which is treated in this example as a "normal" attribute.

The target metamodel is even simpler - it contains only two classes Table and Column, both in the SQL package. The association cols expresses the ownership of Column by a Table, the association pkey - that the corresponding Column is a primary key for the Table.

Two mapping associations link the source and target metamodels - classToTable goes from Class to Table and attributeToColumn from Property.

The transformation to be specified is the following - for each persistent class (i.e., Class instance) we have to build a Table and its primary key Column (with a specifically defined name and type String). For each attribute of such a class, whose type is a primitive one, we have to build a Column in the corresponding Table with the same type, but for an attribute with an Enumeration type - a Column with type String. The Column name coincides with the attribute name. Associations in this oversimplified example are not taken into account.

The Fig. 5.1 shows the metamodel for the example - with both the source and target parts in the same class diagram.

The metamodel example shows that metamodels are defined in MOLA in a standard way, by class diagrams, but only EMOF level facilities are permitted. Generalisation is used in a standard way (however, currently only the single specialisation is supported).

The transformation itself consists of two MOLA procedures - Main (which is really the main one) and ProcessAttribute, which is invoked by Main.

Fig 5.2 shows the procedure Main.

The start and end symbols of a MOLA procedure are represented in the same way as in UML activity diagrams. Control flows are drawn by dashed lines. The first element to be executed in this procedure is a foreach loop (a rectangle with bold lines). This loop consists of the sole loop head rule (a rule is visualised by a grey rounded rectangle). The pattern part of this rule (elements with black borders) contains only one class element - the loop variable cl corresponding to the metamodel class Class (loop variables are distinguished from ordinary elements by bold borders). This class element contains also a constraint specifying that the attribute isPersistent must have the value true. Thus, the semantics of this simple loop/rule/pattern is - the loop is executed for every instance of Class in the source model, where isPersistent has the value true. The action part of the rule contains one class element tbl:Table and one link. The class element is of create type (red dashed borders), and it contains one assignment - the value @cl.name (the value of the attribute name in the matched element cl) must be assigned to the attribute name (of the Table instance to be created). The sole link in the rule is of create type too (a red

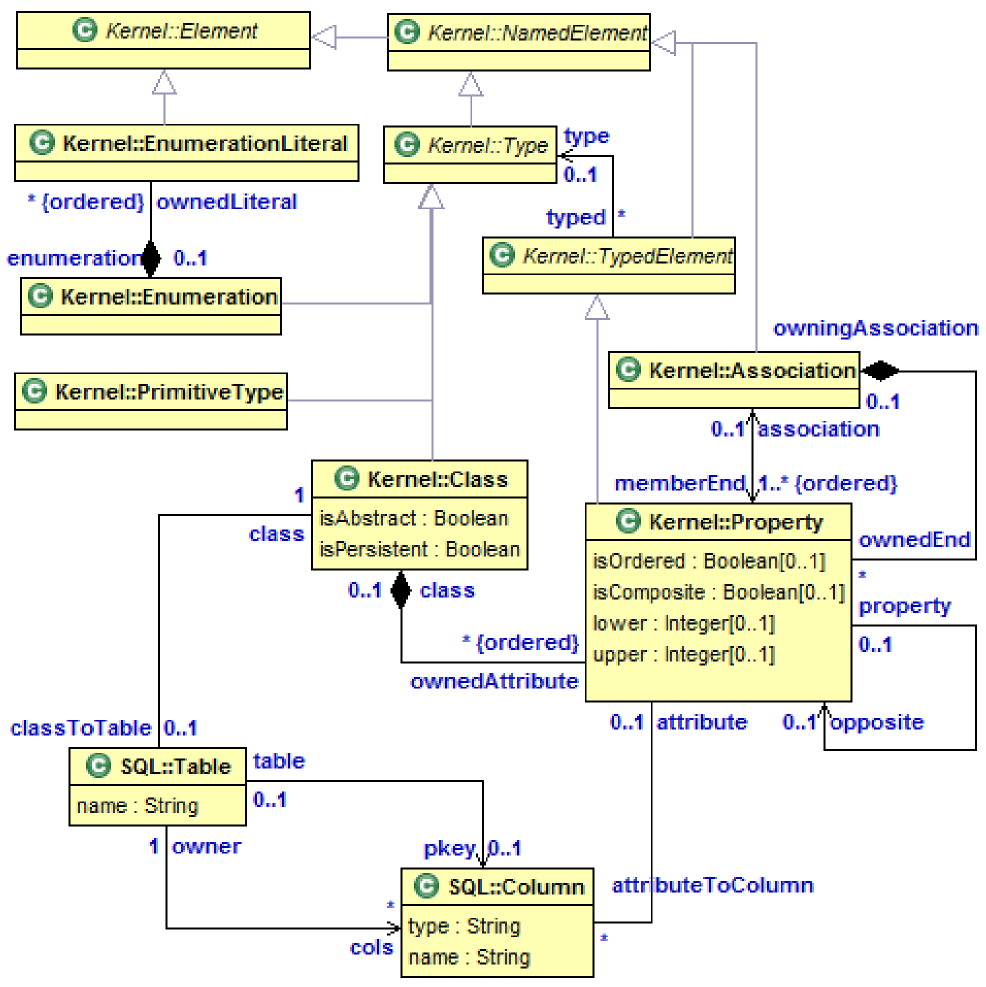


Figure 5.1: The metamodel of the example

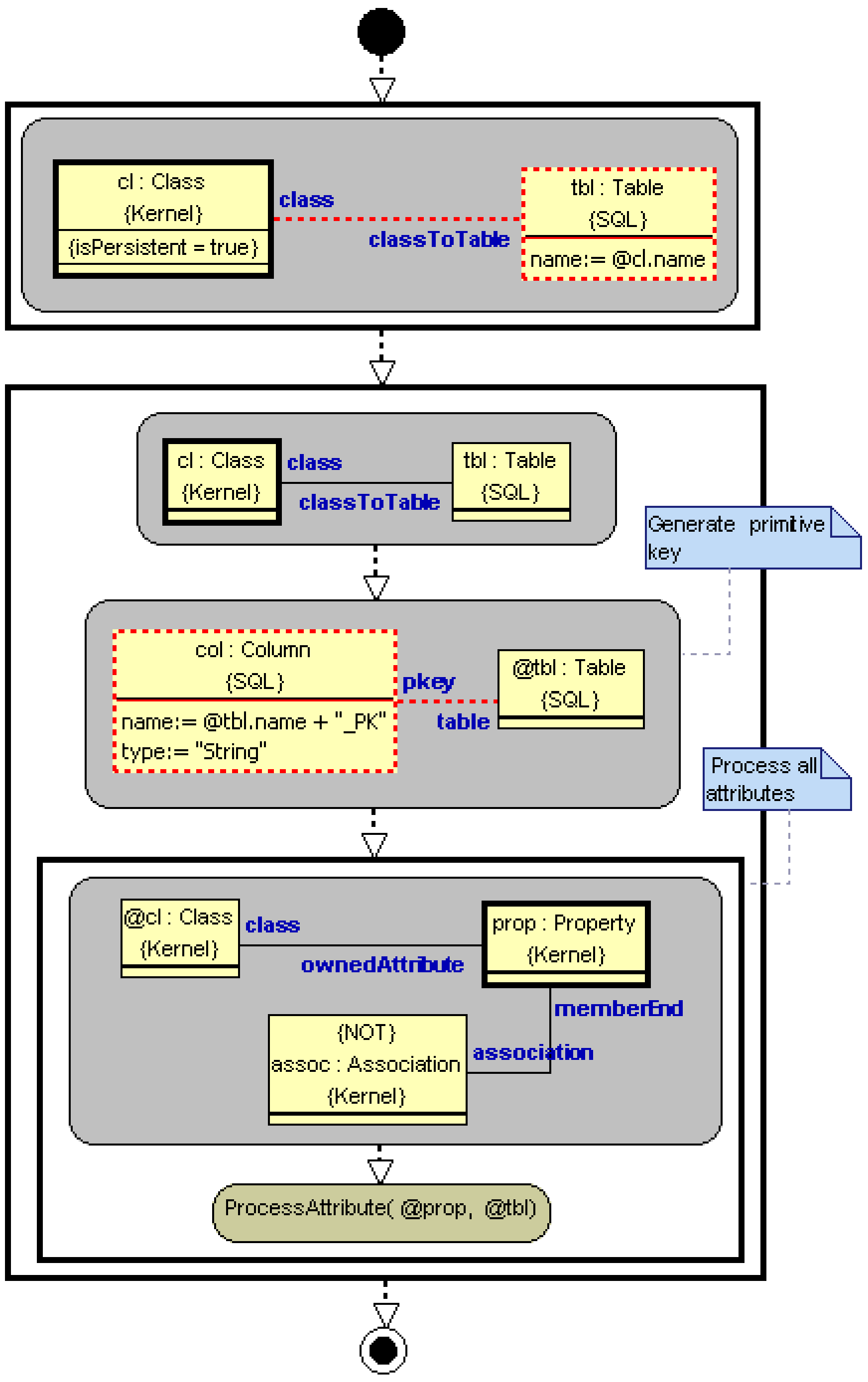


Figure 5.2: The MOLA procedure Main

dashed line) and corresponds to the mapping association in the metamodel (between the Class and Table classes). The correspondence between links in MOLA rules and associations in the metamodel visually is shown via role names, at least one of the role names must be present for a link and UML syntax rules for classes guarantee that a unique specification is possible (the MOLA reference shows that internally a link is directly related to an association). Thus, the first loop is iterated over all persistent Class instances in the source model and for each such instance a new instance of Table is created and its name attribute is set to the same String value as the name of the class. In addition, these two instances are linked by the classToTable link.

This first loop is a typical "design pattern" for simple transformations in MOLA - loop through the instances of a class in the source model and for each valid instance build something in the target model.

The control flow from the first loop leads to the next foreach loop, which again iterates over all classes in the source model (the loop variable is based on Class). However, this time the pattern is more interesting - it contains one more class element (tbl:Table) and one link connecting these elements. The semantics are very natural - only these instances of Class, which have a classToTable link to a Table instance, qualify as valid for an iteration. Since this loop head has no actions, for each iteration immediately the first construct of the loop body - the next rule is executed. It should be noted, that actually the second loop is iterated over literally the same instances as the first loop (persistent classes), since namely for these instances the first loop has built the Table instance and the required link. Therefore in an "optimised program" for this example both loops could be merged in one. The two loops are retained in this example for demo reasons (to demonstrate a pattern for a loop) and because in a more realistic version (where associations also need to be transformed) namely this "two pass" approach can provide a solution.

The next rule in the loop body builds a Column instance (the primary key column), assigns the required values to its attributes and links this new instance to the Table instance located by the loop head. Note the use of **element reference** - @tbl:Table in the rule. The reference construct (an element notation prefixed by the "@" character) says that namely the instance found by a previous rule (here the loop head) must be used. "The previous rule" means the last (according to the execution order) rule, where the referenced element (without the "@" character) was matched in the rule pattern, or created in the action part. If a reference is used in a pattern, it means that no matching is done for this element, simply the known instance is used to build a constraint for other pattern elements, or the instance is used as an end point for the link to be built (this is the given case). The use of the reference as a qualifier for an attribute in an expression has the natural meaning - the attribute value of this instance is taken.

The next construct to be executed in the loop body is a nested loop. It uses the Property class for its loop variable and is meant to loop over the attributes of the current Class instance. The loop head contains a pattern, where the reference @cl:Class says that only the Property instances linked to this known instance must be iterated upon, in addition there must be no Association instance linked to a valid Property (by the association link). The **cardinality constraint NOT** is used in a pattern element to specify that an appropriately linked instance must not exist at all in the model (a NOT-constraint is available also on links in MOLA, but there it says only that a link must not exist). Let us remind that the NOT-constraint is required here to filter these Property instances, which are association ends. The initial part of the loop pattern - the loop variable linked to a reference from the owning loop pattern - is very typical to nested loops in MOLA.

The nested loop in its body has only one construct - the call of the subprocedure ProcessAttribute (which builds the required columns), using references to the known instances prop and tbl as parameters. Certainly, the types (classes) of these parameters must match the parameter definitions in the invoked procedure. Here the classes coincide, but subclass instances may also be supplied (as in OO programming).

This concludes the definition of the Main procedure. When all the relevant iterations are completed, this simple transformation has built the required tables and columns.

It remains to give some comments on the subprocedure ProcessAttribute, which is shown in Fig. 5.3.

The two top symbols in the diagram are parameter definitions (their positions must be numbered, since calls use the positional notation). Parameters can be freely used in patterns, as element references would be.

This MOLA procedure has no loops, since the parameters already provide the exactly required instances. The first rule serves as a typical **if-condition** in an if-then-else construct. It is used to distinguish whether the attribute type is primitive or an enumeration. If the rule pattern matches (the type is primitive), the next rule (followed to via the unlabelled flow) builds the Column instance and sets its attributes. Note that in this rule the reference @pt is legal, since the previous rule has matched and located this instance (it would not be legal to use this reference in the other branch).

If the first pattern fails, the alternative rule (accessed via ELSE-flow) is executed. If its pattern matches, the alternative building rule for the enumeration case is executed. If the second condition fails too (e.g., the attribute type is another class), the external procedure showMsg is

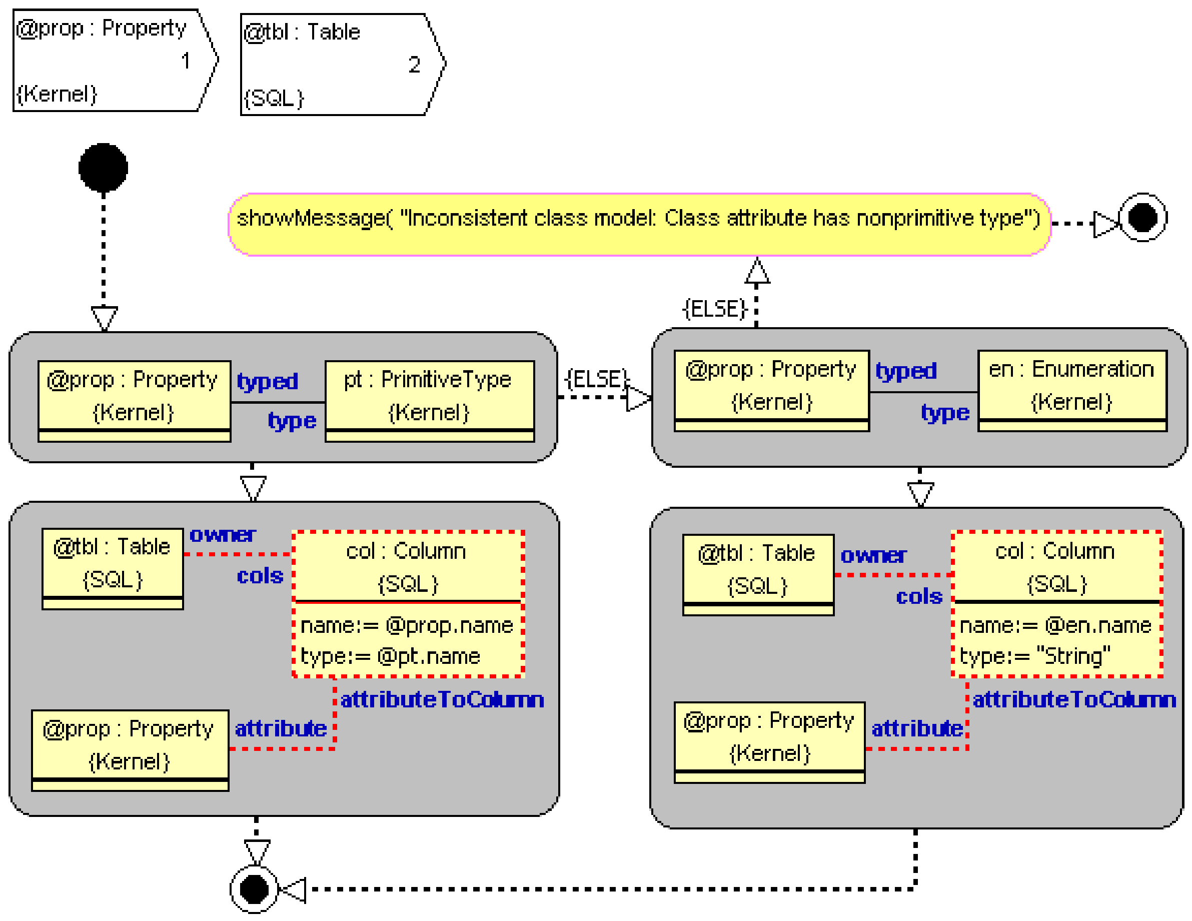


Figure 5.3: The subprocedure ProcessAttribute

invoked. This external procedure actually is built-in within the MOLA environment, it is used to display a simple message box with the provided text.

This completes the comments on the MOLA example.

5.2 MOLA reference

This section describes precise abstract syntax of MOLA and some elements of the semantics. Since MOLA is graphical transformation language which has some textual elements, the abstract syntax for graphical elements is provided via metamodel. The BNF is provided for textual elements.

A transformation in MOLA consists of *a metamodel* (set of class diagrams) and *a set of MOLA procedures* (diagrams similar to activity diagrams)**.** The metamodel describes a model being transformed by the MOLA transformation. The algorithm of the transformation is described by MOLA procedures. Since the metamodel and MOLA procedures are graphical diagrams which contain some textual elements, they have been described using the *MOLA metamodel***.** The MOLA metamodel consists of two packages - Kernel and MOLA. They describe a means for defining a metamodel and MOLA procedures respectively. (Note, that here and further in this document the term *metamodel* without additional qualifiers is used to denote that part of a MOLA transformation)

5.2.1 Structure of the MOLA reference

At first common issues of the MOLA language are discussed. Then descriptions of Kernel and MOLA packages follow.

The description of Kernel and MOLA packages begins with a brief overview and class diagram of the package. The description of the MOLA package includes also descriptions of data types and expression syntax used in MOLA. The class diagram of the MOLA package has been divided into several diagrams to enhance the readability. The precise definition of MOLA package abstract syntax is given by the complete class diagram.

A description of each metaclass follows.

The description has several parts. The first part - a *brief overview* of the metaclass describes the purpose and usage of MOLA elements specified by it.

The second part is the abstract syntax of the metaclass. That includes the following:

- *generalisation*: list of direct superclasses. Actually, there is no multiple generalisation in the MOLA metamodel. Thus there is only one superclass in the list.

- *specialisation*: list of direct subclasses.

- *attributes*: list of attributes owned by this metaclass and brief description of each of them. If the possible values of attributes are described by some grammar, the BNF is presented.

- *associations*: list of associations connected to the metaclass.

Lists of attributes and associations include also attributes and associations that do not belong to the metaclass directly, but are owned by its superclasses. They are called *inherited* in MOLA reference and marked as *[inh]*.

This reference includes also compiler-related attributes (a kind of compiler "directives"). These attributes are marked *[cd]*.

- *constraints*: general constraints on MOLA elements specified by this metaclass. If the constraint is related to attribute or association, then this constraint is included in brief description of attribute or association respectively.

- *notation*: example of the notation of the MOLA element specified by this metaclass. That includes the graphical notation of the MOLA element. It can be a diagram, a graphical or textual diagram element or a project tree node. In fact, this includes an example of the visual representation of the MOLA element specified by this metaclass.

If some features of the metaclass are absent (e.g. no attributes or constraints), then they are not mentioned in the description.

The third part of the description is execution *semantics* of the MOLA elements specified by the metaclass.

The description of metaclasses is followed by a section on MOLA expressions and a section on general issues of semantics and syntax of MOLA.

5.2.2 Naming conventions

Most of elements in MOLA have *identifiers* (names). An identifier can contain letters ('a'-'z', 'A'-'Z'), digits (0-9) and underscore ('_'). No other symbols are allowed. The identifier must start with a letter or underscore. There are some restrictions on naming of several elements of MOLA, which will be explained in sections describing them.

<name>::=<letter>(<letter>|<digit>)*

<digit>::='0'|'1'|. . . |'9'

<letter>::='a'|'b'|. . . |'z'|'A'|'B'|. . . |'Z'|'_'

5.2.3 Kernel package - metamodel definition facilities in MOLA

The Kernel package (see Fig. 5.4) describes means of the metamodel definition in MOLA. The metamodel is defined using the metamodelling language similar to the EMOF definition. This language is called MOLA MOF. In fact, MOLA MOF and EMOF define the same abstract syntax for a metamodel definition language. The main structure of MOLA MOF does not differ from EMOF. A great number of metamodels can be built, which all describe the same set of models. MOLA MOF and EMOF describe the same model (metamodel definition language), but their own metamodels slightly differ. The difference between EMOF and MOLA MOF has been already discussed (see deliverable D3.1 section 7.1).

Element

Brief overview. Element is an abstract metaclass with no superclass. It is used as a common superclass for all metaclasses in the MOLA metamodel.

Specialisations

There are subclasses Comment, NamedElement, Generalization, MOLA::PackageableElement, MOLA::ProcedureElement.

Associations.

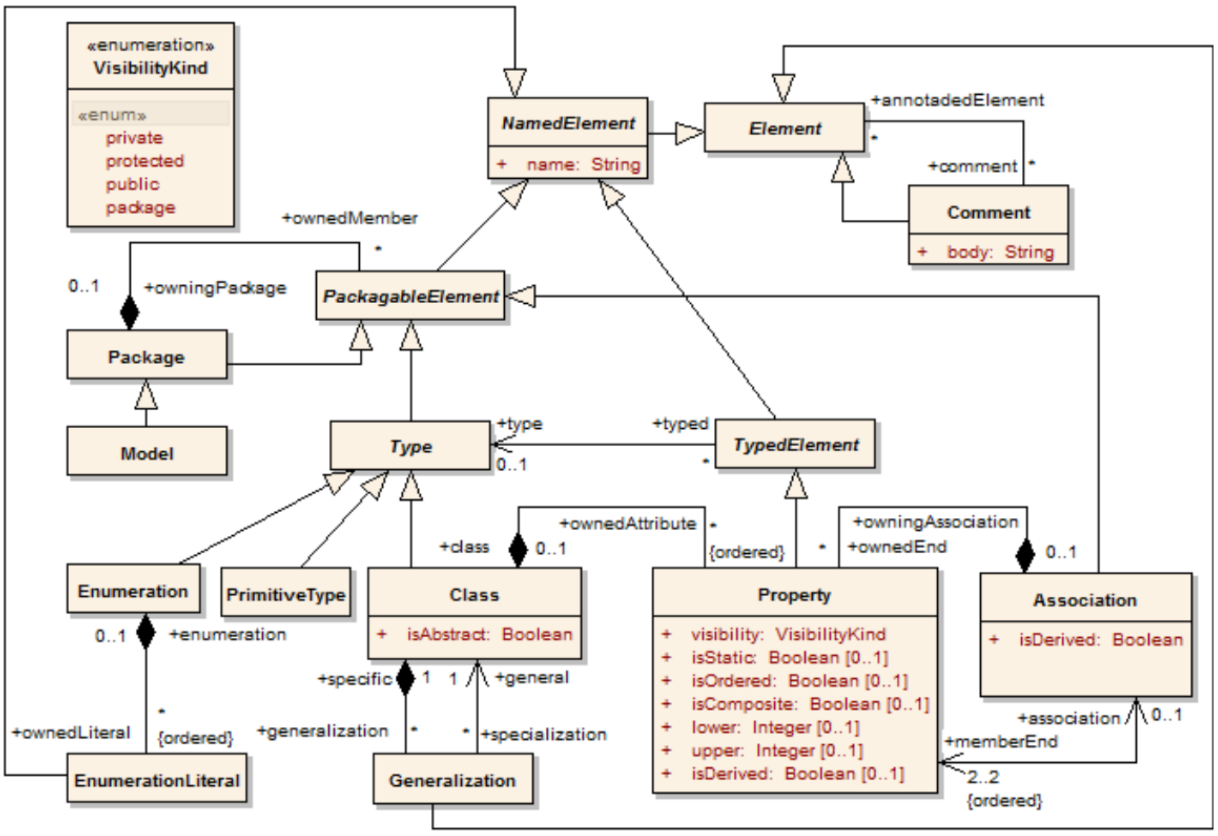


Figure 5.4: Kernel package

comment[*] - comments on this element.

Comment

Brief overview. Comment is an element of a metamodel or MOLA procedure, which provides additional informal information on some elements of a MOLA transformation. It adds no semantic information to the annotated elements, but it can be useful for the transformation writer.

Generalisations

It is specialised from Element

Attributes

body:String - specifies the text of this comment

Associations

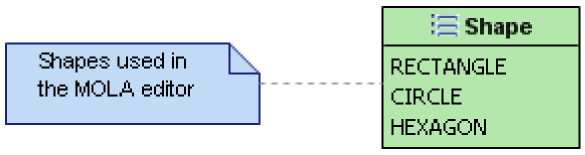


Figure 5.5: Comment and enumeration notation

[inh] comment[*] - comments on this comment.

annotatedElement[*] - elements that is commented by this comment.

Notation. A comment is shown as a rectangle with the upper right corner bent (see Fig. 5.5). The rectangle contains the body of the comment. The connection to each annotated element is shown by a separate dashed line.

NamedElement

Brief overview. NamedElement is an abstract metaclass which represents metamodel elements which have name.

Generalisations

It is specialised from Element.

Specialisations

There are subclasses PackagableElement, TypedElement, EnumerationLiteral.

Attributes

name:String - the name of the named element.

<named-element-name>::=<name>

Associations

[inh] comment[*] - comments on this named element.

PackagableElement

Brief overview. PackagableElement is an abstract metaclass which represents metamodel elements which may be grouped into packages. It is allowed to group only packageable elements. If the element is grouped into package then it is owned by that package. Thus the element may be only in one package.

Generalisations

It is specialised from NamedElement.

Specialisations

There are subclasses Package, Type, Association.

Attributes

[inh] name:String - the name of the packageable element. The name must be unique within the elements grouped into the same package. The name is used to reference this element within the package. To reference a packageable element outside the package the full qualified name must be used. The full qualified name is obtained by prefixing the name with full qualified name of the owning package and '::' symbol.

<full-packageable-element-name>::=[<full-owningpackage-name>'::']<named-element-name>

Associations

[inh] comment[*] - comments on this packageable element.

owningPackage [0..1] - a package that owns this packageable element.

Notation. Packageable element may be shown in the project tree as a node, which has an icon and a name.

Package

Brief overview. Package is an element of the metamodel that is used to group packageable elements. Grouped elements are owned by the grouping package. The package may be owned by other package.

Generalisations

It is specialised from PackagableElement.

Specialisations

There is a subclasse Model.

Attributes

[inh] name:String - the name of the package.

Associations

[inh] comment[*] - comments on this package.

[inh] owningPackage [0..1] - a package that owns this package.

ownedMember[*] - packageable elements that are owned by this package. The package may not contain itself directly or indirectly.

Notation. Package may be shown in the project tree as a node.

Model

Brief overview. Model is a special case of package - the root package - that means the package without owning package. There is exactly one model in a MOLA transformation.

Generalisations

It is specialised from Package.

Attributes

[inh] name:String - the name of the model.

Associations

[inh] comment[*] - comments on this model.

[inh] owningPackage [0..1] - package that owns this model. A model has no owning package.

[inh] ownedMember[*] - packageable elements that are owned by this model.

molaModel[0..1] - MOLA model that conforms to this model (a metamodel)

Notation. Model may be shown in the project tree as a node, which has a special icon and a name.

Type

Brief overview. Type is an abstract metaclass that serves as a constraint on the range of the values represented by typed element of a metamodel and referenceable element of a MOLA procedure.

Generalisations

It is specialised from PackagableElement.

Specialisations

There are subclasses Enumeration, PrimitiveType, Class.

Attributes

[inh] name:String - the name of the type.

<type-name>::=<enumeration-name>|<primitive-type-name>|<class-name>

Associations

[inh] comment[*] - comments on this type.

[inh] owningPackage [0..1] - a package that owns this type.

Typed[*] - typed elements constrained by this type.

molaElem[*] - referenceable elements of a MOLA procedure constrained by this type.

Enumeration

Brief overview. Enumeration is a kind of type, whose instances may be any of a number of user-defined enumeration literals.

Generalisations

It is specialised from Type.

Attributes

[inh] name:String - the name of the enumeration.

<enumeration-name>::=<name>

Associations

[inh] comment[*] - comments on this enumeration.

[inh] owningPackage [0..1] - a package that owns this enumeration.

[inh] Typed[*] - typed elements, actually, class attributes constrained by this enumeration.

[inh] molaElem[*] - referenceable elements, actually, elementary variables of a MOLA procedure constrained by this enumeration.

ownedLiteral[*] - ordered set of literals owned by this enumeration.

Notation. An enumeration is shown as a rectangle which is divided by horizontal line (see Fig. 5.5). The upper compartment contains the name label and a special icon. The lower compartment contains the list of enumeration literals owned by this enumeration. An enumeration may be displayed in the project tree. The enumeration literals may be shown as child nodes of the enumeration node.

EnumerationLiteral

Brief overview. EnumerationLiteral is a user-defined data value for an enumeration.

Generalisations

It is specialised from NamedElement.

Attributes

[inh] name:String - the name of the enumeration literal. The name must be unique within the owning enumeration.

<enumeration-literal-name>::=<name>

Associations

[inh] comment[*] - comments on this enumeration literal.

enumeration[0..1] - an enumeration that owns this enumeration literal.

Notation. An enumeration literal is typically shown as a name, one to a line, in the lower compartment of the enumeration symbol (see Fig. 5.5). An enumeration literal may be shown as child node of the enumeration node in the project tree.

Semantics. Enumeration literals may be used as enumeration constants. Note that enumeration constants are without the prefixed enumeration name. Mainly they are used to set enumeration-typed attributes or elementary variables. Thus, the precise enumeration literal type can be obtained from the context of the enumeration constant usage (attribute type, variable type, etc. . . .).

<enumeration-constant>::=<enumeration-literal-name>

PrimitiveType

Brief overview. PrimitiveType defines a predefined data type, without any relevant substructure (i.e., it has no parts). There are three predefined primitive types in MOLA: String, Integer and Boolean. There are no other primitive types and there is no way to define them in MOLA.

Generalisations

It is specialised from Type.

Attributes

[inh] name:String - the name of the primitive type.

<primitive-type-name>::='String'|'Integer'|'Boolean'

Associations

[inh] comment[*] - comments on this primitive type.

[inh] owningPackage [0..1] - a package that owns this primitive type. A primitive type is owned by the model.

[inh] Typed[*] - attributes constrained by this primitive type.

[inh] molaElem[*] - referenceable elements, actually, elementary variables of a MOLA procedure constrained by this primitive type.

Semantics. Primitive types are predefined data types in MOLA. Values of the String type are sequences of characters. The set of characters allowed in MOLA is unspecified. String constants are specified using the value within single or double quotes. Values of the Integer type are whole numbers. There is no special notation for integer constants - a constant is specified using the value. Values of the Boolean type are *true* and *false*. The *true* and *false* keywords serve also as Boolean constants.

<string-constant>::=’”(<any-character>)*”’ | ’”’(<any-character>)*’”’

<boolean-constant>::=true | false

<integer-constant>::=<whole-number>

Class

Brief overview. Class specifies a classification of objects and the attributes that characterize the structure of those objects. A class is a kind of type which describes a set of objects that have the same attributes, constraints and semantics. Classes may have attributes, connected associations and form class hierarchy.

Generalisations

It is specialised from Type.

Attributes

[inh] name:String - the name of the class.

<class-name>::=<name>

isAbstract:Boolean - indicates that class is abstract.

Associations

[inh] comment[*] - comments on this class.

[inh] owningPackage [0..1] - a package that owns this class.

[inh] Typed[*] - association ends that are connected to this class.

[inh] molaElem[*] - referenceable elements, actually, pointers of a MOLA procedure constrained by this class.

specialization[*] - Generalization instances where this class is a superclass.

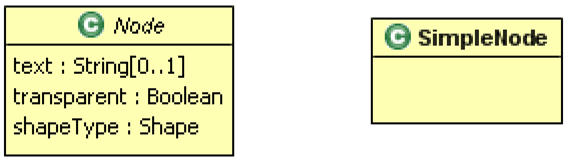


Figure 5.6: Class and attribute notation

generalization[*] - Generalization instances where this class is a subclass. Currently only single generalization for a class is permitted in MOLA.

ownedAttribute[*] - attributes and navigable association ends owned by the class.

Notation. A class is shown as a rectangle which is divided by horizontal line (see Fig. 5.6). The upper compartment contains the name label and a special icon. The lower compartment contains the list of attributes owned directly by the class. An item of the list consists of attribute name, type and multiplicity of property if it is other than 1

It is recommended:

- To put the class name in italics if the class is abstract, otherwise boldface

- Capitalize the first letter of class names.

- Begin attribute names with a lowercase letter.

A class may be displayed in the project tree as a node with a special icon. Attributes may be shown as child nodes of the class node.

Generalization

Brief overview. Generalization is a relationship between a more general class and a more specific class. Each instance of the specific class is also an indirect instance of the general class. Thus, the specific class inherits the properties of the more general class. Circular generalisations are not permitted.

Generalisations

It is specialised from Element.

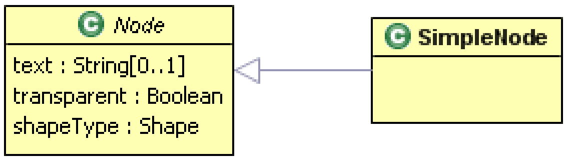


Figure 5.7: Generalization notation

Associations

[inh] comment[*] - comments on this generalization.

general[1] - References the general class in the generalization relationship

specific[1] - References the specializing class in the generalization relationship

Notation. Generalization is shown as line with a hollow triangle as an arrowhead between the symbols representing the involved classes (See Fig. 5.7). The arrowhead points to the symbol representing the general class.

TypedElement

Brief overview. TypedElement is an abstract metaclass which represents metamodel elements which have a type that serves as a constraint on the range of values the typed element can represent.

Generalisations

It is specialised from NamedElement.

Specialisations

There is a subclass Property.

Attributes

[inh] name:String - the name of the typed element.

Associations

[inh] comment[*] - comments on this typed element.

type[0..1] - type which constrains the possible values of typed element.

Property

Brief overview. Property is structural element that characterizes the structure of objects classified by the same class. There are two types of properties - attributes and association ends. Attributes are primitive or enumeration typed in MOLA. They describe properties of the class that can be expressed by value. An association end connects a class to an association that describes a relationship of this class to another class (or the same class)

Generalisations

It is specialised from TypedElement.

Attributes

[inh] name:String - the name of the property. The name must be unique within scope of properties determined by the owning class. More precisely, the name must be unique within the set of all attributes and association ends belonging to this class and all superclasses of it. In MOLA this set includes also non-navigable association ends. No property redefinition is permitted in MOLA. The name of an association end is called *role name* in MOLA.

<property-name>::=<atribute-name> | <role-name>

<atribute-name>::=<name>

<role-name>::=<name>

isOrdered:Boolean[0..1] - indicates that this property is ordered.Only an association end may be ordered in MOLA (it makes sense only for the '*' multiplicity). Both association ends that are members of the same association may not be ordered at the same time. The ordering of an association end specifies that links corresponding to this association are ordered with respect to the instance at the opposite end of these links (it is relevant only for a set of links starting from a common instance). The ordering is taken into account by loops and actions in MOLA.

isComposite:Boolean[0..1] - indicates that this property is composite. Only association ends have this attribute set. That means that the association whose member is the association end, is a composition. The class at the opposite end of the association is part of the class at the composite association end. Both ends of an association can not be composite.

lower:Integer[0..1] - specifies the lower bound of multiplicity interval for this property. Values may be 0 or 1 in MOLA.

upper:Integer[0..1] - specifies the upper bound of multiplicity interval for this property. Upper must be greater or equal than lower. Values may be 1 or '*' (this value is internally coded as -1).The upper bound of multiplicity interval may only be 1 for an attribute.

Other attributes of Property actually are not used in MOLA.

Associations

[inh] comment[*] - comments on this property.

[inh] Type[0..1] - the type which constrains the possible values of the attribute or the class connected by the association end. The type is a primitive type or an enumeration if the property is an attribute. The type is a class if the property is an association end.

class[0..1] - the class that owns this attribute or navigable association end.

owningAssociation[0..1] - the association that owns this non-navigable association end.

association[0..1] - the association whose member is this association end.

assocLinkStart[*] - MOLA association links whose source ends correspond to this association end.

assocLinkEnd[*] - MOLA association links whose target ends correspond to this association end.

Notation. This section provides the notation of properties that are attributes. For notation of association ends see Association 5.2.3. The attribute is shown as string that consists of name, the colon (':'), type name and multiplicity interval string (See Fig. 5.6).

<attr-multiplicity-interval-string>::='0..1' | '1'

Multiplicity interval string may be omitted for '1' (this is the default value).

<attribute-notation>::=

<attribute-name>' : '<type-name>'['<attr-multiplicity-interval-string>']'

Association

Brief overview. Association specifies a semantic relationship that can occur between two classified instances.

Generalisations

It is specialised from PackagableElement.

Attributes

[inh] name:String - the name of the association.

Associations

[inh] comment[*] - comments on this association.

[inh] owningPackage [0..1] - the package that owns this association.

ownedEnd[*] - non-navigable association ends of the association.

memberEnd[2..2] - properties that serves as association ends.

assocLink[*] - MOLA association links which are constrained by this association.

Notation. An association is normally drawn as a solid line connecting two classes, or a solid line connecting a single class to itself (the two ends are distinct) (See Fig. 5.8). A line may consist of one or more connected segments. The individual segments of the line itself have no semantic significance.

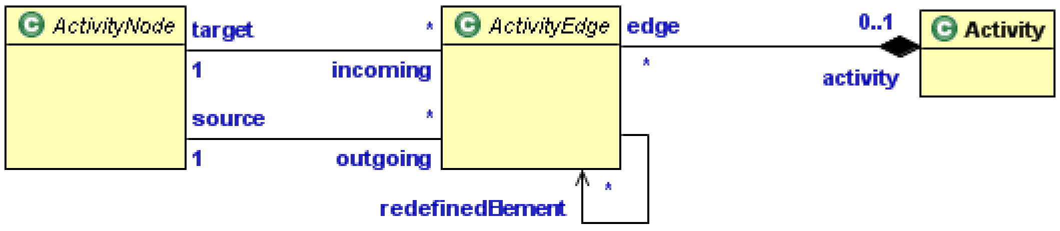


Figure 5.8: Association notation

An association end is the connection between the line depicting an association and the rectangle depicting the connected class. The name of an association end may be placed near the end of the line. The multiplicity string may be placed at the opposite side of the line. If the association end is composite the connection of the line is shown as a filled diamond. If the association end is ordered then the symbol {ordered} is shown near it. If the association end is navigable then it is shown as an open arrow. The navigability is used only to be compatible with the standard notation in MOF. It has no special semantics for MOLA.

5.2.4 MOLA package - transformation definition facilities in MOLA

The MOLA package (see Fig. 5.9) describes facilities for the procedure definition in MOLA. The executable part of a MOLA transformation consists of MOLA procedures (the procedures written in MOLA). They may be grouped into packages. A MOLA procedure is a diagram similar to the activity diagram. A MOLA procedure contains statements that are executed when the transformation is started. The execution order is determined by control flows. There are call statements, loops, rules and text statements in a MOLA procedure. The MOLA procedure contains also start and end points - statements that denote the entry point and termination points of the MOLA procedure. The rule and the foreach-loop are the most typical statements of the MOLA language. Each rule in MOLA has the pattern and action part. Both are defined using class elements and association links. There are also textual elements that specify constraints and assignments in the rule.

The fragment of the MOLA metamodel that describes the means of procedure grouping is shown in 5.10. The executable part of a MOLA transformation consists of procedures. Procedures may be grouped using packages.

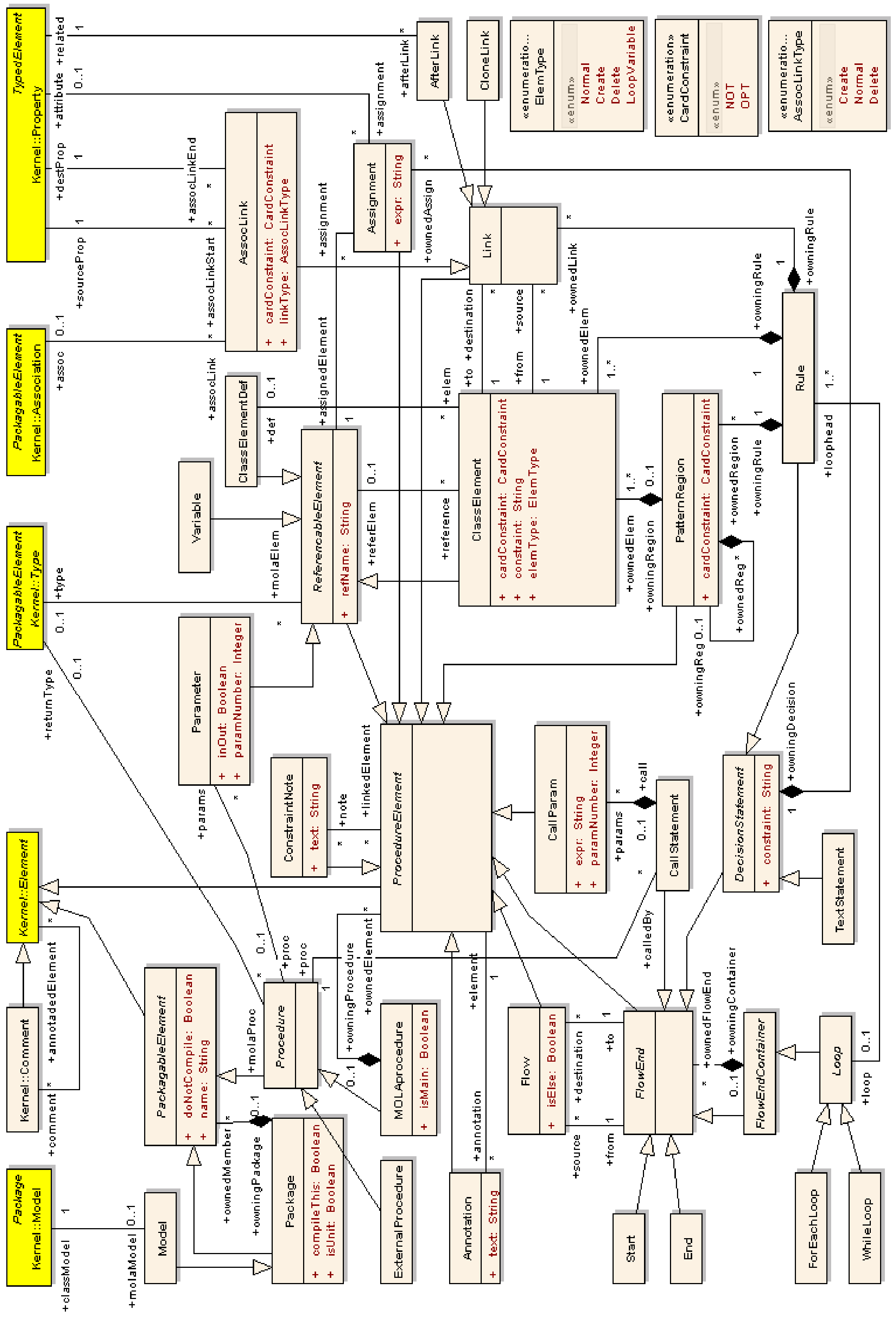


Figure 5.9: MOLA package

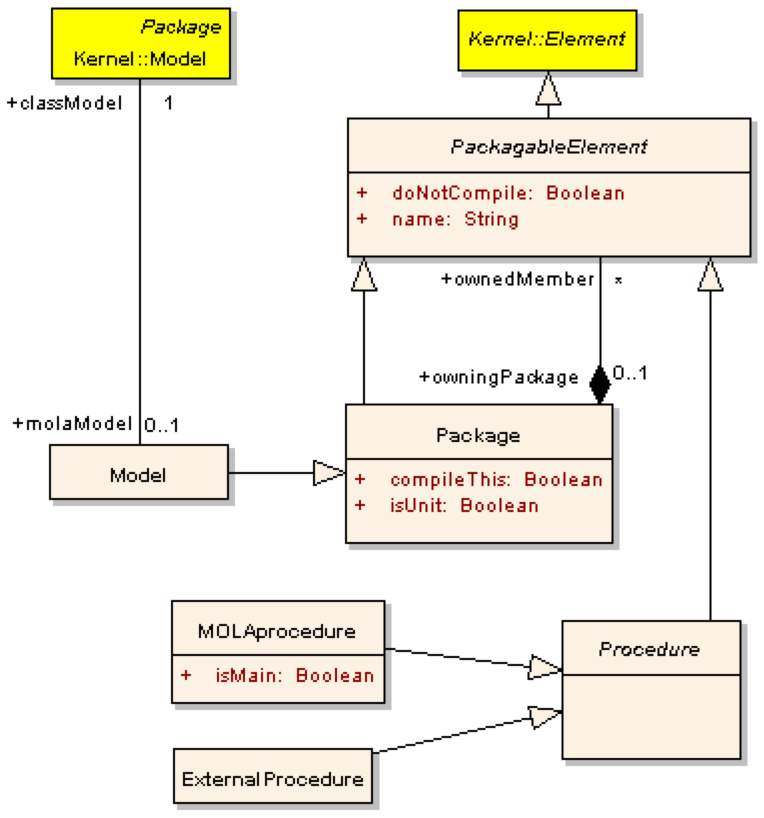


Figure 5.10: Procedures and packages in MOLA

PackagableElement

Brief overview. PackagableElement is an abstract metaclass which represents MOLA elements which may be grouped into packages. If the element is grouped into a package then it is owned by that package.

Generalisations

It is specialised from Kernel::Element.

Specialisations

There are subclasses Package, Procedure.

Attributes

name:String - the name of the packageable element. The name must be unique within the elements grouped into the package. The name is used to reference this element within the package. To reference a packageable element outside the package the full qualified name must be used. The full qualified name is obtained by prefixing the name with full qualified name of the owning package and '::' symbol.

<full-packageable-element-name>::=

[<full-owningpackage-name>'::']<name>

[cd] doNotCompile:Boolean - indicates that the packageable element must be excluded from the compilation scope.

Associations

[inh] comment[*] - comments on this packageable element.

owningPackage [0..1] - a package that owns this packageable element.

Notation. Packageable element may be shown in the project tree as a node, which has an icon and a name.

Package

Brief overview. Package is a MOLA element that is used to group packageable elements. Grouped elements are owned by the grouping package. The package may be owned by other packages. A package may serve as a compilation unit.

Generalisations

It is specialised from PackagableElement.

Specialisations

There is a subclass Model.

Attributes

[inh] name:String - the name of the package.

[inh][cd] doNotCompile:Boolean - indicates that the package must be excluded from the compilation scope.

*[cd]*isUnit:Boolean - indicates that the package is a compilation unit. A compilation unit includes a set of MOLA procedures which all are recompiled if one of them has been changed. The compilation unit includes all MOLA procedures that are owned directly or indirectly by owned packages which are not compilation units.

*[cd]*compileThis:Boolean - indicates that the unit must be compiled. This attribute may be set only if the package is a unit (isUnit=true). If the doNotCompile has been set to *true* then this unit is not compiled.

Associations

[inh] comment[*] - comments on this package.

[inh] owningPackage [0..1] - the package that owns this package.

ownedMember[*] - packageable elements that are owned by this package. The package may not contain itself directly or indirectly.

Notation. Package may be shown in the project tree as a node, which has a special icon and a name. All owned elements may be shown as child nodes of the package node.

Model

Brief overview. Model is a special case of package - the root package - that means the package without owning package. There is exactly one model in a MOLA transformation. The model serves as a default compilation unit.

Generalisations

It is specialised from Package.

Attributes

[inh] name:String - the name of the model.

[inh][cd] doNotCompile:Boolean - indicates that the model will be excluded from compilation scope. By setting this attribute to *true* nothing will be compiled.

*[inh][cd]*isUnit:Boolean -indicates that the package is a compilation unit. A compilation unit includes a set of MOLA procedures which all are recompiled if one of them has been changed. The compilation unit includes all MOLA procedures that are owned directly or indirectly by owned packages which are not compilation units. The model is compilation unit. Thus, isUnit = truefor a model.

*[inh][cd]*compileThis:Boolean - indicates that the unit must be compiled.

Associations

[inh] comment[*] - comments on this model.

[inh] owningPackage [0..1] - a package that owns this model. There is no owning package for a model.

[inh] ownedMember[*] - packageable elements that are owned by this model.

classModel[1] - Kernel::model which specifies metamodel that describes models to be transformed by MOLA procedures.

Notation. Model may be shown in the project tree as a node, which has the special icon and a name.

Procedure

Brief overview. Procedure is an abstract metaclass which represents procedures in MOLA. There are two types of procedures - MOLA procedure and external procedure. A procedure may have parameters. A procedure may be called from a MOLA procedure.

Generalisations

It is specialised from PackageableElement.

Specialisations

There are subclasses MOLAprocedure, ExternalProcedure.

Attributes

*[inh]*name:String - the name of the procedure.

[inh][cd] doNotCompile:Boolean - indicates that this procedure must be excluded from the compilation scope. Call statements that call this procedure are compiled as empty statements doing nothing.

Associations

[inh] comment[*] - comments on this procedure.

*[inh]*owningPackage [0..1] - the package that owns this procedure.

params[*] - parameters of this procedure.

calledBy[*] - call statements that are calling this procedure.

Semantics. A procedure contains statements written in MOLA or represents an external procedure written in any programming languge. A procedure may be *called* from a MOLA procedure. When the execution of the procedure is ended then the control is passed back to the calling procedure.

MOLAprocedure

Brief overview. MOLAprocedure is a procedure which is implemented using MOLA transformation language. One MOLA procedure in a MOLA model is the main procedure.

Generalisations

It is specialised from Procedure.

Attributes

[inh] name:String - the name of the MOLA procedure.

[inh][cd] doNotCompile:Boolean - indicates that the MOLA procedure must be excluded from the compilation scope.

isMain:Boolean - indicates that this MOLA procedure is the main procedure of a MOLA transformation. This procedure is called when the transformation is started. The execution of the transformation ends, when the execution of the main procedure has ended.

Associations

[inh] comment[*] - comments on this packageable MOLA procedure.

[inh] owningPackage [0..1] - the package that owns this MOLA procedure.

[inh] params[*] - parameters of this procedure. These links are not used, since parameters of a MOLA procedure are owned elements of the MOLA procedure and can be obtained via ownedElement association.

[inh] calledBy[*] - call statements that are calling this MOLA procedure.

ownedElement[*] - MOLA procedure elements owned by this MOLA procedure

Notation. A MOLA procedure is a diagram. Facilities used in a MOLA diagram are described in next subsections. MOLA diagram may be shown also as a node in the project tree.

Semantics. When a procedure is called MOLA *statements* are executed.

ExternalProcedure

Brief overview. ExternalProcedure is a procedure which is implemented using another programming language than MOLA (currently C++).

Generalisations

It is specialised from Procedure

Attributes

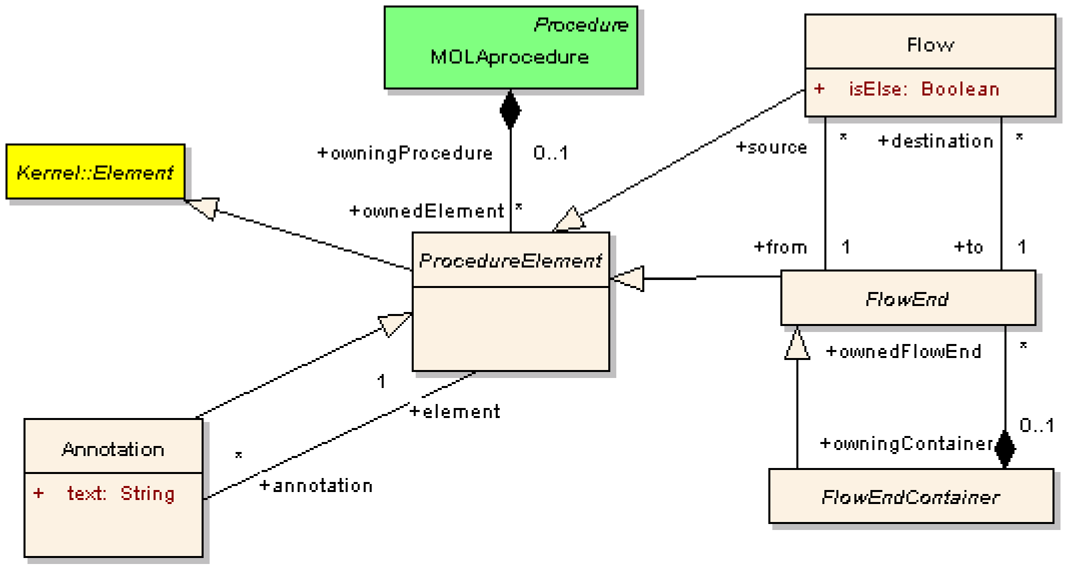


Figure 5.11: Control flows in MOLA

[inh] name:String - the name of the external procedure.

[inh][cd] doNotCompile:Boolean - indicates that the external procedure must be excluded from the compilation scope.

Associations

[inh] comment[*] - comments on this external procedure.

[inh] owningPackage [0..1] - the package that owns this external procedure.

[inh] params[*] - parameters of this external procedure

[inh] calledBy[*] - call statements that are calling this procedure.

Notation. An external procedure is shown in the project tree.

The fragment of the MOLA metamodel that describes the facilities for handling control inside the MOLA procedure is shown in Fig. 5.11. The MOLA procedure consists of statements and control flows.

ProcedureElement

Brief overview. ProcedureElement is an abstract superclass for all classes that describe elements of the MOLA procedure.

Generalisations

It is specialised from Kernel::Element

Specialisations

There are subclasses Annotation, ConstraintNote, Flow, FlowEnd, CallParam, ReferencableElement, Link, Assignment.

Associations.

*[inh]*comment[*] - comments on this procedure element.

owningProcedure[0..1] - the MOLA procedure which owns this procedure element. All procedure elements must have the owning MOLA procedure. The only exception is Parameter (for external procedures).

annotation[*] - annotations added to this procedure element.

Annotation

Brief overview. Annotation is a textual element of the MOLA procedure which adds additional semantic information on an element of the MOLA procedure.

Generalisations

It is specialised from ProcedureElement

Attributes

text:String - the text of the annotation.

Associations.

*[inh]*comment[*] - comments on this annotation.

*[inh]*owningProcedure[1] - the MOLA procedure which owns this annotation.

element[1] - the procedure element this annotation is linked to.

Notation. The notation of an annotation is dependent of an element the annotation is added to.

Semantics. The semantics is dependent of an element the annotation is linked to.

FlowEnd

Brief overview. FlowEnd is an abstract metaclass which represents statements (executable elements) of the MOLA procedure. A statement may be a rule, loop, call, procedure start, procedure end or textual decision statement. The control of the procedure execution is passed to the next statement via control flow. A statement may be owned by another statement - a container.

Generalisations

It is specialised from ProcedureElement

Specialisations

There are subclasses Start, End, FlowEndContainer, CallStatement, DecisionStatement.

Associations.

*[inh]*comment[*] - comments on this statement.

*[inh]*owningProcedure[0..1] - MOLA procedure which owns this statement.

source[*] - outgoing control flows.

destination[*] - incoming control flows

owningContainer[0..1] - the statement container which owns this statement, if any.

Notation. A statement is shown as a node in the control flow graph of the MOLA procedure.

Semantics. A statement is an executable element of the MOLA procedure. A statement is executed accordingly to the semantics of the concrete statement type.

Flow

Brief overview. Flows - control flows - determine the execution order of statements in the MOLA procedure. A control flow is an arrow which connects two statements. A control flow may be marked as an alternate (*ELSE*) control flow.

Generalisations

It is specialised from ProcedureElement

Attributes

isElse:Boolean - indicates that the control flow is an alternate (*ELSE*) control flow.

Associations.

*[inh]*comment[*] - comments on this flow.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this procedure element.

from[1] - the statement the control flow is going from. The flow is outgoing with respect to this statement.

to[1] - the statement the control flow is going to. The flow is incoming with respect to this statement.

Notation. Control flow is shown as a dashed line with a hollow triangle as an arrowhead between the symbols representing involved statements (see Fig. 5.12). The arrowhead points to the symbol representing the statement the control flow is going to. If the control flow is alter-

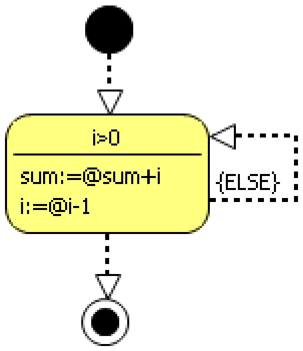


Figure 5.12: Control flow notation

nate (*ELSE*) control flow then a label '{ELSE}' is shown above the line near the statement the control flow is going from.

Semantics. Control flow connects exactly two statements. A statement may have any number of incoming control flows and no more than two outgoing control flows. If there are two outgoing flows then one of them must be an alternate (*ELSE*) control flow. Exact number of incoming and outgoing control flows is determined by the precise type of the statement. The way the control flow that determines the next statement is chosen depends also on the statement type.

FlowEndContainer

Brief overview. FlowEndContainer is an abstract metaclass which represents statements that may contain other statements.

Generalisations

It is specialised from FlowEnd

Specialisations

There is a subclass Loop.

Associations.

*[inh]*comment[*] - comments on this statement.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this container

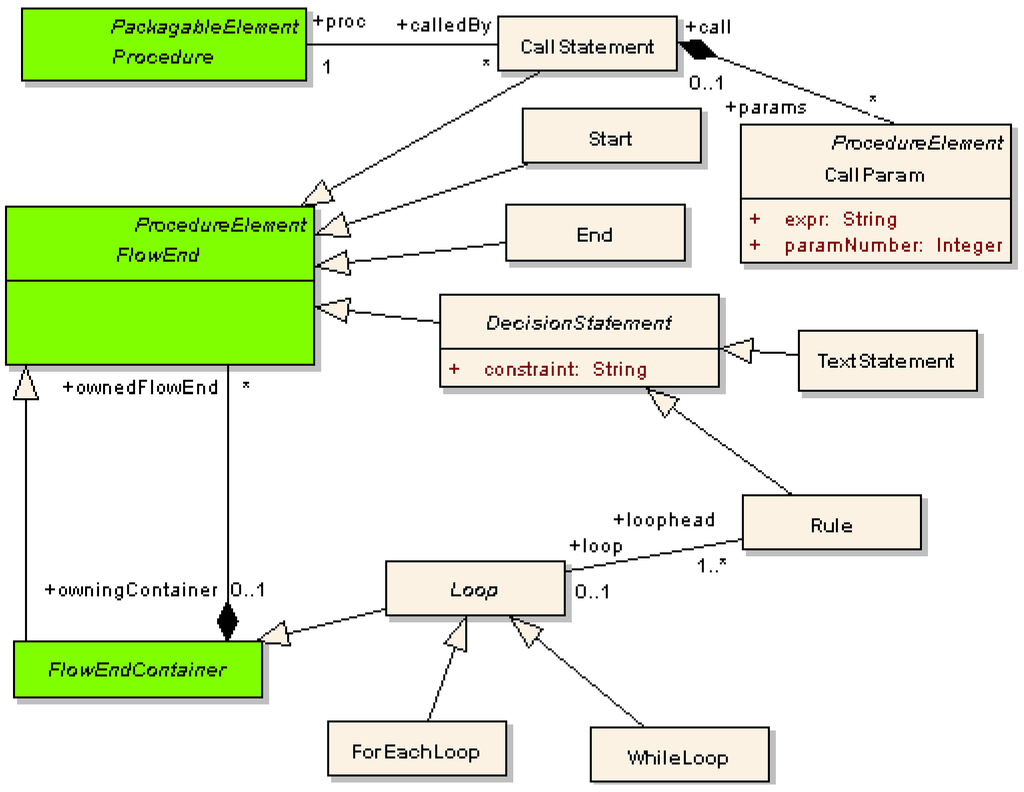


Figure 5.13: Statements in MOLA

*[inh]*source[*] - outgoing control flows.

*[inh]*destination[*] - incoming control flows

*[inh]*owningContainer[0..1] - the statement container which owns this statement, if any.

ownedFlowEnd[*] - statements owned by this container. The container may not contain itself directly or indirectly.

The fragment of the MOLA metamodel that describes all possible statements in MOLA is shown in Fig. 5.13.

Start

Brief overview. Start is an element of MOLA procedure which represents the starting point of a MOLA procedure.

Generalisations

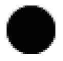


Figure 5.14: Start symbol notation

It is specialised from FlowEnd

Associations.

*[inh]*comment[*] - comments on this starting point.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this starting point

*[inh]*source[*] - outgoing control flows. Starting point has exactly one outgoing control flow. No alternate outgoing control flow is allowed.

*[inh]*destination[*] - incoming control flows. Starting point has no incoming control flows.

*[inh]*owningContainer[0..1] - the statement container which owns this statement, if any. Starting point may not be owned by a container

Notation. A starting point is shown as solid black circle. (See Fig. 5.14)

Semantics. MOLA procedure may have exactly one starting point. The statement which will be executed next is determined by the outgoing flow. All statements that are not inside containers must be reachable by control flows from the starting point of the procedure.

End

Brief overview. End is an element of MOLA procedure which represents the termination point of a MOLA procedure.

Generalisations

It is specialised from FlowEnd

Associations.

*[inh]*comment[*] - comments on this termination point.

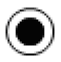


Figure 5.15: End symbol notation

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this termination point

*[inh]*source[*] - outgoing control flows. Termination point has no outgoing control flows.

*[inh]*destination[*] - incoming control flows. Termination point has at least one incoming control flow.

*[inh]*owningContainer[0..1] - the statement container which owns this termination point, if any.

Notation. A termination point is shown as a solid circle with a hollow circle (See Fig. 5.15).

Semantics. A MOLA procedure must have at least one termination point. The termination point ends the execution of the MOLA procedure. Current values of in-out parameters are passed back to the calling MOLA procedure.

CallStatement

Brief overview. CallStatement is an element of MOLA procedure which represents the call of a procedure. It can be either MOLA procedure or external procedure which is called by the call statement. The called procedure is executed. When it terminates the next statement after this call statement is executed. Call parameters are expressions.

Generalisations

It is specialised from FlowEnd

Associations.

*[inh]*comment[*] - comments on this call statement.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this call statement

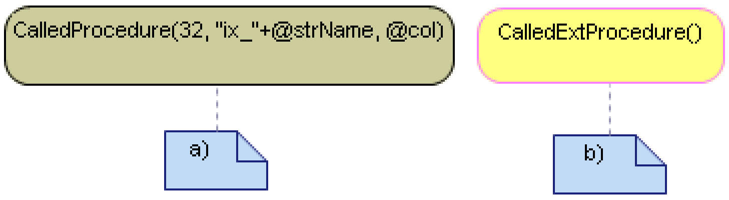


Figure 5.16: Call statement notation a) to a MOLA procedure; b) to an external procedure

*[inh]*source[*] - outgoing control flows. Call statement has one or none outgoing control flow. The outgoing control flow may be absent if the call statement is owned by a loop.

*[inh]*destination[*] - incoming control flows. Call statement has at least one incoming control flow.

*[inh]*owningContainer[0..1] - the statement container which owns this call statement, if any.

proc[1] - the procedure which is called by this call statement.

params[*] - call parameters of the call statement.

Notation. A call statement is shown as a khaki rounded rectangle if the called procedure is a MOLA procedure (see Fig. 5.16). A call statement is shown as a yellow rounded rectangle with pink border if the called procedure is an external one (see Fig. 5.16). The body of the call statement symbol contains the name of the called procedure and the parameter list enclosed in parenthesis. Parameters are separated by commas. If the called procedure does not belong to the same package the full qualified name must be used.

<call-statement-notation>::=

(<called-procedure-name>|<called-procedure-full-qualified-name>)'('[<call-parameters>]')'

<call-parameters>::=<call-parameter>{', '<call-parameter>}*

Semantics. A call statement is used to call MOLA or an external procedure. Call parameters must be supplied according to the order and types that are specified by definition of the called procedure. If a parameter is defined as *in* then the corresponding call parameter may be an arbitrary expression of the same type as the parameter (parameter is passed by value). If a parameter is defined as *in-out* then the corresponding call parameter must be a referenceable element (variable or parameter, but not pointer to class element) of the same type as the parameter (parameter is passed by reference).

The next statement is executed when the called procedure ends the execution. The next statement is determined by the outgoing control flow. The outgoing control flow may be absent if the call statement is owned by a loop. Then the next iteration of the loop is executed.

CallParam

Brief overview. CallParam is a textual element of MOLA procedure which is used to specify the value of the parameters that are passed to a called procedure.

Generalisations

It is specialised from ProcedureElement

Attributes

expr:String - expression (see 5.2.5 for details) that is evaluated and the result is passed as a call parameter value, when the call statement is executed. The type of the expression is constrained by the definition of the called procedure - the parameter with the same number constrains the type of the expression.

<call-parameter>::=<expression>

paramNumber:Integer - the number of the call parameter.

Associations.

*[inh]*comment[*] - comments on this call parameter. No comments may be added to the call parameter.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this call parameter. This link may be absent.

call[1] - call statement that owns the call parameter.

Notation. A call parameter is a textual element and it contains the expression which must be evaluated. A call parameter is shown in the body of the call statement. (see Fig. 5.16). Call parameters are separated by commas.

DecisionStatement

Brief overview. DecisionStatement is an abstract metaclass which represents the conditional statements in MOLA. The condition of a decision statement is examined. The action is executed depending on the state of condition. There are two types of conditional statements - graphical (rule) and textual (text statement). The decision statement may contain *actions (action part)* - object and link creations, deletions, attribute and variable assignments.

Generalisations

It is specialised from Flowend

Specialisations

There are subclasses Rule, TextStatement.

Attributes

constraint:String[0..1] - the constraint expression (See 5.2.5 for details) that forms the textual part of the condition.

<constraint>::=<constraint-expression>

Associations.

*[inh]*comment[*] - comments on this decision statement.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this decision statement

*[inh]*source[*] - outgoing control flows. A decision statement has no more than two outgoing control flows. If there are two outgoing control flows, then one of them must be alternate (*ELSE*) control flow. The outgoing control flow may be absent if the decision statement is owned by a loop.

*[inh]*destination[*] - incoming control flows.

*[inh]*owningContainer[0..1] - the statement container which owns this decision statement, if any.

ownedAssign[*] - attribute and variable assignments owned by this decision statement.

Semantics. A decision statement contains a condition. If the condition holds, then the action part of the decision statement is executed and the next statement is found by outgoing control flow without *ELSE* mark. If the condition fails, then the action part is not executed and the next statement is found by the alternate (*ELSE*) outgoing control flow. The procedure execution is terminated, if the corresponding outgoing control flow is absent and the decision statement is not owned by a loop. If the decision statement is owned by a loop, then the next iteration of the loop is executed.

Rule

Brief overview. Rule is a graphical decision statement - the most typical construct in MOLA. The condition is specified by a *pattern* and a textual constraint. A pattern is built using class elements and association links. A class element corresponds to a metamodel class. An association link connecting two class elements corresponds to an association linking the respective classes in the metamodel. A pattern is a set of class elements and links, which are compatible to the metamodel for this transformation. The pattern has been matched if appropriate instances in the model have been found.

The action part of the rule is built also using class elements and association links. Class elements of the action part represents object creation and deletion, but association links - link creation and deletion. Textual assignments for matched instances are used to set attribute values. Action part may use the instances found by the pattern matching.

There is a special type of rule - *loophead.* A loophead specifies the condition of a loop.

Generalisations

It is specialised from DecisionStatement

Attributes

*[inh]*constraint:String[0..1] - the constraint expression (See 5.2.5 Expressions for details) that forms additional textual part of the condition.

Associations.

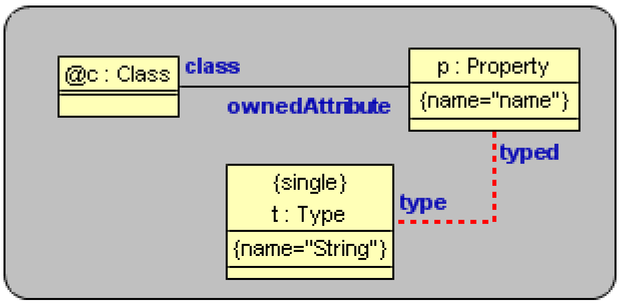


Figure 5.17: Rule notation

*[inh]*comment[*] - comments on this rule.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this rule

*[inh]*source[*] - outgoing control flows. A rule has no more than two outgoing control flows. If there are two outgoing control flows, then one of them must be alternate (*ELSE*) control flow. The outgoing control flow may be absent if the decision statement is owned by a loop. The loophead may not contain alternate outgoing control flow.

*[inh]*destination[*] - incoming control flows. There are no incoming control flows if the rule is a loophead.

*[inh]*owningContainer[0..1] - the statement container which owns this rule, if any.

*[inh]*ownedAssign[*] - attribute and variable assignments owned by class elements in this rule.

loop[0..1] - the loop, if this rule is the loophead of that rule. This loop must be also the owner of this loophead.

ownedElem[1..*] - class elements that specify the pattern and actions of the rule.

ownedLink[*] - association links that specify the pattern and actions of the rule.

Notation. A rule is shown as a gray rounded rectangle (See Fig. 5.17). Class elements and association links are shown inside the symbol of a rule. Class elements may contain constraints, attribute assignments and annotations.

Semantics. The condition of a rule is formed using pattern and an additional textual constraint. The condition holds if the pattern matches and the textual constraint (if present) evaluates to *true*. More on pattern semantics see Section 5.2.6. If the condition holds the action part of the rule is executed.

The action part of the rule includes:

- Instance and link creation via *create* class elements and association links respectively.
- Instance and link deletion via *delete* class elements and association links respectively.
- Assigning values of attributes.

There is a special type of rule - *loophead*. A loophead specifies the condition of a loop. A while-loop is being executed while the condition of the loop holds. The loophead of a foreach loop contains a special class element - *loop variable*. The foreach loop is being executed for each distinct instance that corresponds to the loop variable.

TextStatement

Brief overview. TextStatement is a textual decision statement. The condition of a text statement is specified using a textual constraint. A text statement is used when no pattern matching and no instance creation is needed. The condition operates with variables and already matched instances. A text statement may contain assignments. New values can be set to variables and to attributes of instances.

Generalisations

It is specialised from DecisionStatement

Attributes

*[inh]*constraint:String[0..1] - the constraint expression (see 5.2.5 Expressions for details) that forms the condition of the text statement.

<constraint>::=<constraint-expression>

Associations.

*[inh]*comment[*] - comments on this text statement.

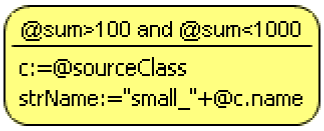


Figure 5.18: Text statement notation

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this text statement

*[inh]*source[*] - outgoing control flows. A text statement has no more than two outgoing control flows. If there are two outgoing control flows, then one of them must be alternate (*ELSE*) control flow. The outgoing control flow may be absent if the text statement is owned by a loop.

*[inh]*destination[*] - incoming control flows. There is at least one incoming control flow.

*[inh]*owningContainer[0..1] - the statement container which owns this text statement, if any.

*[inh]*ownedAssign[*] - attribute and variable assignments owned by this text statement.

Notation. A text statement is shown as yellow rounded rectangle. (See Fig. 5.18). The rectangle is divided by horizontal line. The upper compartment contains textual condition, but the lower compartment contains list of assignments. Any of the compartments may be absent.

Semantics. The condition of a text statement is formed using textual constraint. The constraint is a constraint expression, which is evaluated to find the state of the condition. Assignments form the action part of the text statement.

Loop

Brief overview. Loop is an abstract metaclass which represents the loop statements in MOLA. The loop concept in MOLA is very similar to classical programming languages - some code is repeatedly executed if the condition of the loop holds. There are while and foreach loops in MOLA. A loop is a container - it owns other statements.

Generalisations

It is specialised from FlowEnd

Specialisations

There are subclasses ForeachLoop, WhileLoop.

Associations.

*[inh]*comment[*] - comments on this loop.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this loop

*[inh]*source[*] - outgoing control flows. Loop has one or none outgoing control flow. The outgoing control flow may be absent if the loop is owned by another loop.

*[inh]*destination[*] - incoming control flows. Loop has at least one incoming control flow.

*[inh]*owningContainer[0..1] - the statement container which owns this loop, if any.

loophead[1..*] - loopheads of the loop. The loop must have exactly one loophead.

Notation. The loop may contain other executable elements. They are shown inside the box representing the loop. The precise notation is dependent on the type of the loop.

Semantics. A loop can be *while*, *foreach fixed* and *foreach not-fixed*. A loop owns exactly one rule, called *loophead*. The loophead describes the condition of the loop. The loophead has no incoming control flow and no more than one outgoing control flow which is not alternate (ELSE) control flow. The loophead action part is executed if the condition of the loop holds. All other statements owned by the loop are called a *loop body*. It must be reachable from the loophead by control flows. If a statement in the loop body has no appropriate outgoing control flow then the next iteration of the loop is executed. If the condition of the loop fails then the next statement is executed. The next statement is found via outgoing control flow of the loop. The outgoing control flow may be absent if the loop is owned by another loop. Then the next iteration of the owning loop is executed.

Outgoing control flows may go out of the loop, but it is not permitted to draw an incoming control flow to the statement inside the loop from statement which is not directly or indirectly owned by the loop.

ForeachLoop

Brief overview. ForeachLoop is a loop, which is used to execute statements for each instance of a selected class such that the pattern matches. The condition of a foreach loop is specified by the loophead. A pattern of a loophead contains a special class element - a *loop variable*. The loop variable specifies the set of instances the loop is executed through.

Generalisations

It is specialised from Loop

Attributes

notFixed:Boolean - indicates that a foreach loop is *not-fixed*. Otherwise it is *fixed.*

Associations.

*[inh]*comment[*] - comments on this foreach loop.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this foreach loop

*[inh]*source[*] - outgoing control flows. Foreach loop has one or none outgoing control flow. The outgoing control flow may be absent if the foreach loop is owned by another loop.

*[inh]*destination[*] - incoming control flows. Foreach loop has at least one incoming control flow.

*[inh]*owningContainer[0..1] - the statement container which owns this foreach loop, if any.

*[inh]*loophead[1..*] - loopheads of the foreach loop. The foreach loop must have exactly one loophead.

Notation. A fixed foreach loop is shown as a rectangle with bold borders (see Fig. 5.19). A not-fixed foreach loop is shown as a rectangle with bold borders and the right upper corner bent (see Fig. 5.20). The loophead is shown as rule, which contains a loop variable - a class element with bold borders. The loop body is drawn inside the foreach loop symbol.

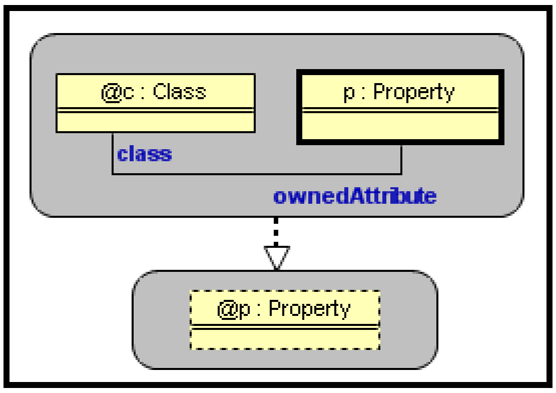


Figure 5.19: Fixed foreach loop notation

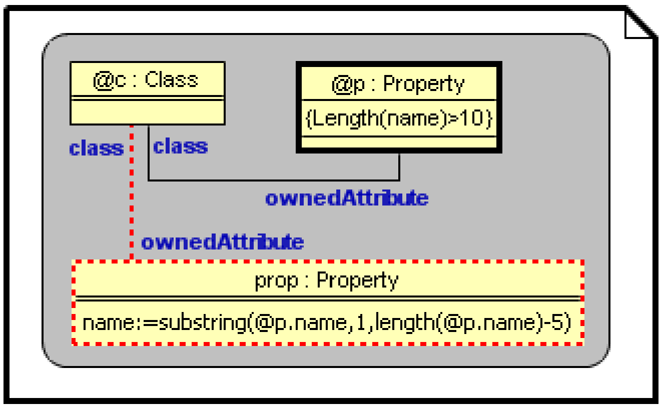


Figure 5.20: Not-fixed foreach loop notation

Semantics. A foreach loop is a loop which is executed over a set of instances that are specified by the *loop variable*. The loop variable is a special class element. There is exactly one loop variable in a loophead. To find the instances a foreach loop must iterate through, the condition (loophead pattern and additional constraints) is evaluated (the pattern matched). The loop body is executed for all possible matches for the loophead, which differ by instances allocated to the loop variable. The order instances are traversed is dependent of the loophead pattern (see 5.2.6).

The condition is evaluated only once for a *fixed* foreach loop. Namely, all instances of the loop variable which have the corresponding pattern match are selected. The actions of the loophead and loop body are executed for each selected instance. The fixed foreach loop corresponds to the foreach loop in traditional programming languages.

The condition is evaluated on every iteration for a *not-fixed* foreach loop. Namely, the first instance of the loop variable which has the corresponding pattern match and has not been selected yet, is selected. The actions of the loophead and loop body are executed for it. Thus the not-fixed foreach loop is used when foreach loop iteration may produce a new instance of the loop variable that must be processed.

WhileLoop

Brief overview. WhileLoop is a loop, which is used to execute statements while the loop condition holds. A while-loop contains a condition which is specified using pattern included in the special rule - *loophead.*

Generalisations

It is specialised from Loop

Associations.

*[inh]*comment[*] - comments on this while-loop.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this while loop

*[inh]*source[*] - outgoing control flows. while loop has one or none outgoing control flow. The outgoing control flow may be absent if the while loop is owned by another loop.

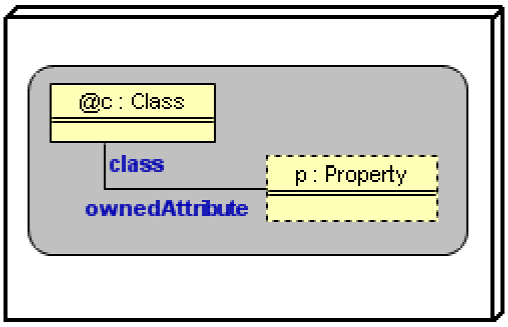


Figure 5.21: While-loop notation

*[inh]*destination[*] - incoming control flows. While-loop has at least one incoming control flow.

*[inh]*owningContainer[0..1] - the statement container which owns this while loop, if any.

*[inh]*loophead[1..*] - loopheads of the while loop. The while-loop must have exactly one loop-head.

Notation. A while-loop is shown as a 3D-rectangle with bold borders (see Fig. 5.21). The loophead is shown as rule which has no incoming control flow. The loop body is drawn inside the foreach loop symbol.

Semantics. A while-loop is a loop which is executed as long as the condition holds. The condition of the while-loop consists of a pattern and a textual constraint. If the condition of the while-loop holds then the loophead action part and loop body is executed, else the next statement found by the outgoing control flow is executed.

The fragment of the MOLA metamodel that describes the means of variable definition in MOLA is shown in Fig. 5.22.

ReferencableElement

Brief overview. ReferencableElement is an abstract metaclass that represents procedure elements which may be referenced in a pattern and in an expression. In fact, referencable elements are used to introduce variable concept in MOLA. A referencable element has a name and a type and may have a value. The value of a referencable element is constrained by a type (class, enumeration or primitive type) defined in the metamodel. Parameters, variables and class elements are referencable elements in MOLA.

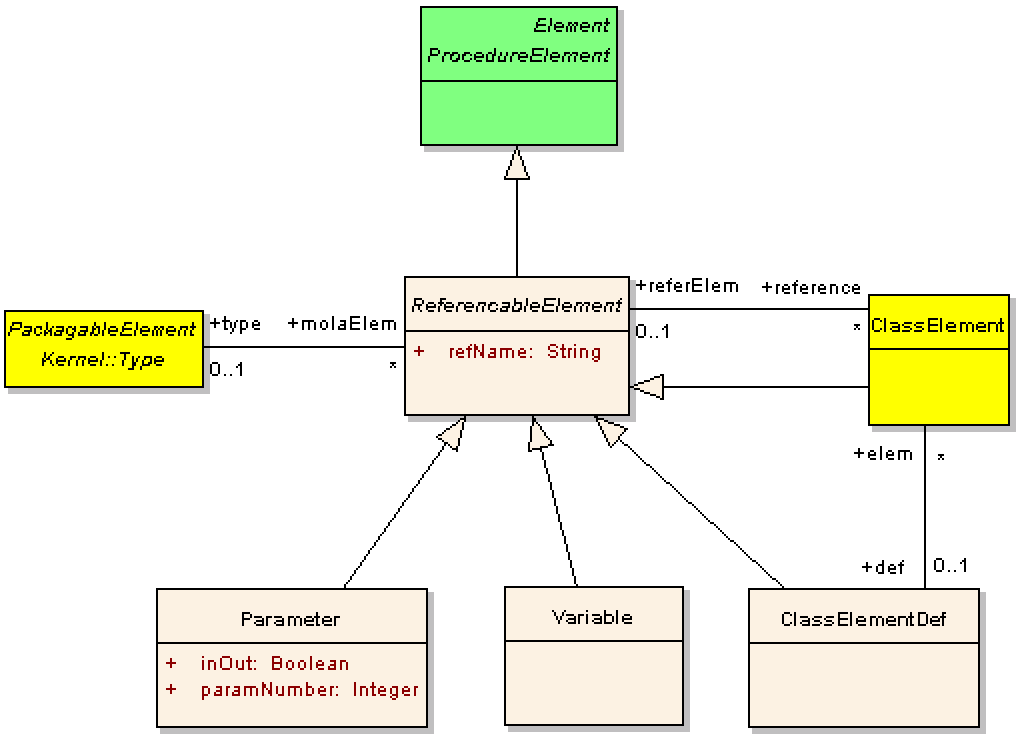


Figure 5.22: Referencable elements

Generalisations

It is specialised from ProcedureElement

Specialisations

There are subclasses Parameter, Variable, ClassElementDef, ClassElement.

Attributes

refName:String[0..1] - the name of the referencable element which is used to reference it within a pattern or an expression. If the name is set for the referencable element, it must be unique within the owning procedure.

<referencable-element-name>::=<pointer-name>|<elementary-variable-name>

<pointer-name>::=<name>

<elementary-variable-name>::=<name>

Associations.

*[inh]*comment[*] - comments on this referencable element.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this referencable element.

*[inh]*annotation[*] - annotations added to this referencable element.

type[0..1] - the type which constrains values of the referencable element. If the referencable element is class-typed, then it is called *pointer*, otherwise - *elementary variable*.

reference[*] - class elements that reference this referencable element. Only pointers may be referenced by a class element.

assignment[*] - assignments where the value of the referenceable element is set. If the referencable element is a pointer then values of instance attributes also may be set.

Semantics. Referencable elements in MOLA are parameters, variables and class elements. A referencable element has name and type, which is specified by a type defined in the metamodel.

A referencable element may have a value in the runtime. A referencable element may be used in the left hand side of an assignment. Then the value of the referenceable element is set. If a referencable element is a pointer, then values of attributes may be also set.

A referencable element may be referenced by a class element.

Parameter

Brief overview. Parameter is a referencable element, which defines a parameter of a procedure. When a procedure is called, then call parameters must be supplied. Those call parameters must comply to the parameter definitions.

Generalisations

It is specialised from ReferencableElement

Attributes

*[inh]*refName:String[0..1] - the name of the parameter which is used to reference it within a pattern or an expression.

inOut:Boolean - indicates that the call parameter is passed to a procedure by reference. Otherwise it is passed by value.

paramNumber:Integer - number of the parameter. Parameters of a procedure are numbered. Numbering starts from one. Parameter numbers determine the order call parameters must be supplied by a call statement.

Associations.

*[inh]*comment[*] - comments on this parameter.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this parameter. If the parameter is used for an external procedure, then this link can not be set.

*[inh]*type[0..1] - the type which constrains values of the parameter. The parameter may be of any type. A call parameter (actually the expression) supplied to call statement must be of the same type as the parameter with the same parameter number.

*[inh]*reference[*] - class elements that reference this parameter. Only class-typed parameters may be referenced by a class element.

*[inh]*assignment[*] - assignments where the parameter is in the left side. The parameter value may be set by an assignment. If the parameter is a pointer then values of instance attributes also may be set.

proc[0..1] - the procedure this parameter corresponds to. This link must be set, if the procedure is external.

Notation. A parameter may be shown in the project tree as child node of the procedure owning it. If a parameter is owned by a MOLA procedure, then it is shown as a white convex flag if the parameter is *in* (see Fig. 5.23), and it is shown as white hexagon, if the parameter is *in-out* (see Fig. 5.23). The body of the parameter symbol contains the upper and the lower part. The upper part is the string which consists of the parameter name, prefixed with ‘@‘-symbol, and type name, prefixed by the ‘ : ‘string. If the type is included in a package - the full name of the owning package in the curly brackets is displayed in the lower part. The lower right corner contains the parameter number.

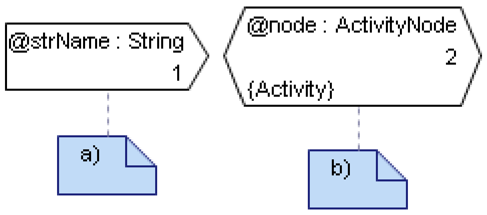


Figure 5.23: Parameter notation a) in b) in-out

Semantics. Parameters define the types and order of arguments - call parameters - that must be supplied when a procedure is called. The type may be a class, an enumeration or a primitive type. The order of parameters is determined by parameter numbers. Parameter numbering must be started from one.

A parameter is a referencable element, whose value is set when procedure is started. Principles of the parameter passing mechanism are identical to traditional programming languages. A parameter may be passed by reference, then it is called *in-out parameter,* or by value, then it is called *in parameter*. Note, if the parameter is pointer and even it is *in*, changes done to the instance (create link, set attribute, etc.) are permanent.

A parameter may be referenced by a class element

Variable

Brief overview. Variable is a referencable element, which defines a variable in the MOLA procedure. A variable gains the value via assignment operation. It may be used in a pattern, in a textual constraint or in an assignment. A variable may be of any type.

Generalisations

It is specialised from ReferencableElement

Attributes

*[inh]*refName:String[0..1] - the name of the variable which is used to reference it within an expression or assignment.

Associations.

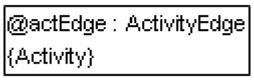


Figure 5.24: Variable notation

*[inh]*comment[*] - comments on this variable.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this variable.

*[inh]*type[0..1] - the type which constrains values of the variable.

*[inh]*reference[*] - class elements that reference this variable. Only class-typed variables may be referenced by a class element.

*[inh]*assignment[*] - assignments where the value of the variable is set. If the variable is a pointer then values of instance attributes also may be set.

Notation. A variable is shown as a white rectangle (see Fig. 5.24). The body of the variable symbol contains the upper and the lower part. The upper part is the string which consists of the variable name, prefixed with '@'-symbol, and type name, prefixed by ' : '-string. If the type is included in a package then the full name of the owning package in the curly brackets is displayed in the lower part.

Semantics. A variable can be used in a pattern, in an expression or in an assignment.

A variable may be referenced by a class element

ClassElementDef

Brief overview. ClassElementDef is a referencable element that represents the definition of a non-reference class element. A class element definition defines a pointer, which can be reused by several class elements. This is a MOLA element which has no direct counterpart in the graphical syntax of MOLA.

Generalisations

It is specialised from ReferencableElement

Attributes

*[inh]*refName:String[0..1] - the name of the class element.

<class-element-name>::=<name>

Associations.

*[inh]*comment[*] - comments on this class element definition. No comments may be added to the class element definition.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this class element definition.

*[inh]*type[0..1] - the type which constrains values of the class element. This type may only be a class.

*[inh]*reference[*] - class elements that reference this class element definition. In fact, these class elements reference the class element which is defined by this definition.

*[inh]*assignment[*] - assignments where the value of attribute of the instance (pointed to by a class element using this definition) is set. A class element definition may be used in the left side of the assignment only to set the value of an attribute in text statement.

elem[*] - non-reference class elements that are defined by the class element definition. There must be at least one such class element. All defined class elements must be in different rules.

Semantics. This metaclass is added to the MOLA metamodel mainly to solve the problem of the name reuse of class elements. Class elements may have the same name and type, if they are used in different patterns. When the reference to such class element is needed, then there is ambiguity. To solve this issue this little bit artificial metaclass is introduced. Thus for each non-reference class element the corresponding class element definition is introduced “behind the scene“ - such class element must have def link to the appropriate definition. Certainly, several class elements may share the same definition if they have the same name. There is no graphical representation of the class element definition. If the class element references other class element, technically the referElem link must be set to the definition of that class element, not to the class element itself.

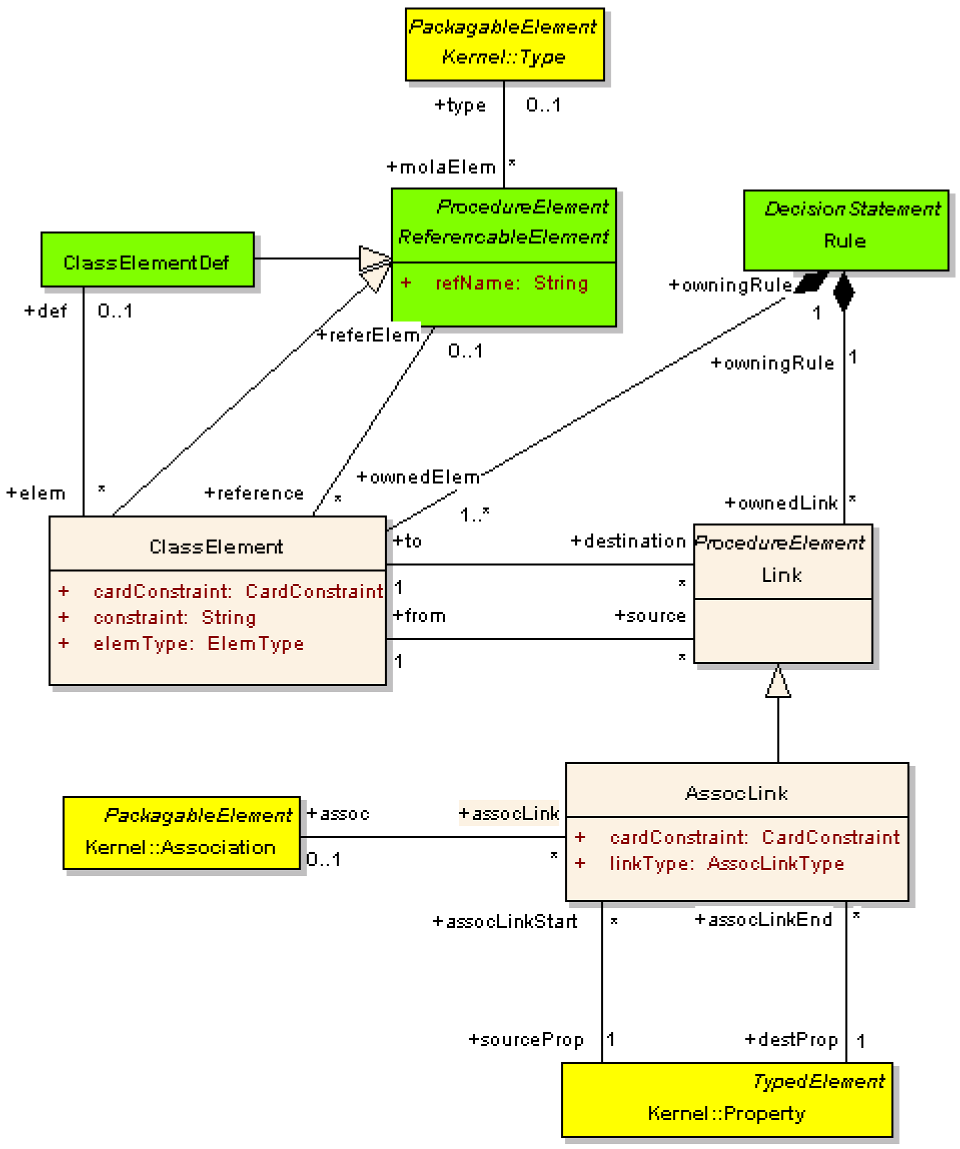


Figure 5.25: Pattern definition elements

The fragment of the MOLA metamodel that describes the facilities of the pattern definition is shown in Fig. 5.25. The patterns in MOLA are defined using class elements and association links.

ClassElement

Brief overview. ClassElement is a graphical element that is used to define a pattern and actions of a rule. The class element represents an instance of a particular metamodel class. The class element may be *non-reference* or *reference*. A non-reference class element is used to get a pointer to a particular instance. A reference class element is used to reference a known instance. Non-reference class elements can be *normal*, *create*, *delete* and *loop variable*. Reference class element can be *normal* or *delete*. Normal class element is used to build a pattern. It specifies that the condition of the rule may be *true* only if there is an instance which matches to the features defined by the class element. Create class element is used to create an instance of the particular type. Delete elements are used to delete the particular instance. Loop variable is a special type of class element that is used to denote the instance set a foreach loop is iterating through. All class elements may contain attribute value assignments, which are executed if the condition of the rule holds.

Generalisations

It is specialised from ReferencableElement

Attributes

*[inh]*refName:String[0..1] - the name of the class element. It is derived from the class element definition or the referenced element.

constraint:String[0..1] - additional constraint on this class element. It is a constraint expression (see section 5.2.5). The instance is matched only if the constraint is *true*. Constraints can be added to class elements that are used to build the pattern of the rule. They can be *normal*, *delete* or *loop variable* class elements. Mainly the constraints are set on attribute values. Note, that a constraint may contain a reference (pointer) to another class element of the same pattern. For example, in order to find two instances with the same name, the constraint actually must apply to the whole pattern (not only to the class element).

<constraint>::=<constraint-expression>

elemType - the type of the class element. It may be *normal, create*, *delete* or *loop variable*.

cardConstraint:CardConstaint[0..1] - the cardinality constraint. It may be set for a normal class element. Currently, the only cardinality constraint is the *NOT* constraint. A class element with NOT constraint is called the *NOT-element*. The semantics of the NOT-element is opposite to a normal class element without cardinality constraint. The condition of a pattern holds only if there is no instance which matches to the features defined by the NOT-element. A NOT-element may not participate in any action.

Associations.

*[inh]*comment[*] - comments on this class element.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this class element.

*[inh]*annotation[*] - annotations added to this class element. Annotations are used to add semantics to the pattern. An annotation may be added to a *normal* or *delete* non-reference class element which has no cardinality constraints. No more than one annotation may be added. Currently there are two types of annotations - *single* and *start*. More on semantics of the pattern see section 5.2.6.

*[inh]*type[0..1] - the type which constrains values of the class element. It is derived from the class element definition or referenced element. Class element may be only class-typed.

*[inh]*reference[*] - class elements that reference this class element. Actually, this link is never set. If any class element references another class element, the link is set to the definition of this class element.

*[inh]*assignment[*] - assignments where the values of instance attributes are set.

owningRule[1] - the rule that is owning this class element.

def[0..1] - class element definition for a *non-reference* class element.

referElem[0..1] - the referencable element that is referenced by this class element. This link is set for class element that is a *reference*. A class element may reference a class-typed referencable element (variable, parameter or class element). If it references other class element, then this link actually is set to the definition of that class element.

source[*] - outgoing association links.

destination[*] - incoming association links.

Notation. Class element is shown as a rectangle. It may be placed only inside a rule. A class element has three compartments. The upper compartment contains the name of the class element and the type name. The name of the class element is prefixed by '@'symbol if the class element is a reference. If the class element is annotated or it has cardinality constraint then the annotation text or cardinality constraint text respectively is shown above the name of the class element in curly brackets. If the type is owned by a package, the full name of the package is shown below the class element name (also in curly brackets). The middle compartment of the class element may contain additional constraints on this class element - mainly constraints on attribute values. Constraints are also placed in curly brackets. The lower compartment of the

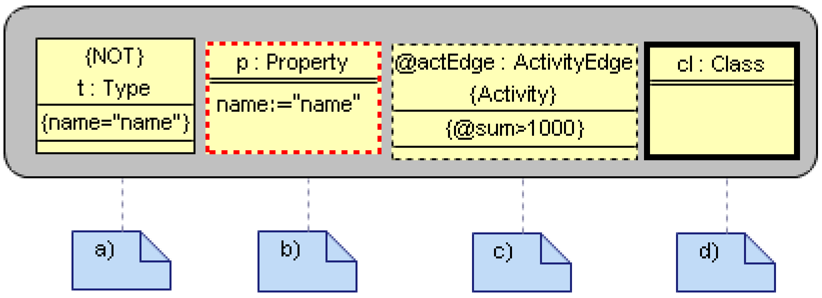


Figure 5.26: Class element notation a) normal b) create c) delete d) loop variable

class element may contain the list of attribute assignments (one per line). If the class element is *normal* then the border of the class element symbol is a black solid line (see Fig. 5.26). If the class element is *create* then the border of the class element symbol is a red dotted line (see Fig. 5.26). If the class element is *delete* then the border of the class element symbol is black dashed line (see Fig. 5.26). If the class element is a *loop variable* then the border of the class element symbol is black bold line (see Fig. 5.26.

Semantics. A class element represents a particular instance (exactly one) in the model. It has a name - an identifier that can be used to refer to this instance in the current MOLA procedure. A class element has also the type specification - class from the metamodel - that constrains the scope of possible instances that can be represented by this class element.

Class elements may be of two kinds - *reference* and *non-reference*. A non-reference class element gets the pointer to particular instance when a pattern has been matched or a create instance operation has been performed in the rule owning the class element (during the execution of the rule). A reference class element gets the pointer to particular instance before the execution of the rule. It refers to the referencable element that already has the value. It may be a parameter, a variable or already matched class element.

A *normal* class element is used in a pattern. It expresses the fact that the condition of the rule may hold (i.e., the pattern matches) only if there exists an instance of the type specified by the class element and the constraint for this class element evaluates to *true*. If the pattern matches then the class element represents the matched instance. The *normal* class element may be a reference. Then it represents the particular instance in the pattern. There may be a constraint also set on the reference. The role of a reference class element in a pattern is to add additional constraints on non-reference class elements in the pattern. The *delete* element plays the same role as a *normal* element in a pattern, but the particular instance is deleted in the action part of the rule. When the instance is deleted, all links connected to it also are deleted.

The *create* class element is used to create an instance of the particular type. A create class element may be only *non-reference*. If the condition of the rule holds (pattern matches), the instance denoted by a *create* class element is created. Then it represents the newly created instance.

The *loop variable* is used in the loophead of a foreach loop. It denotes the class element that is used to find the set of instances to iterate through. It is a *non-reference* class element. Otherwise it has the same semantics as *normal* class element.

A class element may contain assignments, which set attribute values for the corresponding instance. They are performed if the condition of the rule (or loophead) holds.

A class element may be connected to another class element by an association link. An association link may play different roles. It may be part of the pattern - it may form a constraint on links connecting the relevant instances. An association link may also be in the action part of the rule - be a create link or delete link.

Link

Brief overview. Link is an abstract metaclass that represents directed connectors that may be drawn in a rule between two class elements. Mainly it is used to specify constraint on particular link or to create or delete a particular link.

Generalisations

It is specialised from ProcedureElement

Specialisations

There is a subclass AssocLink.

Associations.

*[inh]*comment[*] - comments on this connector.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this connector.

*[inh]*annotation[*] - annotations added to this connector.

owningRule[1] - the rule that owns the connector.

from[1] - the class element the connector is going from - the source of the connector.

to[1] - the class element the connector is going to - the target of the connector.

Constraints.

The source and the target class elements must be in the same rule.

Notation. A connector is a line that connects two class elements. A line may consist of one or more connected segments. The individual segments of the line itself have no semantic significance. The precise notation is dependent on the precise type of a connector.

AssocLink

Brief overview. AssocLink is a connector, called association link. It is used to define a pattern and actions of a rule. The association link represents an instance of the particular metamodel association - a link. It has no identifier, thus it can not be referenced. The association link is used as a part of the pattern. Then it adds the constraint to the condition - the existence of a link between two instances. The association link is used also to create or to delete a link between two instances.

Generalisations

It is specialised from Link

Attributes.

linkType:AssocLinkType - type of the association link. It may be *normal*, *create* or *delete*.

cardConstraint:CardConstaint [0..1] - the cardinality constraint. It may be set for a *normal* association link. Currently, the only cardinality constraint is *NOT* constraint. An association link having the NOT constraint is called a *NOT-link*. The semantics of the NOT-link is opposite to a normal association link without cardinality constraint. The condition of the pattern holds only if there is no link (of the specified association) which connects the two given instances.

Associations.

*[inh]*comment[*] - comments on this association link.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this association link.

*[inh]*annotation[*] - annotations added to this association link. Currently there may be only one annotation - *c* annotation. It adds semantics to the pattern. More on semantics of the patterns see section 5.2.6.

*[inh]*owningRule[1] - the rule that owns the association link.

*[inh]*from[1] - the class element the association link is going from - the source of the association link. The association link actually is not directed. Therefore it is irrelevant whether the connected class element is a source or target.

*[inh]*to[1] - the class element the association link is going to - the target of the association link. The association link actually is not directed. Therefore it is irrelevant whether the connected class element is a source or target.

assoc[1] - the association the association link corresponds to. It must be a valid association that connects the classes that correspond to the source and the target class elements.

sourceProp[1] - the association end that is connected to the source class (the class corresponding to the source class element). It must be a member end of the association denoted by the assoc link.

destProp[1] - the association end that is connected to the target class (the class corresponding to the target class element). It must be a member end of the association denoted by the assoc link.

Notation. An association link is shown as a line that connects two class element symbols. Names of association ends are shown at the appropriate end of the line. If the association link is constrained or annotated, then the text of the constraint or annotation is shown in curly brackets at the middle of the line.

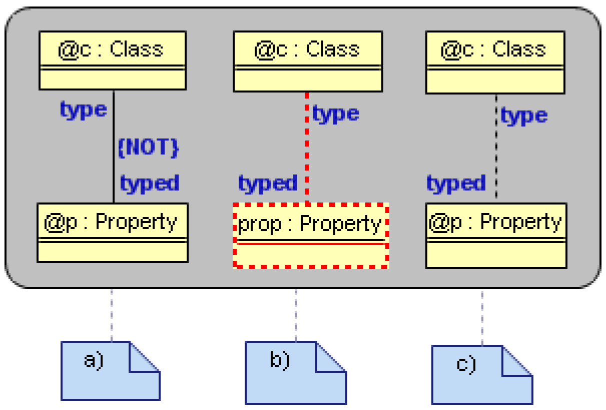


Figure 5.27: Association link notation a) normal b) create c) delete

The *normal* association link is shown as a black solid line (see Fig. 5.27). The *create* association link is shown as a red dotted line (see Fig. 5.27). The *delete* association link is shown as a black dashed line (see Fig. 5.27).

Semantics. An association link represents a particular link in the model. Each association link has the type specification - it corresponds to an association in the metamodel. The association must be a valid association that connects classes the class elements correspond to.

A *normal* association link is used as a constraint in the pattern. The pattern matches only if a link of the given type exists between instances denoted by the source and the target class elements. A *normal* association link may have a *NOT* constraint. Then it has the opposite semantics - the pattern matches only if there is no link of the given type between the instances denoted by the source and the target class elements. A *normal* link may connect *normal*, *delete* and *loop variable* class elements.

A *create* association link is used to create a link of the given type between instances denoted by the source and the target class elements. A *create* association link may connect *normal*, *create* and *loop variable* class elements. If an association end of the appropriate association is ordered then the created link is added as the last one in the existing sequence of ordered links.

A *delete* association link is used as a *normal* association link in the pattern, except that it may not have the *NOT*-constraint. If the pattern matches then the appropriate link is deleted. It may connect *normal* and *loop variable* class elements.

The fragment of the MOLA metamodel that describes the facility for the assignment definition is shown in Fig. 5.28.

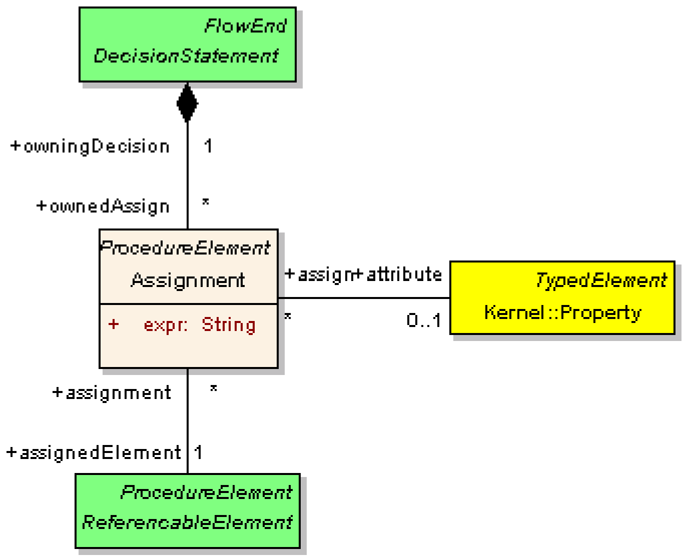


Figure 5.28: Assignments in MOLA

Assignment

Brief overview. Assignment is a textual element that is used to set value of a referencable element or an attribute value of a particular instance. It may be in the action part of a text statement or in a class element.

Generalisations

It is specialised from ProcedureElement

Attributes.

expr:String - expression (see 5.2.5 Expressions for details) that is evaluated and the result is assigned to a referencable element or an attribute. The type of the expression is constrained by the definition of the referencable element or attribute.

<assignment-expression>::=<expression>

Associations.

*[inh]*comment[*] - comments on this assignment.

*[inh]*owningProcedure[0..1] - the MOLA procedure which owns this assignment.

assignedElement[1] - the referencable element whose value is being set, if the attribute link is not present. Otherwise it denotes the pointer to the instance whose attribute is set.

owningDecision[1] - the decision statement (rule or text statement) which owns the assignment.

attribute[0..1] - the attribute of the instance which must be set. The pointer to the instance is denoted by the assignedElement link. The attribute must be a valid attribute of the class that corresponds to the pointer.

Notation. Assignments are textual elements that may be contained by class elements and text statements (See Fig. 5.26 and Fig. 5.18). An assignment is a string which consists of the left hand side that specifies the element whose value is set, the assignment sign ':='and the right hand side - an expression.

<assignment-notation> ::= (<referencable-element-name> | <attribute-specification>)
':=' <assignment-expression>

Semantics. The assignment is a textual element of the MOLA procedure. It is used to assign value to referencable elements or to the attribute of the instance denoted by a pointer. The assignment has two parts - the left hand side and the right hand side. The left hand side contains the element to be set. It may be an attribute specification, variable or parameter. If an assignment is within a class element then the left hand side may only be an attribute name of the corresponding class. If an assignment is within a text statement then the left hand side may be a variable name, parameter name or attribute specification. The attribute specification actually is a pointer (a class-typed variable or parameter or a class element) name, followed by an attribute name. A value may not be assigned to a pointer being a class element reference - it can receive its value only via pattern matching or instance creation. The right hand side of the assignment contains an expression of the appropriate type.

5.2.5 Expressions

MOLA is a graphical model transformation language, which has also textual elements. The most important part of textual elements in MOLA is *expressions*. They are used to describe constraints and to assign values to various elements - attributes of an instance, pointers, elementary variables and call parameters. Each expression has the result - a value of the particular type. The result may be a value of the primitive type (String, Integer or Boolean) or enumeration, or a pointer to an instance in the model. This section provides the description of expressions that are allowed in MOLA and the BNF-grammar that describes them.

Base elements

Base elements of expressions in MOLA are constants, elementary variables, pointers, navigations and attribute specifications. These are elements that hold values directly. There are four types of constants in MOLA:

- string constants (See 5.2.3 PrimitiveType)
- integer constants (See 5.2.3 PrimitiveType)
- boolean constants (See 5.2.3 PrimitiveType)
- enumeration constants (See 5.2.3 EnumerationLiteral)

An elementary variable (See 5.2.4 ReferencableElement) holds a value of the primitive type or enumeration. It is specified using the name of the elementary variable.

A pointer (See 5.2.4 ReferencableElement) points to an instance in the model. It is specified using the name of the pointer. The prefix '@' must be used before the name of the pointer. The prefix may be omitted only if the pointer is used in the constraint of the pattern and it points to the non-reference class element of the same pattern. If the pointer is used in a class element and it points to itself, then the keyword self must be used.

The navigation is a similar construct to the OCL navigation. It denotes the set of instances obtained via navigable links. The navigation is specified using a pointer name and appropriate rolenames separated by dots. The pointer denotes the instance the navigation starts from. The rolename separated from pointer name by dot denotes the type of links the navigation is performed over. The rolename must be a valid name of the association end leading to the class from the class denoted by the pointer. A set of instances is obtained this way. Each next rolename in the navigation string denotes the next type of a link to navigate over. The result is a set of instances. The simplest navigation is a pointer. It is interpreted as a set that consists of one instance.

An attribute specification holds a value of the particular attribute of the instance. This instance is specified using the navigation which results to one instance. It may be a pointer or a navigation via 1 or 0..1 links. The navigation may be omitted if the expression is used in a class element and the attribute specification denotes the attribute of the instance this class element points to.

<elementary-variable>::=<elementary-variable-name>

<pointer>::=[‘@‘]<pointer-name> | ‘**self**‘

<navigation>::=<pointer>(‘**.**‘<role-name>)*

<attribute-specification>::=[<navigation>’**.**’]<atribute-name>

Pointer expression

The result of the pointer expression is a pointer. The pointer expression may be:

- pointer
- NULL keyword denotes the value of the pointer which does not point to any instance.
- down-cast. The pointer may be down-casted accordingly to the class hierarchy defined in the metamodel. The down-cast is specified using the full qualified class name. The upcast is performed automatically in MOLA and does not need special specification.

<pointer-expression>::=<pointer> | ‘**NULL**‘ | <type-cast>

<type-cast>::=<full-class-name>’**(**’<pointer>’**)**’

Set expression

The result of the set expression is a set of instances. The set expression is specified using navigation.

<set-expression>::=<navigation>

Enumeration expression

The result of the enumeration expression is an enumeration value. The enumeration expression may be:

- enumeration constant
- attribute specification. The attribute type must be the enumeration.
- elementary variable whose type is the enumeration.
- toEnum function. The function converts the result of a string expression to the enumeration constant. Note, the exact enumeration type of the result is determined by the context the function is used.

<enum-expression>::=

<enumeration-literal-name> | <attribute-specification> |

<elementary-variable> | '**toEnum(**'<string-expression>'**)**'

Integer expression

The result of the integer expression is an integer value. The integer expression base elements are:

- integer constant
- attribute specification. The attribute type must be Integer.
- elementary variable whose type is Integer.
- Integer functions:
  - toInteger function converts the result of a string expression to an integer value.
  - size function returns the size of the result of a string expression or the size of the set specified by a set expression.

   – indexOf function returns the starting position of the substring in the string. Both are specified by a string expression. Note that character numbering starts form 1 in MOLA for strings. First argument specifies the substring to find, the second - the string value where to search.

Integer expression base elements may be used in more complex integer expressions. Standard arithmetical operations may be applied - addition (+), subtraction (-) and multiplication (*). The parenthesis may be also used.

<integer-expression>::=<int-factor> | <int-expression>(‘**+**‘|‘**-**‘)<int-factor>

<int-factor>::=<int-term> | <int-factor> ‘*‘ <int-term>

<int-term>::=

<integer-constant> | ‘**(**‘<integer-expression>‘**)**‘ |

<int-operation> | <attribute-specification> | <elementary-variable>

<int-operation>::=

‘**toInteger(**‘<string-expression>‘**)**‘ |

‘**size(**‘(<string-expression>|<set-expression>)‘**)**‘ |

‘**indexOf(**‘<string-expression>‘**,**‘<string-expression>‘**)**‘

String expression

The result of the string expression is a string value. The string expression base elements are:

- string constant
- attribute specification. The attribute type must be String.
- elementary variable whose type is String.
- String functions:

- toString function converts the result of an integer, an enumeration or a simple boolean expression to a string value.
- substring function returns the substring of the string. The substring is specified using the starting character and ending character positions in the string. The first argument is a source string specified by the result of a string expression. The second argument is the starting character position specified by the result of an integer expression. The third argument is the ending character position specified by the result of an integer expression. If the third argument is omitted, then the last character of the string is taken as ending character.
- toLower function converts all characters of the string value specified by the result of the string expression to the lowercase characters.
- toUpper function converts all characters of the string value specified by the result of a string expression to the uppercase characters.

String expression base elements may be used in more complex string expressions. The concatenation (+) operation may be applied.

<string-expression>::=<str-factor> | <string-expression>‘**+**‘<str-factor>

<str-factor>::=

<string-constant> | <str-operation> |

<attribute-specification> | <elementary-variable>

<str-operation>::=

‘**toString(**‘(integer-expression> | <enum-expression> | <bool-expression>)‘**)**‘ |

‘**substring(**‘<string-expression>‘**,**‘<integer-expression>[‘**,**‘<integer-expression>]‘**)**‘ |

‘**toLower(**‘<integer-expression>‘**)**‘ | ‘**toUpper(**‘<integer-expression>‘**)**‘

Simple boolean expression

The result of the simple boolean expression is a boolean value. The simple boolean expression may be:

- boolean constant
- attribute specification. The attribute type must be Boolean.
- elementary variable whose type is Boolean.
- toBoolean function converts the result of a string expression to a boolean value.
- type checking operations:
  - isTypeOf operation examines if the instance denoted by the pointer is of a particular type. This operation returns *true* if the instance is exactly of the specified class.
  - isKindOf operation examines if the instance denoted by the pointer is of a particular kind. This operation returns *true* if the instance is of specified class or its subclass.
- emptiness verification operations:
  - isEmpty operation examines if the particular attribute of an instance specified by an attribute specification is not set or the set specified by a set expression contains no elements. This operation returns *true* if the attribute is not set or the set is empty respectively.
  - notEmpty operation examines if the particular attribute of an instance specified by an attribute specification is set or the set specified by a set expression contains any element. This operation returns *true* if the attribute is set or the set is not empty respectively.

<bool-expression>::=

<boolean-constant> | <bool-operation> |

<attribute-specification> | <elementary-variable> |

<bool-operation>::=

'**toBoolean(**'<string-expression>'**)**' | <null-operation> | <type-check-operation>

<type-check-operation>::=

<pointer>'**.isTypeOf(**'<full-class-name>'**)**' |

<pointer>'**.isKindOf(**'<full-class-name>'**)**'

<null-operation>::=

(<attribute-specification>|<set-expression>)'**->isEmpty()**' |

(<attribute-specification>|<set-expression>)'**->notEmpty()**'

Simple expressions summary

Thus, there are six expression types: the integer expression, the string expression, the enumeration expression, the simple boolean expression, the pointer expression and the set expression.

<expression>::=

<integer-expression> | <string-expression> | <enum-expression> |

<bool-expression> | <pointer-expression> | <set-expression>

Simple constraints

Simple constraints may be described in MOLA using expressions described above and relational symbols = (equal), <> (unequal), < (less, strong subset), <= (less or equal, weak subset), > (greater, strong superset), >= (greater or equal, weak superset). If forms *the condition*, and the result of a simple constraint is *true* if this condition holds.

The = and <> symbols are used to compare two expressions of the same type. The condition holds if results of both expressions are equal or unequal respectively.

The <, <=, >, >= are used to compare the results of an integer expressions. The condition holds if the inequality is correct. These symbols may be used also to compare the results of two set expressions. The < symbol denotes that the set specified by the left expression is a subset of the set specified by the right expression. The <= symbol denotes that both sets may be also equal. > and >= symbols are used similar, only the set specified by the right expression is a subset of the set specified by the left expression.

The simple boolean expression also may be used as simple constraint. The condition holds if the result of a simple boolean expression is *true*.

<simple-constraint>::=

<bool-expression>| <expression>('**=**'|'**<>**')<expression>|

<integer-expression> ('**>**'|'**<**'|'**>=**'|'**<=**')<integer-expression> |

<set-expression>('**>**'|'**<**'|'**>=**'|'**<=**')<set-expression>

Constraint expression

A textual constraint is built using the constraint expression in MOLA. The constraint expression is a boolean expression. The standard operations may be used - or, and, not. Parentheses are also permitted. The operands of the constraint expressions are simple constraints.

<constraint-expression>::=<bool-factor> | <constraint-expression>' **or** '<bool-factor>

<bool-factor>::=<bool-term> | <bool-factor> ' **and** ' <bool-term>

<bool-term>::=

<simple-constraint> | '**(**'<constraint-expression>'**)**' | '**not** '<constraint-expression>

5.2.6 Additional remarks on syntax and semantics

This section contains the description of the semantics of patterns in MOLA and some additional remarks on semantics of the rule.

Pattern semantics

This section describes semantics of *the pattern* in MOLA. Patterns are used in graphical conditional statements - *rules*. A pattern forms the graphical part of the condition of a rule. A pattern specifies the fragment of the model that must be found. The model consists of instances, which may have links connecting them and values of attributes. Each instance corresponds to a class defined in the metamodel, as well as, each link corresponds to an association. Instances may have attribute values set. Thus, a pattern must contain information on instances, links and

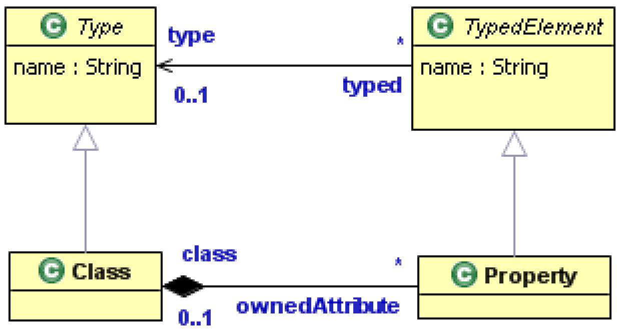


Figure 5.29: The metamodel used in examples

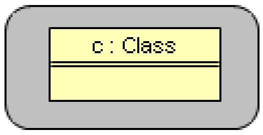


Figure 5.30: Pattern example

attribute values. Actually, a pattern describes what instances must be found. The process that is used to find the corresponding instances is called a *pattern matching*.

A pattern is defined by means of class elements (see ClassElement section 5.2.4) and association links (see AssocLink 5.2.4) in MOLA.

Normal, *delete* and *loop variable* class elements (pattern elements) form a pattern. A pattern element represents an instance of a particular metamodel class. A *non-reference* pattern element specifies that the condition of the rule may be *true* only if there is an instance which matches to the features defined by it. More precisely, if there is an instance that corresponds to the specified class. See an example Fig. 5.30. The metamodel that is used for examples of this section is shown in Fig. 5.29. The pattern matches if there is an instance of the type Class - there is a class in the model. The condition of the rule may be expressed using *many-sorted first-order logic* (∃c:Class).

Note that if there is more than one matching instance then first matched instance is chosen. If there is more than one pattern element in a pattern then all appropriate instances must exist. See an example Fig. 5.31. The pattern matches if there is an instance of the type Class and an instance of the type Property - there is a class and a property in the model. (∃c:Class∧∃p:Property).

Note that names of the pattern elements that are used in the same pattern must be unique. A pattern element may contain a constraint. The constraint is a Boolean expression (see 5.2.5 for details) that must evaluate to *true* for an instance in order to match it. A constraint may be used

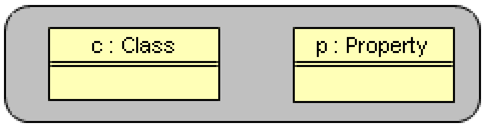


Figure 5.31: Pattern example

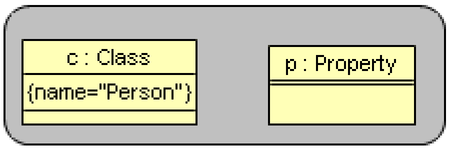


Figure 5.32: Pattern example

to constrain values of attributes. See an example Fig. 5.32. The pattern matches if there is an instance of the type Class whose "name" attribute value is "Person" and an instance of the type Property - there is a class *Person* and a property in the model. (∃c:Class (name(c)='Person') ∧∃p:Property).

Normal and *delete* association links (pattern links) are used in pattern to add an additional constraint to instances that are being matched. It defines that a link between appropriate instances must exist in order to match them. See an example Fig. 5.33. The pattern matches if there is a class *Person* which has an attribute in the model. (∃c:Class (name(c)='Person')∧∃p:Property (ownedAttribute(p,c)).

A pattern may contain also *reference* pattern elements. A reference pattern element represents already known instance. It is used to examine particular features of that instance or to add additional constraint to instances that are being matched. See an example Fig. 5.34. The pattern matches if there is a type set for the already known property in the model. (∃t:Type (type(t,@p))).

A *normal non-reference* class element which has a *NOT* cardinality constraint (*NOT-element*) specifies that the condition of the rule may be *true* only if there is no instance which matches to the features defined by it. See an example Fig. 5.35. The pattern matches if there is a class which has no *name* attribute in the model (∃c:Class ¬∃p:Property (ownedAttribute(p,c) ∧ name(p) = 'name'). Two *NOT-elements* may not be linked by an association link.

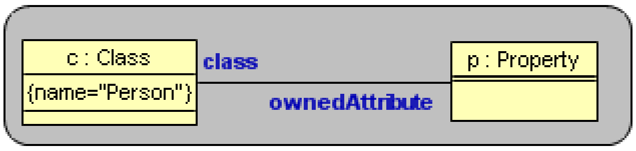


Figure 5.33: Pattern example

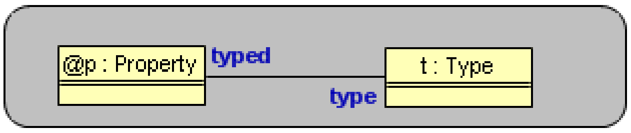


Figure 5.34: Pattern example

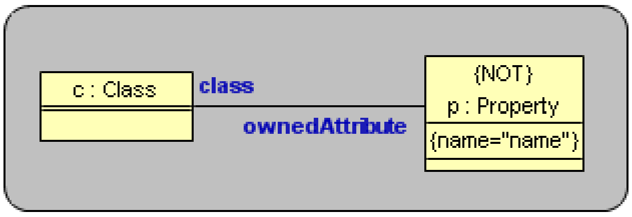


Figure 5.35: Pattern example

A *normal* association link which has a *NOT* cardinality constraint (*NOT-link*) specifies that the condition of the rule may be *true* only if there is no link of the given type between instances denoted by source and target pattern elements. See an example Fig. 5.36. The pattern matches if there is a property which is not typed by the already known type instance in the model. (∃p:Property (¬typed(p,@t))). A *NOT-link* may not be connected to a *NOT-element*.

Pattern elements that are linked by pattern links form a *pattern fragment*. Also, if two parts of a pattern fragment are linked by a *c-link* (see later) or *NOT-link* only, they are considered to be separate pattern fragments. The pattern may contain any number of pattern fragments. A pattern matches if all pattern fragments match. A pattern matching is a performance "bottleneck" of the implementation of transformation languages. Therefore it is strongly recommended to use annotations in MOLA. Each pattern fragment must contain an *initial* pattern element. It may be a *reference* class element or a *non-reference* class element that is annotated using *single* or *start* annotation to improve the performance of the pattern matching in MOLA. The *single* and *start* annotations denote that the pattern matching must start from this pattern element. Thus there can be only one annotated pattern element in a pattern fragment. The *single* annotation denotes also that only the first instance found must be examined. In other words, a "dynamic singleton situation" is assumed for this class. If no instance is found, the pattern fails. If there is an attribute constraint in this pattern element, the constraint is evaluated on the chosen instance, and if it is *false* the pattern fails. The *start* annotation means that the pattern matching starts from the annotated pattern element and if the first found instance does not satisfy constraints

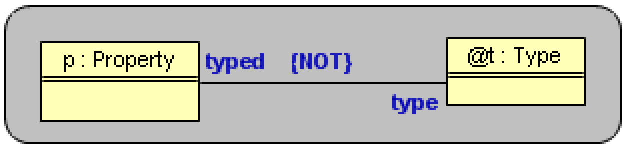


Figure 5.36: Pattern example

then the next instance is sought, and so on. In a loophead the *start* annotation may not be used in the pattern fragment which contains the *loop variable*. If a pattern fragment contains a *reference* pattern element then the annotated element is not necessary in this fragment. The search always starts from this pattern element. If there is more than one reference pattern element, an arbitrary one is chosen as the first, such situation should be avoided whenever possible. The pattern fragment which contains the loop variable may be without an annotated or reference class element. Then the loop variable is used as the initial pattern element.

To sum up, each pattern fragment must contain one *initial* pattern element - reference, *single* or *start*. Reference pattern elements may be more than one, if there is a *single* or *start* pattern element, then namely this annotated element is used as *initial*, but not references. If the pattern fragment contains several reference pattern elements, but no *single* or *start*, it is recommended to split up the fragment using *c-links*, e.g., to decide from which of the reference pattern elements a natural match should start and mark pattern links going to other references as *c-links* (these pattern links then mean additional constraints during the match, but other reference pattern elements themselves are treated as separate trivial pattern fragments). Hence, each pattern fragment is treated as a *rooted tree*, which is traversed from the root (*initial* pattern element) in all possible directions during the match.

A pattern link may have also an annotation - *c*. Then this link is called a *c-link*. The meaning of *c-link* is “check the presence of a link”. It is used to avoid situations when a pattern fragment contains a closed loop or there are more then one potential *initial* pattern element (no annotated pattern elements and more then one reference pattern element). The main goal of *c-links* is to make a rooted tree or several rooted trees from a pattern fragment.

A *loop variable* may be not the initial pattern element in the rooted tree of the pattern fragment. Then it is reachable by association links from the initial pattern element. The order in which the corresponding instances are traversed in the loop may be determined by the appropriate association end (in the direction towards the loop variable). If it is ordered then instances are traversed in the order determined by this ordering. Otherwise, the order is undetermined.

For example, if a loop variable is linked to a reference pattern element (which serves as the initial pattern element) and this association link corresponds to an association with an ordered end at the loop variable then instances of the loop variable are traversed in the order defined by this association. In particular, normally it is the order in which the appropriate links have been created.

Action part of the rule

The action part of the rule is executed if the condition of the rule holds, actually, if the pattern matches. Matched instances may be referenced in the action part using names of corresponding pattern elements.

The order actions are performed in a rule is:

- Creation of instances. It is specified using *create* class elements.
- Creation of links. It is specified using *create* association links.
- Assignment of attribute values. It is specified using assignments in class elements.
- Deletion of links. It is specified using *delete* association links.
- Deletion of instances. It is specified using *delete* class elements.

Class attributes which have cardinality *1* ("mandatory") must be assigned values in some MOLA assignment. If the value is not set, they get the value *NULL* ("undefined"). Attributes with the cardinality *0..1* simply may be absent.

Chapter 6

Possibilities of automatic transformations between models

6.1 Role of automatic transformations

To propose transformation rules for generating architectural model from requirements specification, we should decide what can be generated automatically and which decisions should be left for software analyst/architect. Some analytical decisions can be made on requirements level just before applying transformation on a model. Such decision is for instance grouping vocabulary notions into vocabulary packages and functional requirements into requirements packages. Basing on such a preprocessed model we can generate draft architectural model (details of what we can generate are presented below).

Generated model should be corrected and completed by software architect. Communication between analyst and architect is crucial for creating good software architecture reflecting user requirements.

The rules for automatic transformations should fulfil following criteria:

- the result architecture model should be conforming to good practices of creating such models (proper granularity of components, good ratio of interfaces per component and methods per interface, etc.)
- changes in source model should have predictable impact on result (target) model
- the rules should be flexible (customizable in a part at least)

In the following section we define source and target models for proposed transformation. The transformation rules are described and discussed in sections 6.2.1 – 6.2.4. In section 6.3 we present examples of transformation based on proposed rules.

6.2 Transformation rules for precise requirements model defined in RSL

While the source of proposed transformation is set as requirements specification written in RSL, the target should be also defined precisely. For that purpose the model of 4-layer architecture will be used. The significance of all tiers of 4-layer architectures is described in section 4.2.2 and transformation rules are discussed in section 6.2.1.

6.2.1 Generating multi-layer architecture from requirements structure

Having consistent requirement specification and transformation output model, we are able to generate draft system architecture. First activity we should perform is to structure requirement specification in packages. It consists of dividing vocabulary into packages of logically related Notions and dividing functional requirements into packages of functionally related UseCases.

Having above activity complete generate draft 4-layer architecture applying following rules (see Fig. 6.1, 6.2):

- every vocabulary package is transformed into one business component and one data access component
- every notion used in SVO sentence is transformed into data transfer object. Notions which occurs only in "high-level requirements" and system vision are ignored.
- for every notion used in SVO sentence, DAO interface is generated in corresponding data access component. This interface provide CRUD operations for given notion.
- notions in given vocabulary package are transformed into one interface provided by business component. Interface correspond to this vocabulary notion is generated only if the method for this notion is called on business component (in scenario we have self message with this notion in predicate).
- every functional requirements package is transformed into one logical component; it is important to keep number of functional requirements in packages low (for instance: less

than 4), to prevent creation of too complex components with too many interfaces (see also next rule)

- every UseCase (functional requirement) is transformed into one interface provided by logical component
- one UI component with one interface is generated for whole application; further division of this component can be done by an architect at later stage, when storyboards are ready and UIElements are placed in vocabulary
- all actors are transferred with no changes to architectural model

Currently if notion has "business" method, business interface for this notion is generated (one interface per notion).

Notions in vocabulary package could be "clustered". Such a group of notions could be transformed into one business interface. This decision could be also defer for architectural design. In such cases business interface will be generated as presented above.

6.2.2 Generating architectural details

To generate further architectural detail, sequence diagram from requirements specification needs to be transformed into sequence diagram for generated architecture.

In requirements sequence diagram we have several types of messages:

- initial actor predicate - the first message from an actor to the system
- actor's predicate - every message from actor to the system (Every sentence with actor as subject)
- system predicate - every message from the system to the actor (Every sentence with system as subject, followed by sentence with Actor as subject)
- System's "self-message" - every message from the system to itself (Every sentence with system as subject, followed sentence with system as subject)

Basing on above terminology we should apply following rules of transformation to create more detailed architectural model (see Fig. 6.3):

- initial actor predicate (description of an action that initiates a UseCase) is transformed into two calls: one from Actor to presentation layer component and the other one from presentation layer component to application logic component [1]
- each actor predicate is transformed analogously to initial actor's predicate
- every system predicate (description of an action that is a system's response to Actors activity) is transformed into call from logical layer component to presentation layer component
- every "invoke" construct is transformed into call from "current" logical layer component (the one for "current" UseCase) to logical layer component of invoked UseCase
- every System's "self-message" is transformed into call from logical layer component to business layer component; this business layer component corresponds to notion which occurs in predicate in object
- each of the above calls is transformed into a method of a related interface
- calls to business logic layer component's interface should correspond to verb phrases of Notion which forms this interface (see Fig. 6.4):
  - if verb phrase in predicate is a simple verb phrase, then verb is used for method's name, and object part of predicate is a method's parameter (example: [[v: add n: user]] => *add(User)*)
  - if phrase is a complex one, then verb and direct object form method's name and both direct and indirect objects are method's parameters (example: [[v: add n: user p: to n: user list]] => *addUser(User, UserList)*)
- actors in architectural models should be separated from actors from the requirements model (but they should have same names)
- calls to UI interface are asynchronous by default (but can be changed later by an architect to synchronous if needed)
- no returns are shown on architectural sequence diagrams
- methods do not return anything - return types should be set at later stage by an architect

[1]Both components are specific for this UseCase - see above

Naming of architectural model elements

Although for names of architectural model elements the RSL requirements model element names could be used, we propose also a set of rules for renaming elements resulting from notions, use cases, verb phrases, etc.:

- all element names should be converted to UpperCamelCase (aka PascalCase), e.g. *user list* => *UserList*
- in Business Logic layer all component names should have word “services” added at the end, e.g. *FacilityUses* => *FacilityUsesServices*
- in Business Logic layer all interface names should have word “service” added at the end, e.g. *FacilityDetails* => *FacilityDetailsService*
- in Data Access layer all component names should be in plural form and have words “data access” added at the end, e.g. *Device* => *DevicesDataAccess*
- in Data Access layer all interface names should have acronym “DAO” added at the end, e.g. *Account* => *AccountDAO*
- all notions (which form data transfer objects) should have names ending with acronym “DTO”, e.g. *RejectionMessage* => *RejectionMessageDTO*

These rules allow model names to be more conforming to the software naming standards. The rule set should be customisable in ReDSeeDS tool, as no design pattern or platform specific naming should be enforced on ReDSeeDS framework users.

6.2.3 Issue of traceability between generated model and model edited by an architect

The above transformations aim to generate platform independent model (PIM in the MDA terminology). The platform specific detailed design model can be generated/designed basing on a given architecture, which the architect should carefully examine and improve.

The important issue is a way in which an architect’s changes are marked in the final model, so re-generating the model again from requirements (for example: after requirement changes) would not overwrite an architect’s work. The solution could be based on keeping traces between generated model and final model. The example of such link could be IsAllocatedTo (see section

8.2.3 and figure 8.5). Also, changed elements in a model should be indicated with some kind of "change flag". During re-generation of architecture the architect would be prompted for overwriting their changes. Of course there are still few problems to solve: for example if architect splits component into two new components, and after generation a new method appears in original component, the decision has to be taken: to which of the resulting two components the new method should be added.

6.2.4 Other issues concerning automatic architectural model generation

Presented transformation rules are appropriate for generating 4-layer architecture. For other architectures (like for the real time systems, service-oriented architectures or client-server architecture) different transformation rules should be proposed. In fact we need a "family" of transformations which probably should be customisable.

While 4-layer architecture seems to be most illustrative for purpose of creating transformation rules, transforming to other architectures following layer architectural design patterns could be based on rules presented above; e.g. transforming to 3-layer architecture could be very similar with exception of merging UI and application logic layers (calls between UI and application logic become UI layer self-messages).

One of the decisions, that the architect has to make, is a declaration of methods for implementation of collections in DAOs. Most of them cannot be generated automatically, because *getList* type methods often take some kind of list filtering criteria, which is not easy to determine during transformation from requirements to architecture.

The generated architecture model, which is platform independent, in the next step will be transformed to platform specific model. Depending on the technology stack used on a given platform, different transformation profiles could be used for that purpose – for instance, while using Hibernate as persistence layer, Data Access Objects (DAOs) and Data Transfer Objects (DTOs) should be converted to appropriate DAO interfaces, Java classes (POJOs) and hbm files.

Another issue is the placement of hardware used by systems on the appropriate layer - it can be either the user interface layer (when user interacts with hardware) or datastorage/hardware layer, if hardware is used for storing/retrieving data.

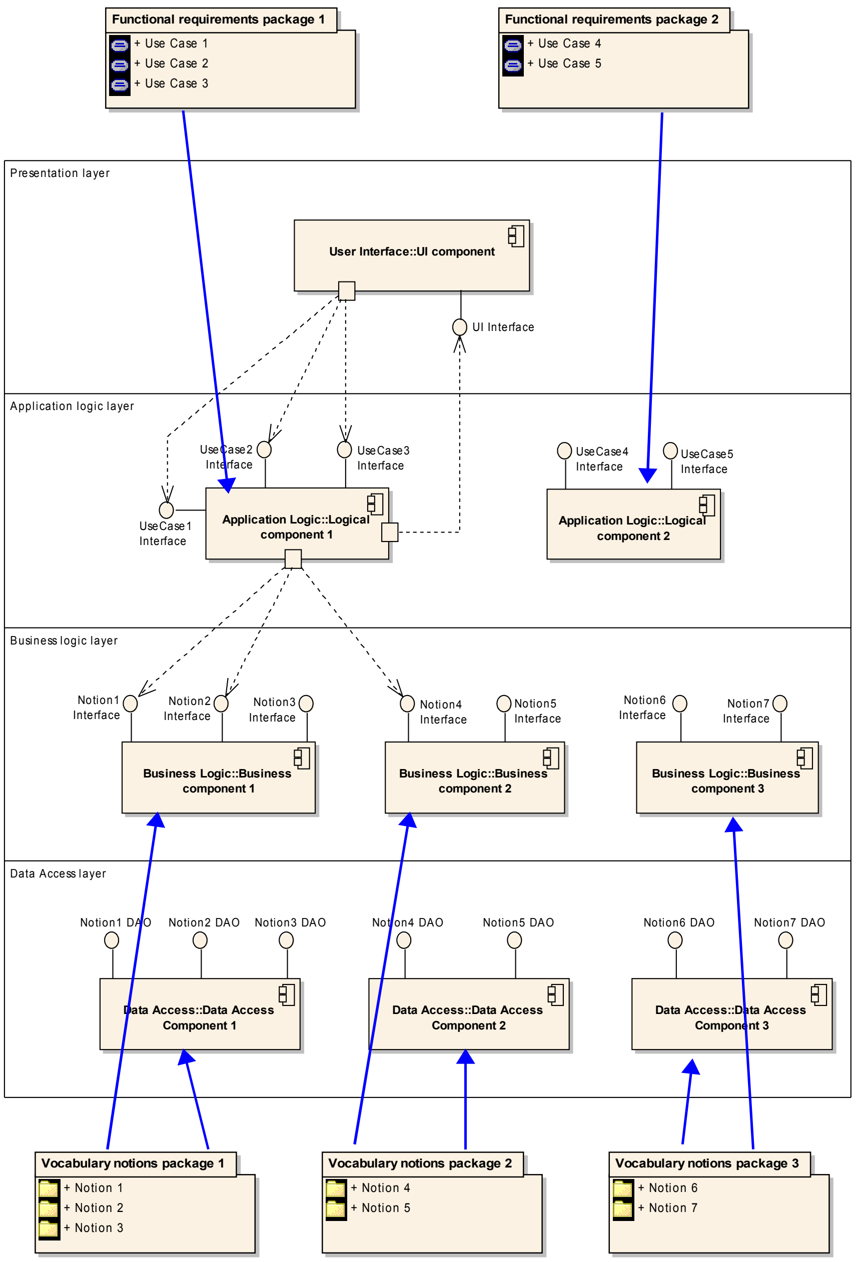


Figure 6.1: 4-layer architecture generation overview

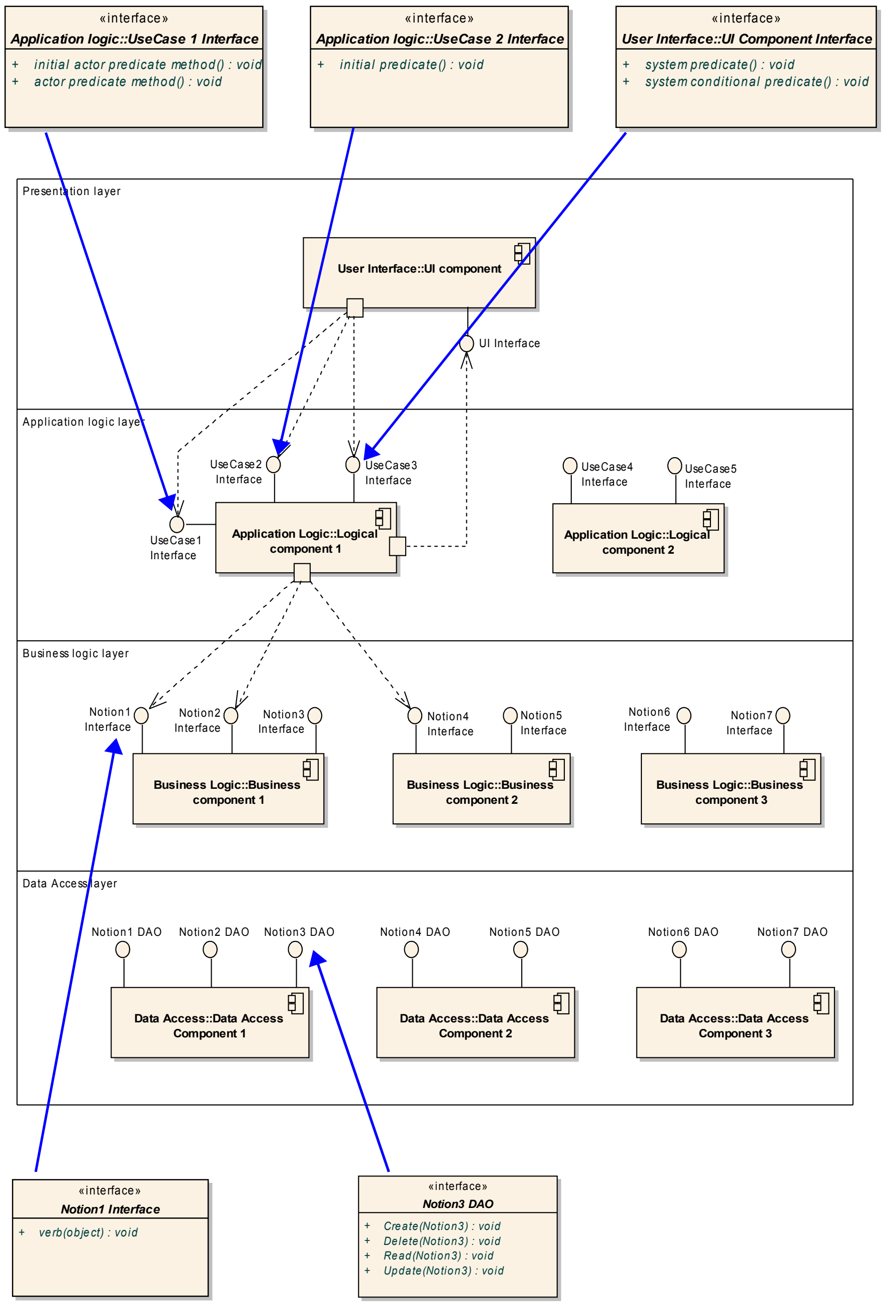

«interface»
Application logic::UseCase 1 Interface
+ initial actor predicate method() : void
+ actor predicate method() : void
«interface»
Application logic::UseCase 2 Interface
+ initial predicate() : void
«interface»
User Interface::UI Component Interface
+ system predicate() : void
+ system conditional predicate() : void
Presentation layer
User Interface::UI component
UI Interface
Application logic layer
UseCase1 Interface
UseCase2 Interface
UseCase3 Interface
UseCase4 Interface
UseCase5 Interface
Application Logic::Logical component 1
Application Logic::Logical component 2
Business logic layer
Notion1 Interface
Notion2 Interface
Notion3 Interface
Notion4 Interface
Notion5 Interface
Notion6 Interface
Notion7 Interface
Business Logic::Business component 1
Business Logic::Business component 2
Business Logic::Business component 3
Data Access layer
Notion1 DAO
Notion2 DAO
Notion3 DAO
Notion4 DAO
Notion5 DAO
Notion6 DAO
Notion7 DAO
Data Access::Data Access Component 1
Data Access::Data Access Component 2
Data Access::Data Access Component 3
«interface»
Notion1 Interface
+ verb(object) : void
«interface»
Notion3 DAO
+ Create(Notion3) : void
+ Delete(Notion3) : void
+ Read(Notion3) : void
+ Update(Notion3) : void

Figure 6.2: 4-layer architecture generation - interfaces

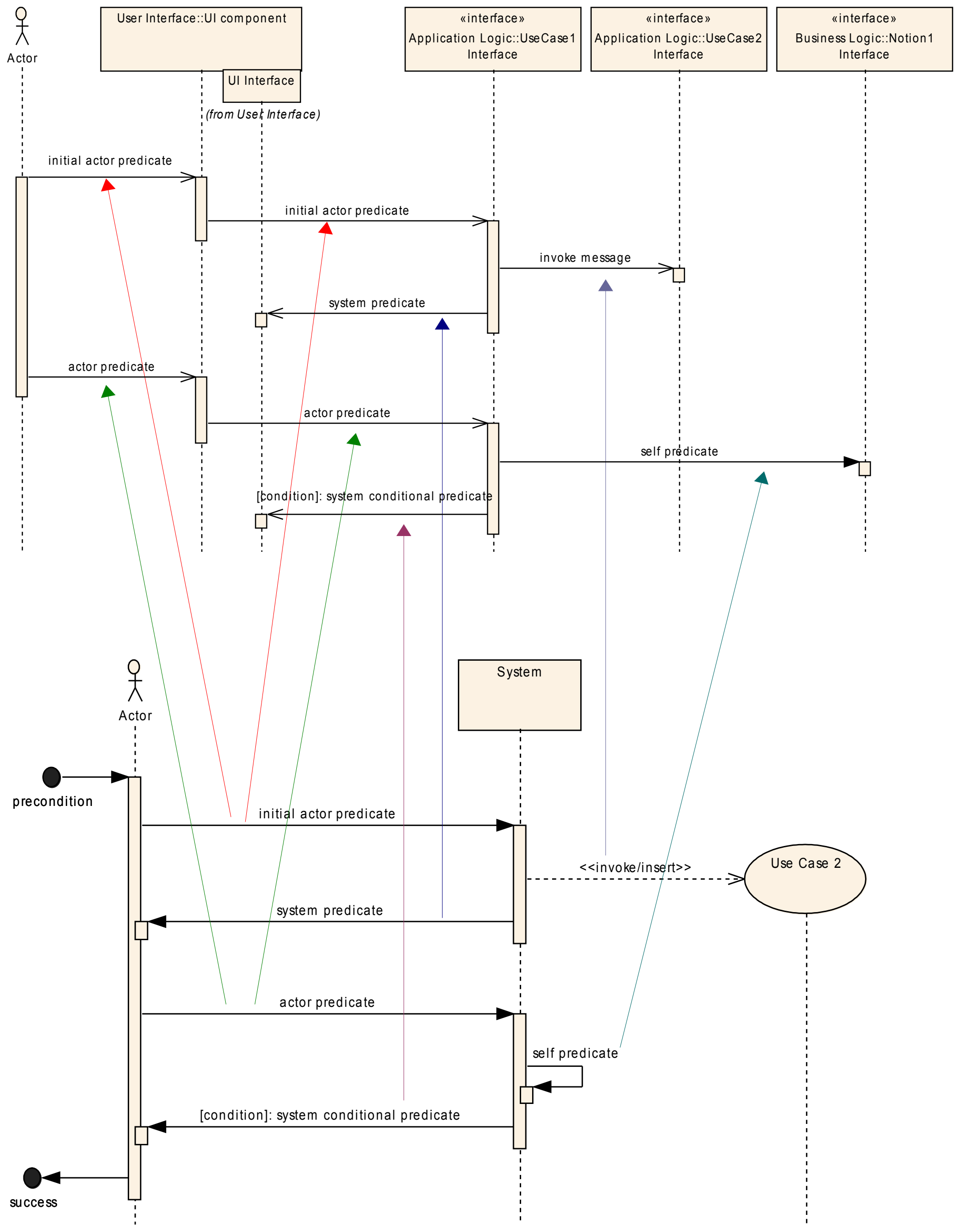


Figure 6.3: 4-layer architecture generation - sequence diagrams

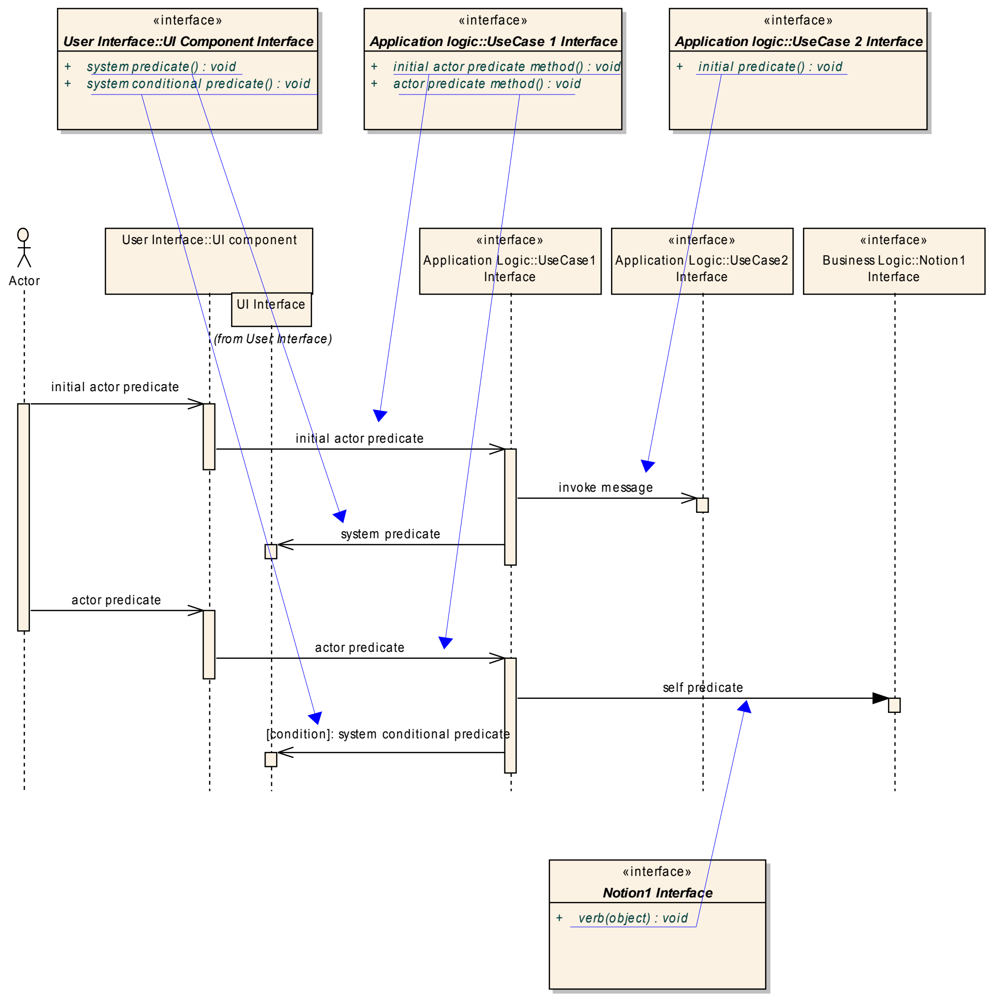

«interface»
User Interface::UI Component Interface
+ system predicate() : void
+ system conditional predicate() : void
«interface»
Application logic::UseCase 1 Interface
+ initial actor predicate method() : void
+ actor predicate method() : void
«interface»
Application logic::UseCase 2 Interface
+ initial predicate() : void
Actor
User Interface::UI component
UI Interface
(from User Interface)
«interface»
Application Logic::UseCase1 Interface
«interface»
Application Logic::UseCase2 Interface
«interface»
Business Logic::Notion1 Interface
initial actor predicate
initial actor predicate
invoke message
system predicate
actor predicate
actor predicate
self predicate
[condition]: system conditional predicate
«interface»
Notion1 Interface
+ verb(object) : void

Figure 6.4: 4-layer architecture generation - methods of interfaces

6.3 Examples of possible model transformations in SCL

In this section an example usage of transformation rules from section 6.2 will be presented. The rules have been applied in an automatic way, but without using any scripting/transformation tool (by manual edition of architectural model).

The source for presented transformation was the RSL Elaborate Example, which is a complete requirements specification written in the RSL for imaginary Fitness Club software. The specification contains 36 requirements – high-, low-level and non-functional, among them 12 Use Cases, which have several scenarios. The vocabulary part consists of 45 notions with 122 domain statements.

The output of the example transformation is divided into 4-layer architecture as described in example from 4.2.2. The resulting model contains 16 components with 62 interfaces having 213 methods and using 40 data transfer objects (DTOs). This gives satisfactory results from the perspective of method-per-interface and interface-per-component ratios. Please note that most of the components and interfaces exist in Data Access layer, which also has the most complex components (in terms of number of interfaces).

The components with interfaces for all layers are presented on Figures 6.6 – 6.9.

Example also contains few sequence diagrams for selected use cases, that are shown on figures 6.10 – 6.15.

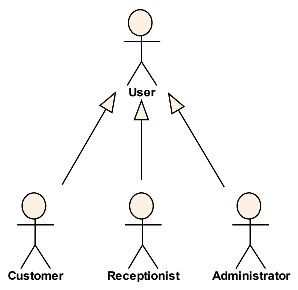


Figure 6.5: 4-layer architecture model example - actors

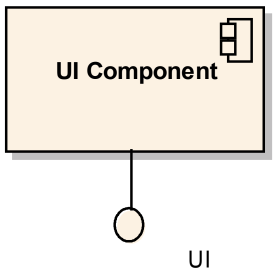


Figure 6.6: 4-layer architecture model example - Presentation layer components with interfaces

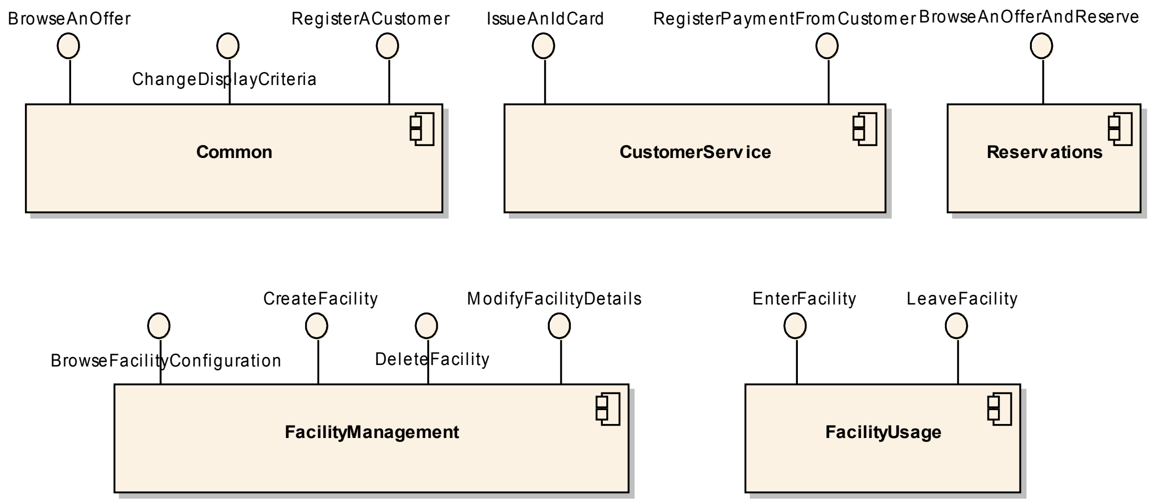


Figure 6.7: 4-layer architecture model example - Application Logic layer components with interfaces

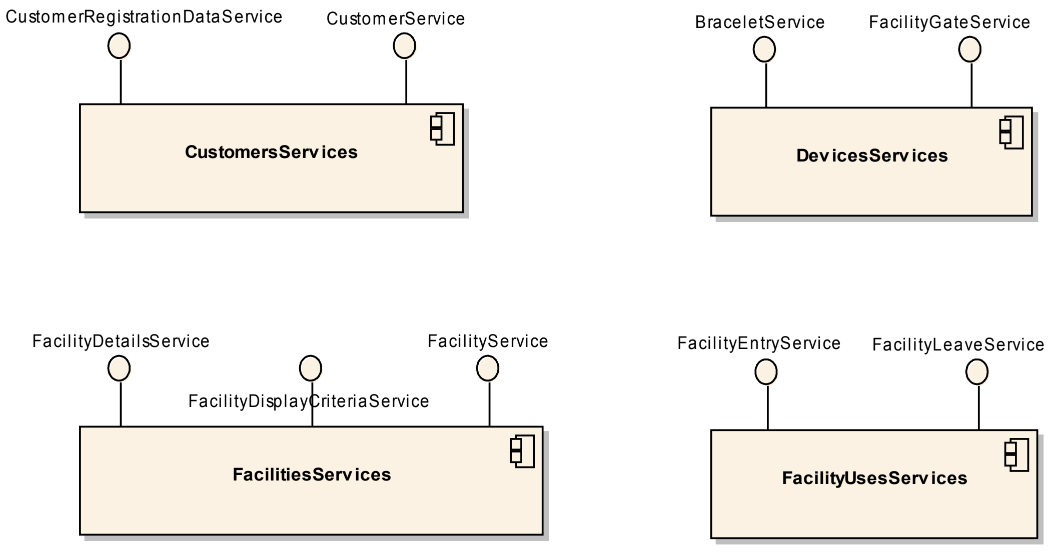


Figure 6.8: 4-layer architecture model example - Business Logic layer components with interfaces

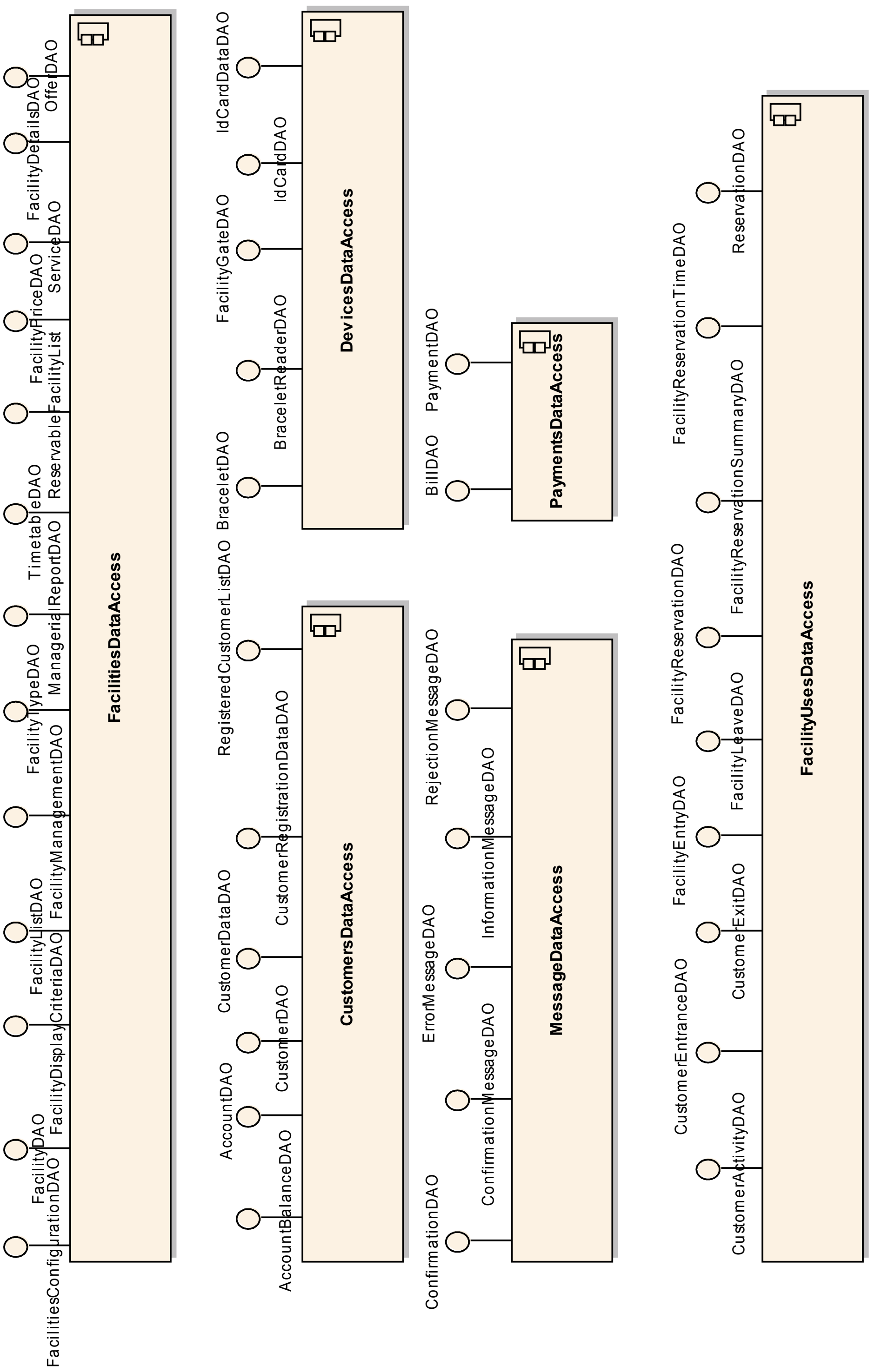


Figure 6.9: 4-layer architecture model example - Data Access layer components with interfaces

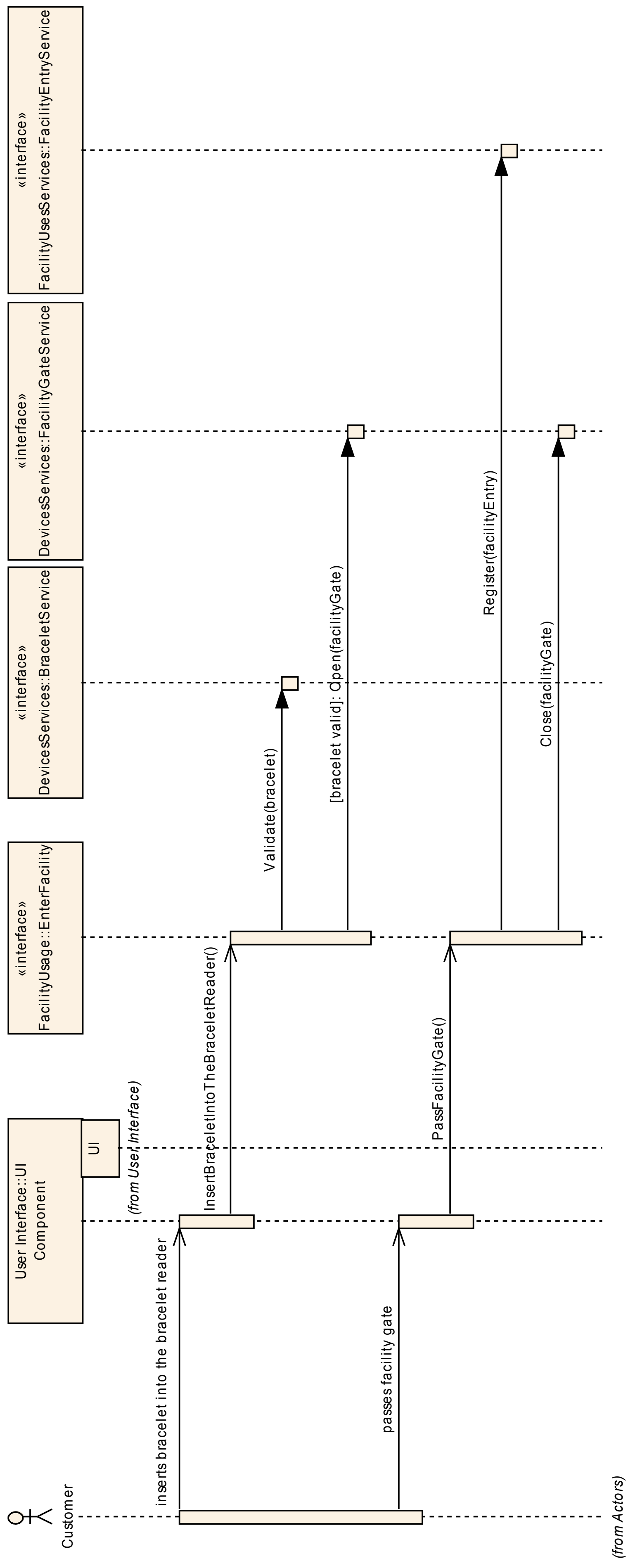

Customer
(from Actors)
User Interface::UI Component
UI
(from User Interface)
«interface» FacilityUsage::EnterFacility
«interface» DevicesServices::BraceletService
«interface» DevicesServices::FacilityGateService
«interface» FacilityUsesServices::FacilityEntryService
inserts bracelet into the bracelet reader
InsertBraceletIntoTheBraceletReader()
Validate(bracelet)
[bracelet valid]: Open(facilityGate)
passes facility gate
PassFacilityGate()
Register(facilityEntry)
Close(facilityGate)

Figure 6.10: 4-layer architecture model example - sequence diagram for basic path scenario of Enter Facility Use Case

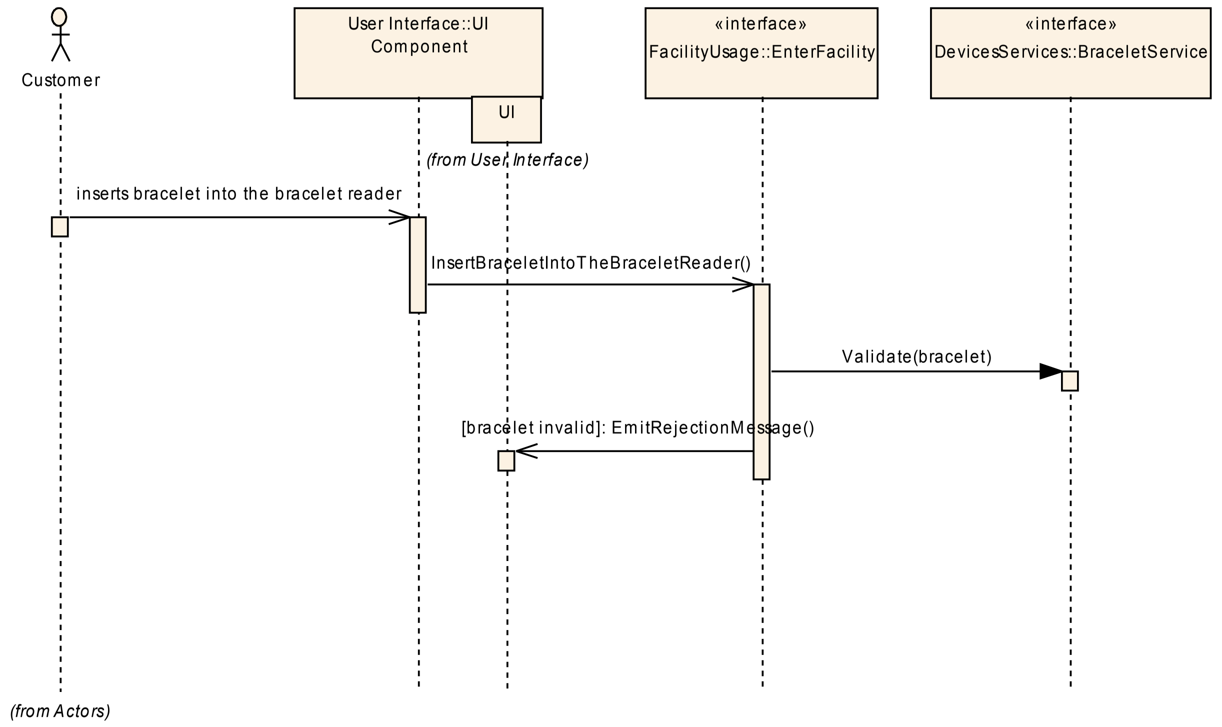

Customer
(from Actors)
User Interface::UI Component
UI
(from User Interface)
«interface»
FacilityUsage::EnterFacility
«interface»
DevicesServices::BraceletService
inserts bracelet into the bracelet reader
InsertBraceletIntoTheBraceletReader()
Validate(bracelet)
[bracelet invalid]: EmitRejectionMessage()

Figure 6.11: 4-layer architecture model example - sequence diagram for 1st alternate path scenario of Enter Facility Use Case

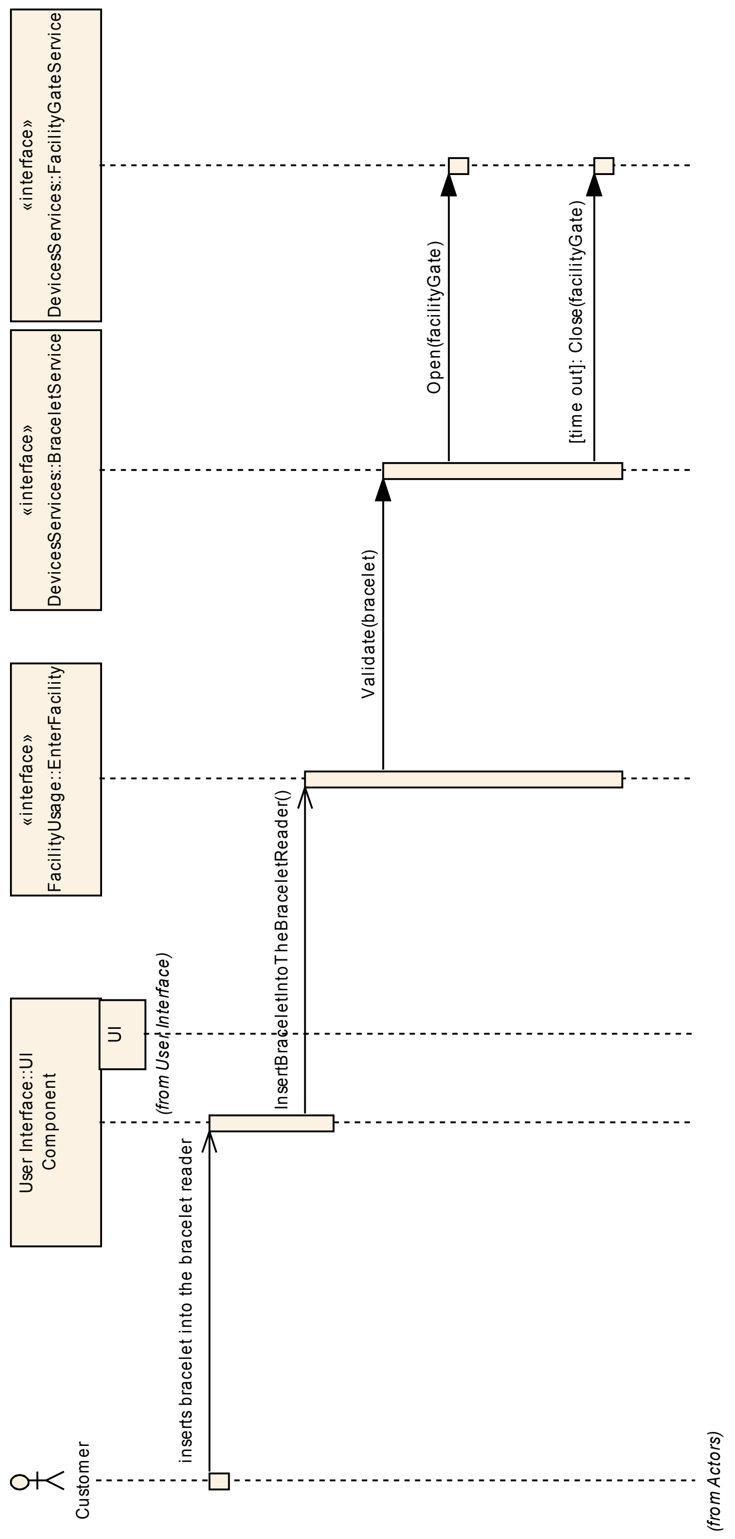

Customer
(from Actors)
User Interface::UI Component
UI
(from User Interface)
«interface» FacilityUsage::EnterFacility
«interface» DevicesServices::BraceletService
«interface» DevicesServices::FacilityGateService
inserts bracelet into the bracelet reader
InsertBraceletIntoTheBraceletReader()
Validate(bracelet)
Open(facilityGate)
[time out]: Close(facilityGate)

Figure 6.12: 4-layer architecture model example - sequence diagram for 2nd alternate path scenario of Enter Facility Use Case

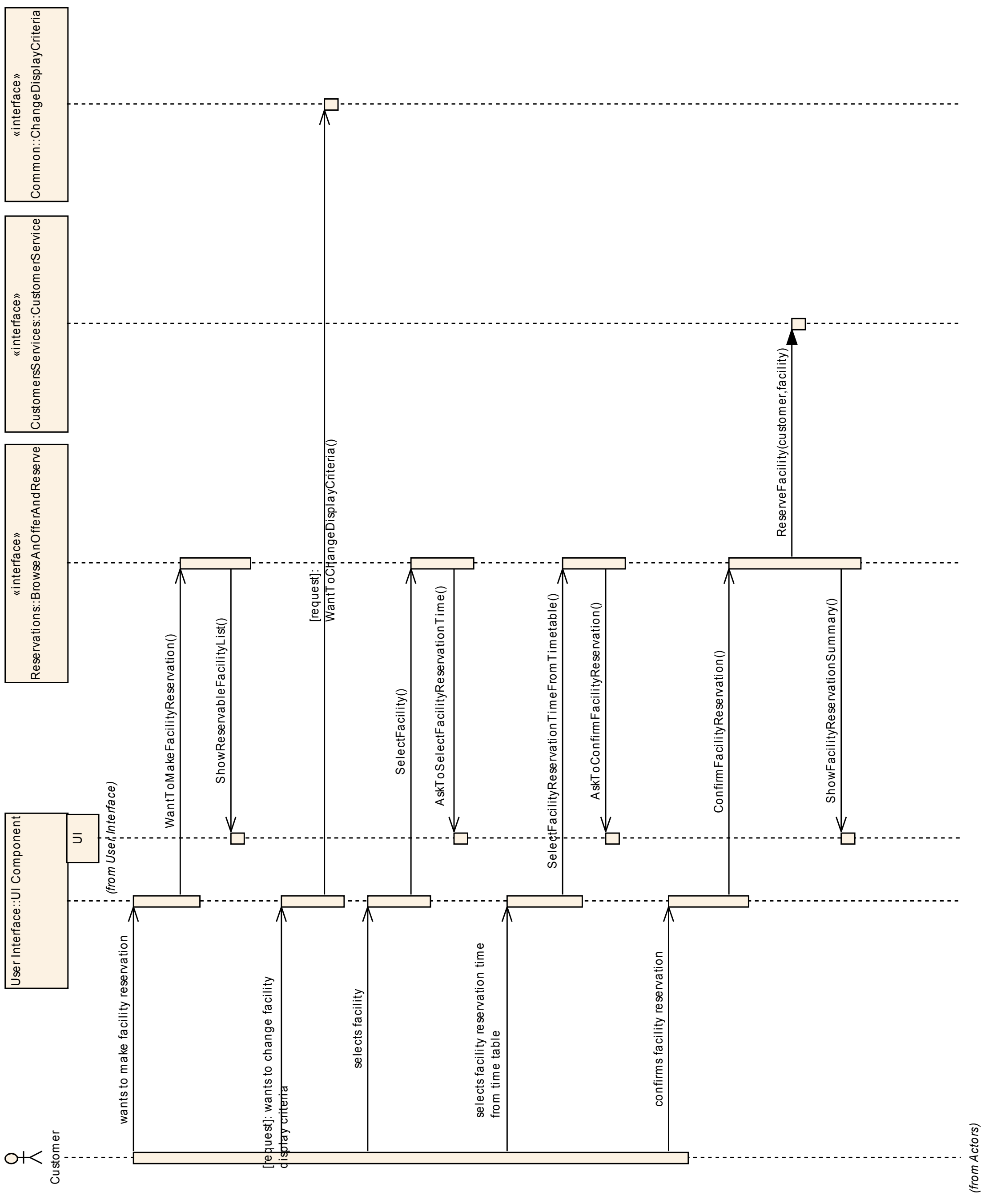

«interface»
Common::ChangeDisplayCriteria
«interface»
CustomersServices::CustomerService
«interface»
Reservations::BrowseAnOfferAndReserve
User Interface::UI Component
UI
(from User Interface)
Customer
(from Actors)
wants to make facility reservation
WantToMakeFacilityReservation()
ShowReservableFacilityList()
[request]: wants to change facility display criteria
[request]:
WantToChangeDisplayCriteria()
selects facility
SelectFacility()
AskToSelectFacilityReservationTime()
selects facility reservation time from time table
SelectFacilityReservationTimeFromTimetable()
AskToConfirmFacilityReservation()
confirms facility reservation
ConfirmFacilityReservation()
ReserveFacility(customer,facility)
ShowFacilityReservationSummary()

Figure 6.13: 4-layer architecture model example - sequence diagram for basic path scenario of Browse An Offer And Reserve Use Case

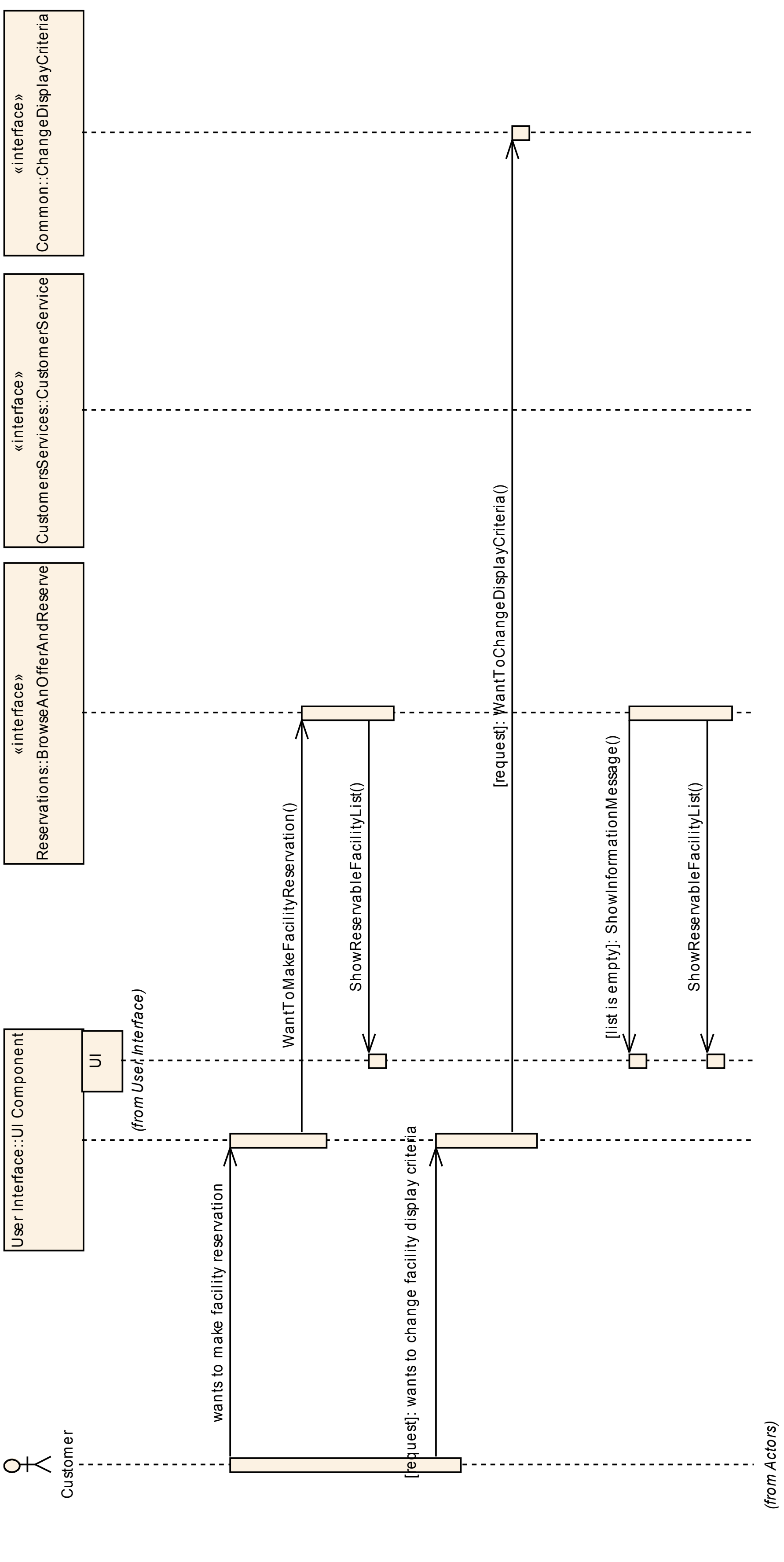

Customer
(from Actors)
User Interface::UI Component
UI
(from User Interface)
«interface» Reservations::BrowseAnOfferAndReserve
«interface» CustomersServices::CustomerService
«interface» Common::ChangeDisplayCriteria
wants to make facility reservation
WantToMakeFacilityReservation()
ShowReservableFacilityList()
[request]: wants to change facility display criteria
[request]: WantToChangeDisplayCriteria()
[list is empty]: ShowInformationMessage()
ShowReservableFacilityList()

Figure 6.14: 4-layer architecture model example - sequence diagram for 1st alternate path scenario of Browse An Offer And Reserve Use Case

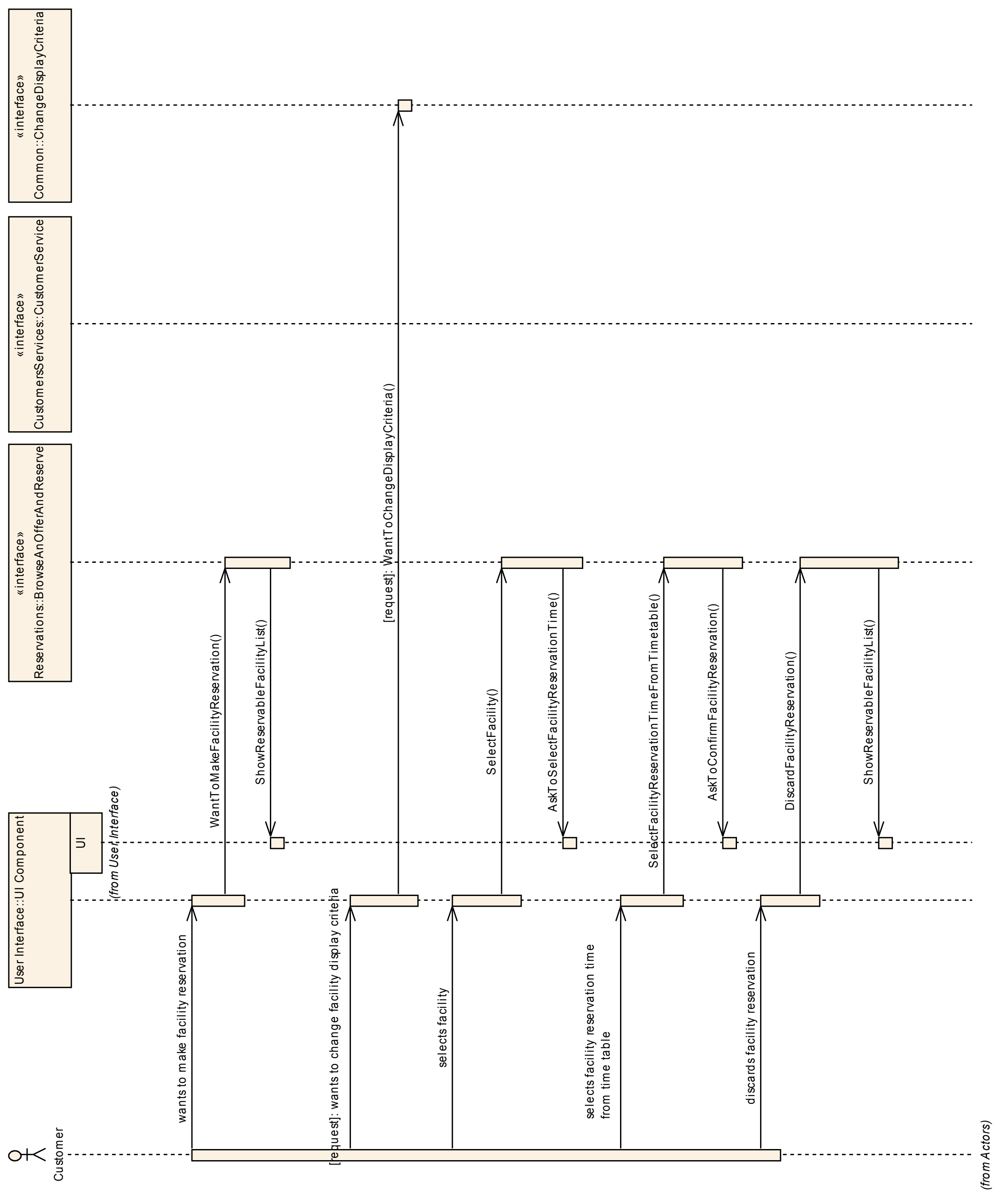

Customer
(from Actors)
User Interface::UI Component
UI
(from User Interface)
«interface»
Reservations::BrowseAnOfferAndReserve
«interface»
CustomersServices::CustomerService
«interface»
Common::ChangeDisplayCriteria
wants to make facility reservation
WantToMakeFacilityReservation()
ShowReservableFacilityList()
[request]: wants to change facility display criteria
[request]: WantToChangeDisplayCriteria()
selects facility
SelectFacility()
AskToSelectFacilityReservationTime()
selects facility reservation time
from time table
SelectFacilityReservationTimeFromTimetable()
AskToConfirmFacilityReservation()
discards facility reservation
DiscardFacilityReservation()
ShowReservableFacilityList()

Figure 6.15: 4-layer architecture model example - sequence diagram for 2nd alternate path scenario of Browse An Offer And Reserve Use Case

6.4 UI model-to-model transformation example

This section illustrates how a UI specification in RSL can be transformed into a graphical UI model, that can be implemented by a GUI toolkit like Java Swing. The source for this transformation is, more precisely, a UIBehaviourRepresentations::UIStoryboard together with a scenario in constrained language representation and static representation by UIElements::UIElements.

The target of the transformation is a GUI model conforming to a conceptual GUI meta-model describing a GUI toolkit. The conceptual GUI meta-model presented here is based on the Java Swing Model but is generalised in a way to be easily adaptable to other GUI toolkits like Windows Forms (e.g., we omit the "J" in front of each class name). It should be mentioned that the GUI meta-model presented here is not part of the RSL and the SCL. According to the transformation described in section 6.2, the target GUI model is a more detailed design of the User Interface::UI component residing in the presentation layer.

We use the "Making facility reservation" scenario of our "Fitness club" example to illustrate the transformation. This transformation example is explained in detail below in section 6.4.3.

The subsequent sections are organised as follows. First, we explain the conceptual GUI meta-model that the transformation results are based upon. Second, we describe some essential transformation rules that generate the GUI elements out of the UI requirements specification. Finally, we apply the transformation rules to the "Make facility reservation" scenario and a corresponding storyboard given in RSL, for illustration purposes.

6.4.1 Conceptual GUI meta-model

The conceptual GUI meta-model explained in this section provides a possible way for specifying the structure of the User Interface::UI component and the GUI, respectively. It organises all GUI widgets in the form of a tree as common to current GUI toolkits. The conceptual GUI meta-model is represented as a UML class diagram. It also supports the possibility to establish *traceability links*.

Figure 6.16 shows part of the conceptual GUI meta-model relevant to this transformation example. The classes and structure of the meta-model are mainly based on Java Swing. The base class for all widgets is the abstract class Component. The specialised classes Window, Dialog, Frame and Panel in the left part of Figure 6.16 depict GUI containers that aggregate other widgets. Since in Java Swing each component is also a container, the conceptual GUI meta-model

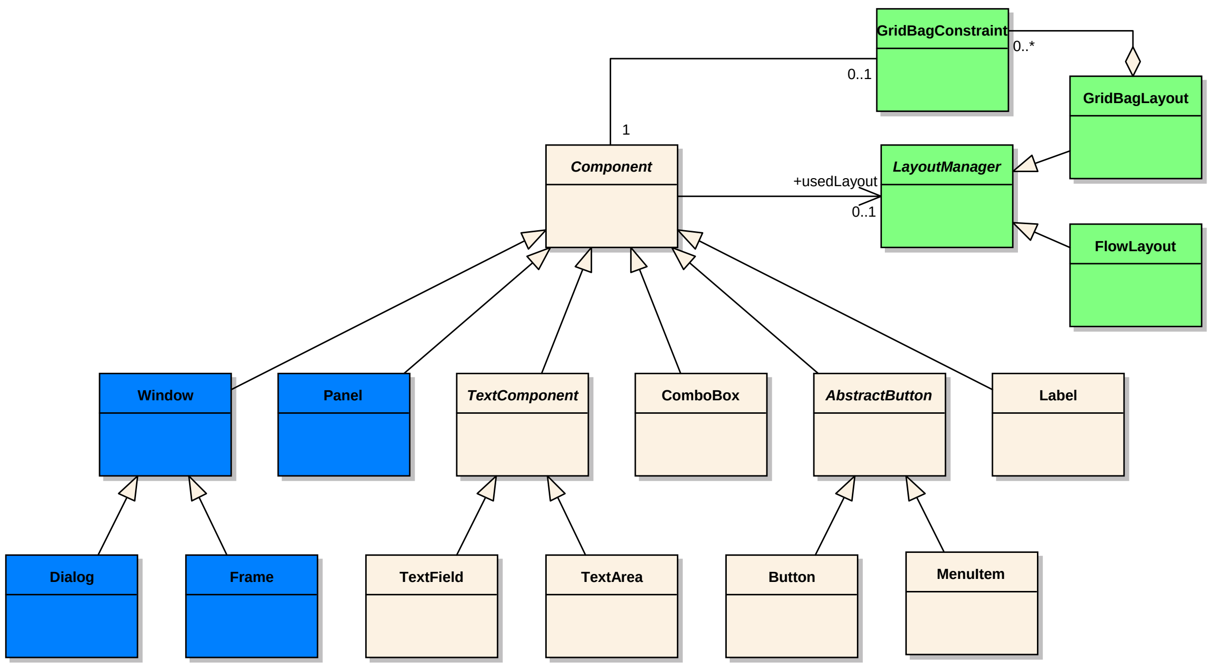


Figure 6.16: A conceptual GUI meta-model

presented here also does not explicitly distinguish between container and non-container elements. Conforming to Java Swing, all GUI widgets can be associated with a LayoutManager, but usually it makes sense to attach layout managers only to GUI containers. Some layout managers like GridBagLayout assign additional constraints (e.g., row, column, padding, . . .) to the GUI widgets aggregated in the associated GUI container. Other layout managers like FlowLayout only depend on the order the widgets are added to the container. In the resulting GUI models we specify the order by adding tagged values to the objects in the object diagrams.

6.4.2 GUI transformation rules

Applying the following transformation rules to the UI specification written in RSL results in a detailed design for the UI Interface::UI component put onto the presentation layer and the component's behaviour.

1. The UIBehaviourRepresentations::UIStoryboard results in a finite state machine according to the State Design pattern.
2. Every UIBehaviourRepresentations::UIScene results in a state assigned to the created finite state machine.
3. Every UIElements::UIPresentationUnit contained in a UIBehaviourRepresentations::UIScene results in a modal Dialog or a Frame, depending on whether the

UI scene attached ScenarioSentences::SVOScenarioSentence contains the modifier "modal" or not. The created Dialog or Frame is assigned to the state belonging to the containing UI scene.

4. Every UIBehaviourRepresentations::UserAction results in a transition. The source of the transition is the state belonging to the predecessor UI scene and the target of the transition is the state belonging to the successor UI scene.

5. Every UIElements::TriggerUIElement associated with a UIBehaviourRepresentations::UserAction results in the corresponding specialisation of AbstractButton and an action method assigned to the button that triggers the state transition associated with UIBehaviourRepresentations::UserAction. If the SVOSentences::SVOSentence associated with the UIBehaviourRepresentations::UserAction is a ConditionalSentence, the condition has to be evaluated (e.g., through a manually implemented method) and its result is used to select alternative transitions.

The content of a window is determined by the following rules. If a detailed specification of a UIElements::UIPresentationUnit is available, the following rules apply:

1. Every UIElements::UIContainer is transformed into a Panel. By default, a FlowLayout is associated with the Panel. All UIElements::UIElement contained in the UIElements::UIContainer have to be processed in the order defined by the associated UIElements::PresentationOrder. The following rules are applied to the contained UIElements::UIElement. Each generated widget is added to the Panel.

2. Every UIElements::InputUIElement is transformed into a TextField. If it is possible to determine the type of data through an associated ScenarioSentences::SVOScenarioSentence, it is possible to generate a text area, spinner, date/time picker, gauge or slider instead of a text field.

3. Every UIElements::SelectionUIElement is transformed into either a ComboBox, ListBox, CheckBox or RadioButtons depending on the number of contained OptionUIElement and if multiple selection is allowed or not.

4. Every UIElements::TriggerUIElement are transformed according to the rule described above.

If no detailed specification of a UIElements::UIPresentationUnit is available, the following rules can be used to generate a first prototype for the UI Interface::UI component that needs to be manually reworked by the designer:

1. Every SVOSentences::SVOSentence attached to a UIBehaviourRepresentations::UIScene where the subject refers to the system and the verb denotes a required selection by the user like "offer" is transformed into a Label representing the object and either a ComboBox, ListBox, CheckBox or RadioButtons depending on the number of contained data and if multiple selection is allowed or not.

2. Every SVOSentences::SVOSentence attached to a UIBehaviourRepresentations::UIScene where the subject refers to the system and the verb denotes gathering information from the user like "ask" or "request" is transformed into a Panel containing a list of pairs of Label and input widgets. These widgets represent the domain information referred to by the object of the sentence. The name of the domain element is transformed into the Label and the type of the domain element determines the type of the input widget. By default, the type of the input widget is TextField allowing to enter any kind of information.

3. Every SVOSentences::SVOSentence attached to a UIBehaviourRepresentations::UIScene where the subject refers to the system and the verb denotes the presentation of information like "inform" is transformed into a Label representing the information denoted by the object of the sentence.

4. Every SVOSentences::SVOSentence attached to a UIBehaviourRepresentations::UserAction where the subject refers to the user of the system results in the creation of a Button that allows submission of information that the user entered based on other SVO sentences attached to predecessor UIBehaviourRepresentations::UIScene of the UIBehaviourRepresentations::UserAction. If there is also a ConditionalSentence associated with the UIBehaviourRepresentations::UserAction, the condition evaluation is also included in the action method of the generated button.

6.4.3 Transformation example

For the purpose of illustration, these transformation rules can be applied to the "Making facility reservation" scenario and the corresponding UIBehaviourRepresentations::UIStoryboard as follows. Figure 6.17 shows the constrained language representation of the scenario "Making facility reservation". The model is based on the SVOSentences metamodel. The invoked use cases "Select facility" and "Pay Fee" are treated as ScenarioSentences::SVOScenarioSentences in the scenario. Figure 6.18 shows the UIBehaviourRepresentations::UIStoryboard corresponding to the "Making facility reservation" scenario.

source	view
pre: [[FCS]] offers [[services]]	pre: FCS offers services.
1. [[n:Customer]] [[v:selects n:service p:for n:reserving facility]]	1. Customer selects service for reserving facility.
2. [[n:FCS]] offers [[n:facilities]]	2. FCS offers facilities.
3. [[n:Customer]] [[v:selects n:facility]]	3. Customer selects facility.
4. [[n:FCS]] [[v:requests n:date and time]]	4. FCS requests date and time.
5. [[n:Customer]] [[v:selects n:date and time]]	5. Customer selects date and time.
6. [[n:FCS]] [[v:checks n:facility availability]]	6. FCS checks facility availability.
==> cond:[[facility]] is available	➔ cond: facility is available
7. [[n:FCS]] [[v:requests n:user identification]]	7. FCS requests user identification.
8. [[n:Costumer]] [[v:identifies n:herself/himself]]	8. Costumer identifies herself/himself.
9. [[n:FCS]] [[v:checks n:user identity]]	9. FCS checks user identity
==> invoke/insert: [[v:Pay n:fee]]	➔ invoke/insert: Pay fee
10.[[n:FCS]] [[v:books n:facility]]	13. FCS books facility.
11.[[n:FCS]] [[v:informs p:about n:reservation status]].	14. FCS informs about reservation status.
post: [[Customer]] has reserved a [[facility]] and the [[facility]] is booked.	post: Customer has reserved a facility and the facility is booked.

Figure 6.17: Constrained language representation of scenario "Making facility reservation"

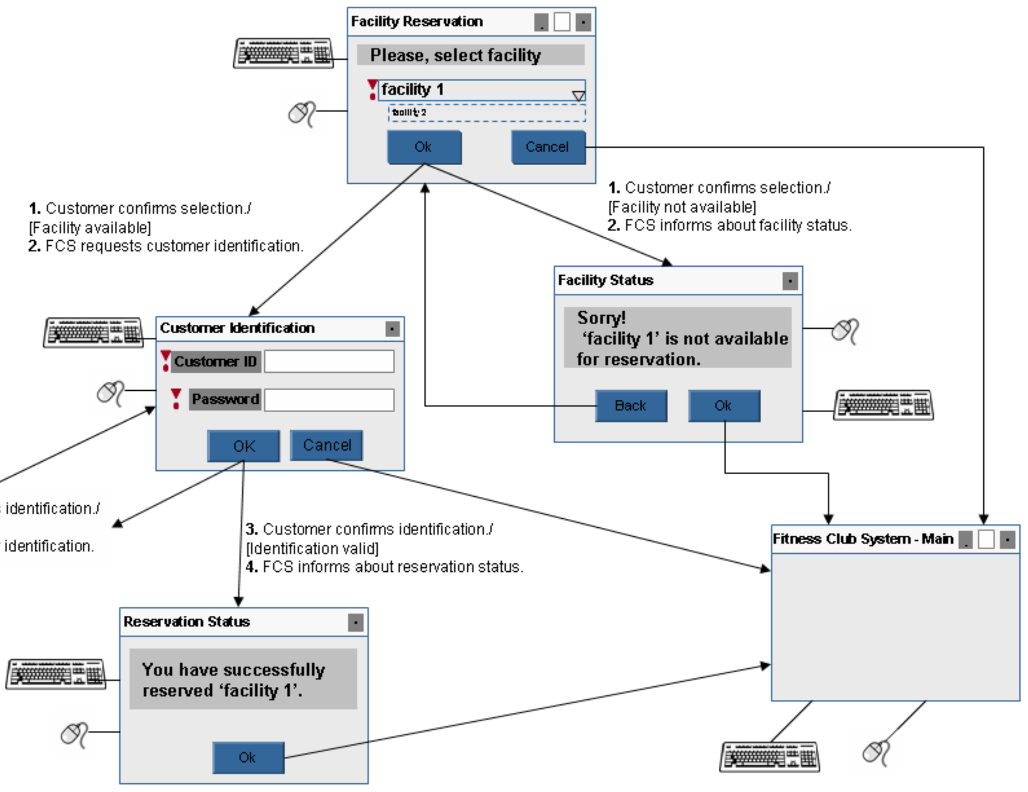

Facility Reservation
Please, select facility
facility 1
facility 2
Ok
Cancel
1. Customer confirms selection./
[Facility available]
2. FCS requests customer identification.
1. Customer confirms selection./
[Facility not available]
2. FCS informs about facility status.
Facility Status
Sorry!
'facility 1' is not available
for reservation.
Back
Ok
Customer Identification
Customer ID
Password
OK
Cancel
3. Customer confirms identification./
[Identification invalid]
2. FCS requests user identification.
3. Customer confirms identification./
[Identification valid]
4. FCS informs about reservation status.
Fitness Club System - Main
Reservation Status
You have successfully
reserved 'facility 1'.
Ok

Figure 6.18: UIBehaviourRepresentations::UIStoryboard for "Making facility reservation" Scenario

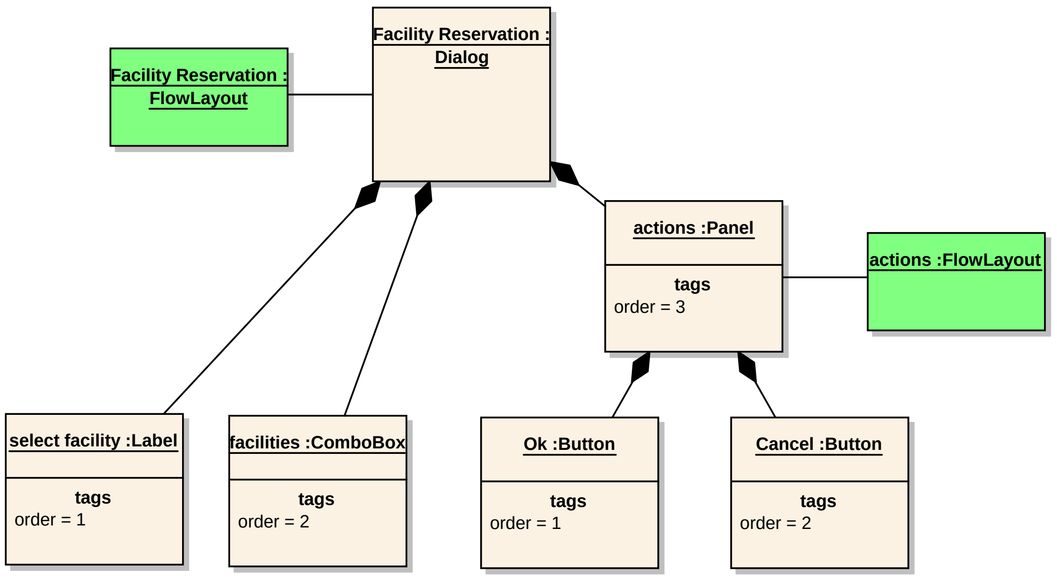


Figure 6.19: Object diagram for Facility Reservation

Every UIBehaviourRepresentations::UIScene of the UIBehaviourRepresentations::UIStoryboard corresponds to an ScenarioSentences::SVOScenarioSentence of the scenario. The described transformation rules are applied on both models, the SVOScenarioSentences and the UIBehaviourRepresentations::UIStoryboard. Figure 6.19 illustrates the GUI Model for "Facility reservation" which results from ScenarioSentences::SVOScenarioSentence number 2 and the corresponding UIBehaviourRepresentations::UIScene in the UIBehaviourRepresentations::UIStoryboard. It represents the static structure of the UIBehaviourRepresentations::UIScene "Facility reservation". The Dialog is composed of a Label, ComboBox and a Panel, which contains two Buttons. The "PresentationOrder" is transformed to instances of FlowLayout assigned to Panel and Dialog. The order is defined by the tagged value "order". The lower the number the more to the left it is positioned in the Dialog. If there is not enough space in one line it is positioned in the next line.

Figure 6.20 illustrates the GUI Model for "Customer identification" which results from ScenarioSentences::SVOScenarioSentence number 7 and the corresponding UIBehaviourRepresentations::UIScene in the UIBehaviourRepresentations::UIStoryboard. It represents the static structure of the UIBehaviourRepresentations::UIScene "Customer identification". The Dialog is composed of 2 Labels, 2 TextFields and a Panel, which contains two Buttons. The Dialog and the Panel are arranged with the FlowLayout.

Figure 6.21 illustrates the GUI Model for "Reservation status" which results from ScenarioSentences::SVOScenarioSentence number 14 and the corresponding UIBehaviourRepresentations::UIScene in the UIBehaviourRepresentations::UIStoryboard. It represents the static

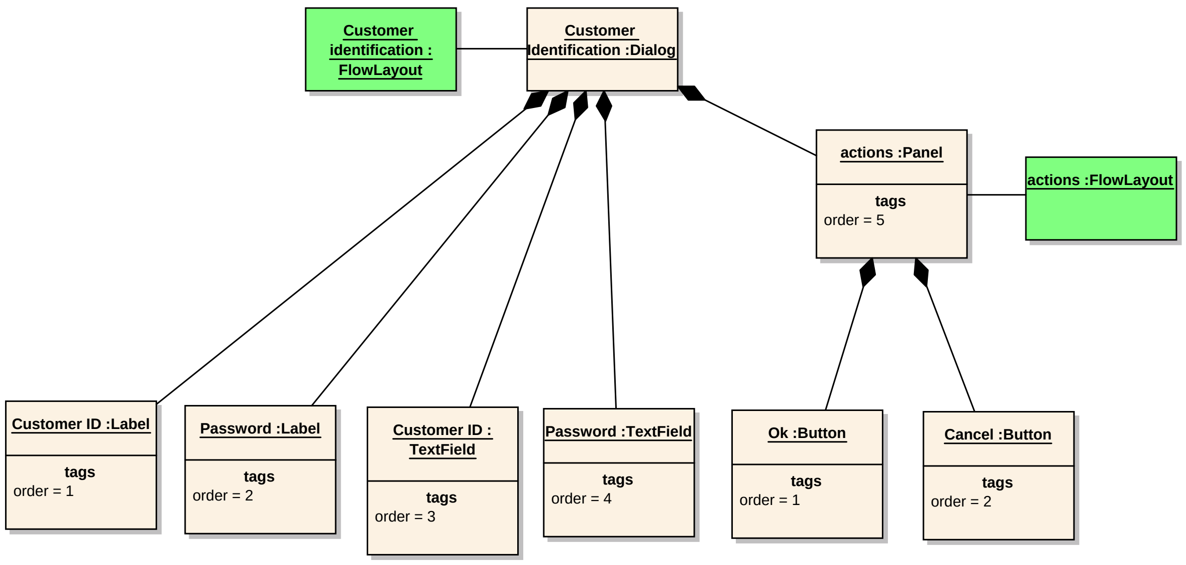


Figure 6.20: Object diagram for customer identification

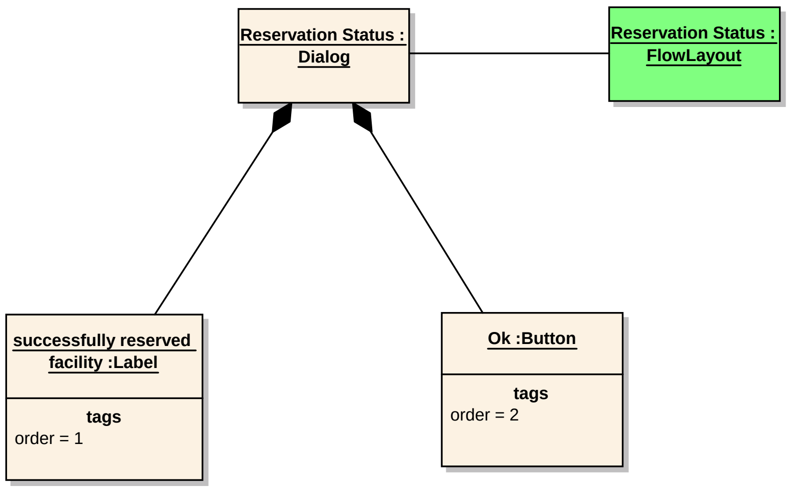

Reservation Status : Dialog
Reservation Status : FlowLayout
successfully reserved facility :Label
tags
order = 1
Ok :Button
tags
order = 2

Figure 6.21: Object diagram for reservation status

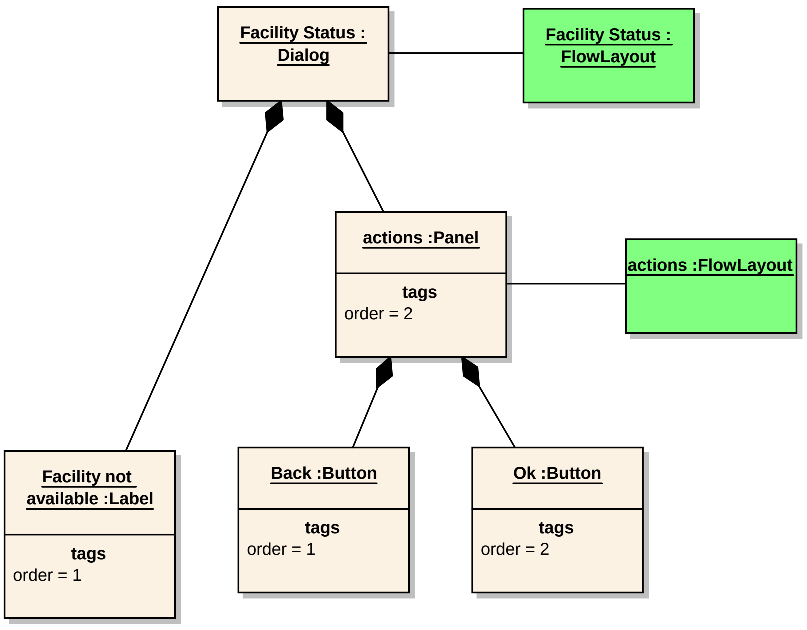


Figure 6.22: Object diagram for facility status

structure of the UIBehaviourRepresentations::UIScene "Reservation status". The Dialog is composed of a Label and a Button arranged with the FlowLayout.

Figure 6.22 illustrates the GUI Model for "Facility status" which results from a ScenarioSentences::SVOScenarioSentence conveying the condition "facility not available" and the corresponding UIBehaviourRepresentations::UIScene in the UIBehaviourRepresentations::UIStoryboard. It represents the static structure of the UIBehaviourRepresentations::UIScene "Facility status". The Dialog is composed of a Label and a Panel, which contains two Buttons. The Dialog and the Panel is arranged with the FlowLayout.

Figure 6.23 illustrates the GUI Model for "Facility club system" which results from the UIBehaviourRepresentations::UIScenes of the UIBehaviourRepresentations::UIStoryboard. The "fitness club system" is represented as a Frame containing all Dialogs.

Figure 6.24 illustrates the UML State Chart that defines the UI behaviour. It is completely and exclusively derived from the UIBehaviourRepresentations::UIStoryboard. The UIElements::TriggerUIElements are written in square brackets and the conditions are defined after the backslash. Every transition in the State Chart represents a UIBehaviourRepresentations::UserAction. All state transitions are triggered by the pressing of the corresponding Button. The transition from the "Facility reservation" state to the "Facility status" state is fulfilled when the condition "facility not available" is active. The transition from the "Facility reservation"

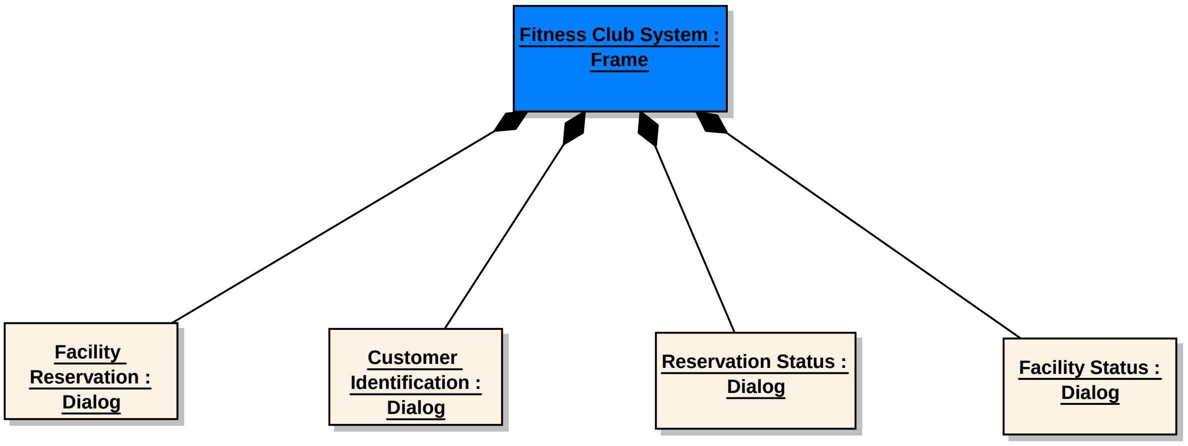

Fitness Club System : Frame
Facility Reservation : Dialog
Customer Identification : Dialog
Reservation Status : Dialog
Facility Status : Dialog

Figure 6.23: Object diagram for fitness club system

state to the “Custumer identification” state is fulfilled when the condition “facility available” is active.

6.5 Example of model transformation in MOLA

The goal of this section is to demonstrate the feasibility of defining formal ReDSeeDS model transformations in MOLA. The main criteria are the readability and maintainability of transformations. In addition, this example serves as a mini-tutorial for MOLA, specific to typical ReDSeeDS tasks.

The example is based on the informal, but precise transformation algorithms in sections 6.2.1 and 6.2.2 for building an initial architecture model in UML from the requirements in RSL. Consequently, the example structure of the architecture model described in sections 4.2.2 and 6.2 is used as the target.

Only those RSL elements, whose transformation is informally specified in 6.2, are used as the source model for the MOLA transformation. Other ones, such as general hyperlinked sentences, are simply ignored. The transformation description in 6.2 consists of two parts. The section 6.2.1 describes how the selected static structure in architecture model is built from the static part of the requirements. These algorithms are fully implement in the MOLA example. The section 6.2.2 describes how the behaviour representation in requirements is to be transformed into sequence diagrams (interactions) of the architecture model. This algorithm is implemented only partially, only the “main path” is implemented. This is due to the fact that there are so many special cases, which would make the complete algorithm too large for the example. The real algorithm, which is just all special cases implemented too, will come in the deliverable 3.3.

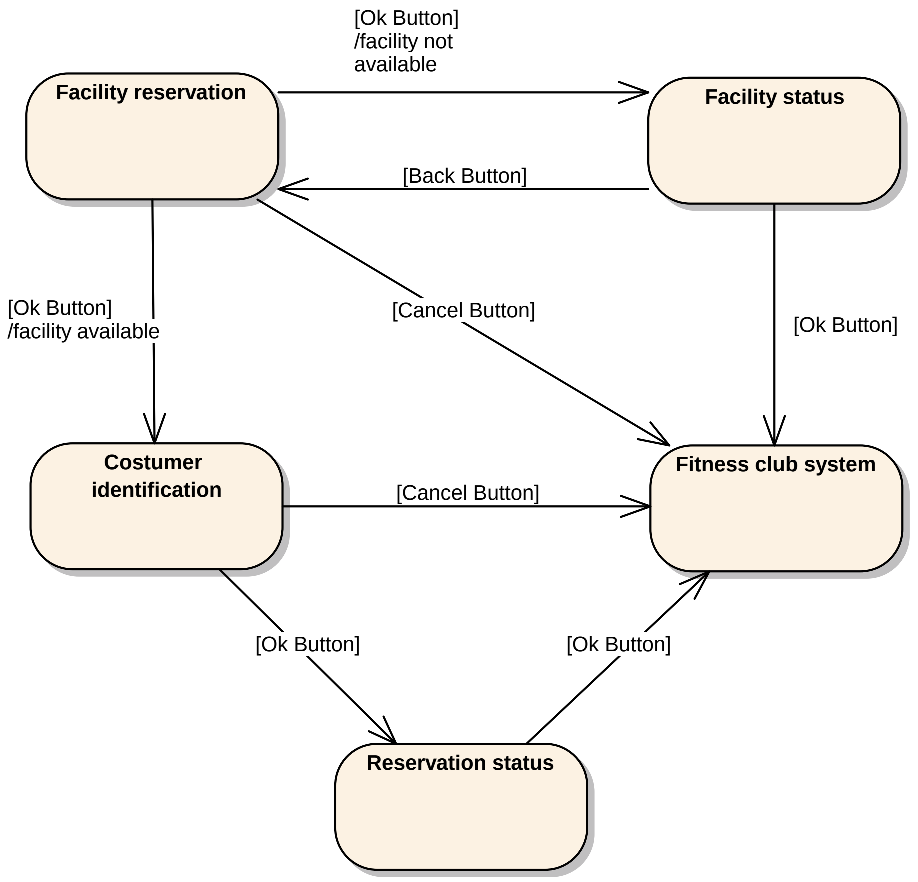


Figure 6.24: State Chart defining UI behaviour

But the structure of the provided MOLA transformation will remain the same. In short, the following restrictions are assumed. The source behaviour representation in RSL is assumed to be in activity scenarios, since this is the most understandable form. Only **one path** defined by control flows is assumed to be present in a scenario, and consequently only one sequence diagram is generated from the scenario (adding branching would require a lot of details, but no new ideas). The generation of other use case invocations is not implemented in detail (there some more detailed description of the informal algorithm would also be needed). Also not all possible sequences of subjects in consecutive activity nodes are supported - only those appearing in the examples of 6.2. These restrictions will be commented in more detail at the corresponding MOLA procedures. One more issue completely not mentioned is that in reality the RSL model must be taken from the would-be RSL tool, and the generated architecture model placed in a UML tool, most probably Enterprise Architect, but all this is delayed to the deliverable 3.3.

The section 6.5.1 describes the metamodels really used for the transformations, but the section 6.5.2 - the MOLA procedures which implement the transformations. All examples are built using the MOLA 2 tool (the version which is meant for use in ReDSeeDS).

6.5.1 Metamodel definitions in MOLA

As it was described in chapter 5, transformation design in MOLA starts with the specification of source and target metamodels in MOLA metamodelling language - "MOLA MOF".

Certainly, the source metamodel is based on the tool-ready version of the RSL metamodel - the chapter 3 of this document. However, this version is still not EMOF - there are {subsets} and {redefines} property strings, which should be completely eliminated. Further, in MOLA MOF currently only single specialisation (inheritance) is permitted. These are two issues which must be solved in the adapted version of the RSL metamodel.

Actually, not the complete RSL metamodel is required for building transformations. The transformations in 6.2 (and any reasonable similar transformations) can rely only on the "precise and formal part" of the RSL. These are use cases and their "formal" representations (activity, interaction, SVO sentences), use case packaging and relations, notions and their grouping and relations and, finally, phrase and term related elements which define the textual structure of the previously mentioned RSL elements. It is hard to imagine that descriptive sentences and related items could be used in formal transformations. Therefore an adequate subset of the RSL metamodel can be selected.

This subset selection significantly eases the solution of the two above mentioned issues. Most of the high level abstract metaclasses now have only one specialisation in the subset. Associations, which could be inherited from them, mainly have the “derived union” kind - the kind whose inheritance has no value for MOLA (they are subsetted anyway at a lower level). Therefore these high level metaclasses (together with their “derived union” kind associations) are simply dropped from the subset. Only few top level abstract metaclasses are retained, which provide real inheritance of attributes or associations, and the specialisation hierarchy is “flattened” accordingly. In fact, this approach has totally excluded the need for {subsets} and {redefines}. In most cases, single specialisation has been achieved too, but in few places a special semantically equivalent shift of attributes or associations has been applied. The result is a **semantically equivalent** (to the part of the tool-ready RSL metamodel) **MOLA-ready metamodel**. At instance level any RSL model part (corresponding to this metamodel part of interest) in the future RSL tool can be imported into MOLA repository without any loss of information according this MOLA-ready metamodel. It should be noted that the current metamodel contains only activity representations of use cases, but interaction or SVO sentence scenario based representations could be included as well in the future. Therefore this example RSL metamodel could be used as a base for future real MOLA-ready metamodels for transformations.

The next figures show this MOLA-ready metamodel for the relevant RSL part.

Figure 6.25 shows the remaining top-level RSL metaclasses. The top metaclass SCLElement is retained, since it contains the uid attribute and serves as a general anchor for any RSL element (e.g., for traceability). This diagram contains also everything what is really needed for transformations from use cases and top-level elements for activity representation of use cases. The name attribute has been moved to SCLElementsPackage in order to avoid multiple specialisation.

The figure 6.26 shows SVO sentences and other constrained language sentences used for describing the possible contents of activity nodes and activity edges. Nearly the whole fragment is taken from RSL tool-ready metamodel as presented there.

The next fragment - figure 6.27 presents the required for algorithm definition phrases and terms. Actually only the nouns and verbs are directly used in defining the transformations of SVO sentences. The important issue is distinguishing between the direct and indirect object of a complex verb phrase.

Finally, the figure 6.28 shows the notions, their packaging and relationships and links to other elements. Only the facts relevant for transformations are shown.

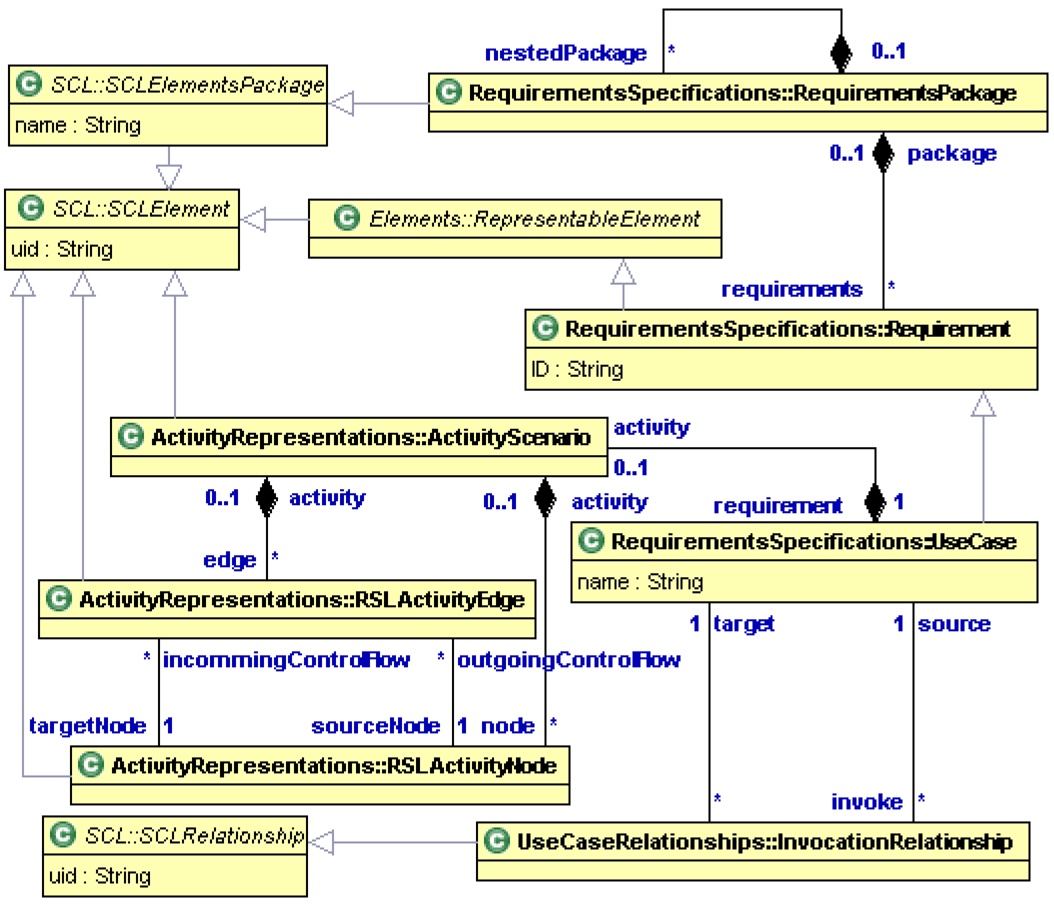

SCL::SCLElementsPackage
name : String
RequirementsSpecifications::RequirementsPackage
nestedPackage
*
0..1
0..1
package
SCL::SCLElement
uid : String
Elements::RepresentableElement
requirements
*
RequirementsSpecifications::Requirement
ID : String
ActivityRepresentations::ActivityScenario
activity
0..1
0..1
activity
edge
*
0..1
activity
requirement
1
RequirementsSpecifications::UseCase
name : String
ActivityRepresentations::RSLActivityEdge
* incommingControlFlow
* outgoingControlFlow
1 target
1 source
targetNode 1
sourceNode 1
node *
ActivityRepresentations::RSLActivityNode
*
invoke *
SCL::SCLRelationship
uid : String
UseCaseRelationships::InvocationRelationship

Figure 6.25: Top RSL metaclasses in MOLA-ready metamodel

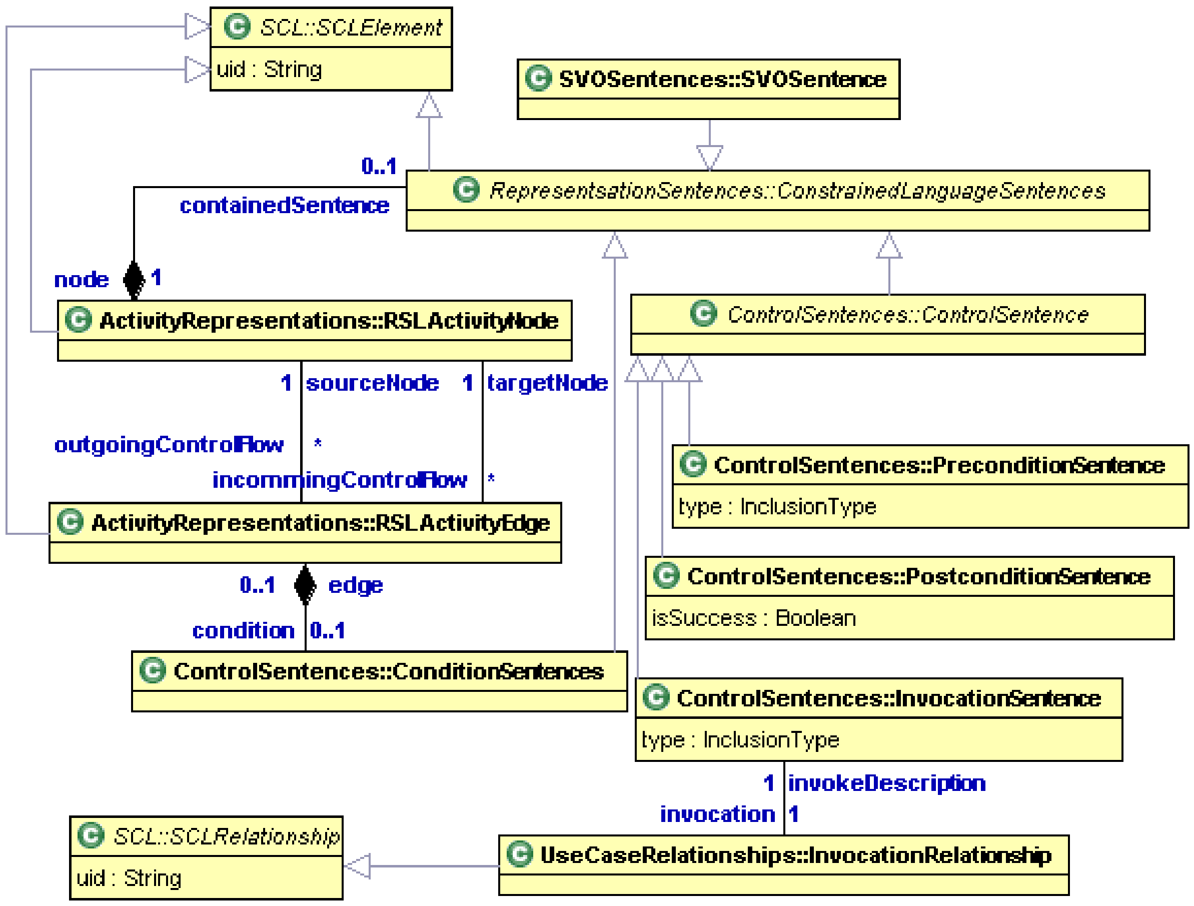

SCL::SCLElement
uid : String
SVOSentences::SVOSentence
0..1
containedSentence
RepresentsationSentences::ConstrainedLanguageSentences
node
1
ActivityRepresentations::RSLActivityNode
ControlSentences::ControlSentence
1
sourceNode
1
targetNode
outgoingControlFlow
*
incommingControlFlow
*
ControlSentences::PreconditionSentence
type : InclusionType
ActivityRepresentations::RSLActivityEdge
ControlSentences::PostconditionSentence
isSuccess : Boolean
0..1
edge
condition
0..1
ControlSentences::ConditionSentences
ControlSentences::InvocationSentence
type : InclusionType
1
invokeDescription
invocation
1
SCL::SCLRelationship
uid : String
UseCaseRelationships::InvocationRelationship

Figure 6.26: Constrained language sentences for activity scenario elements

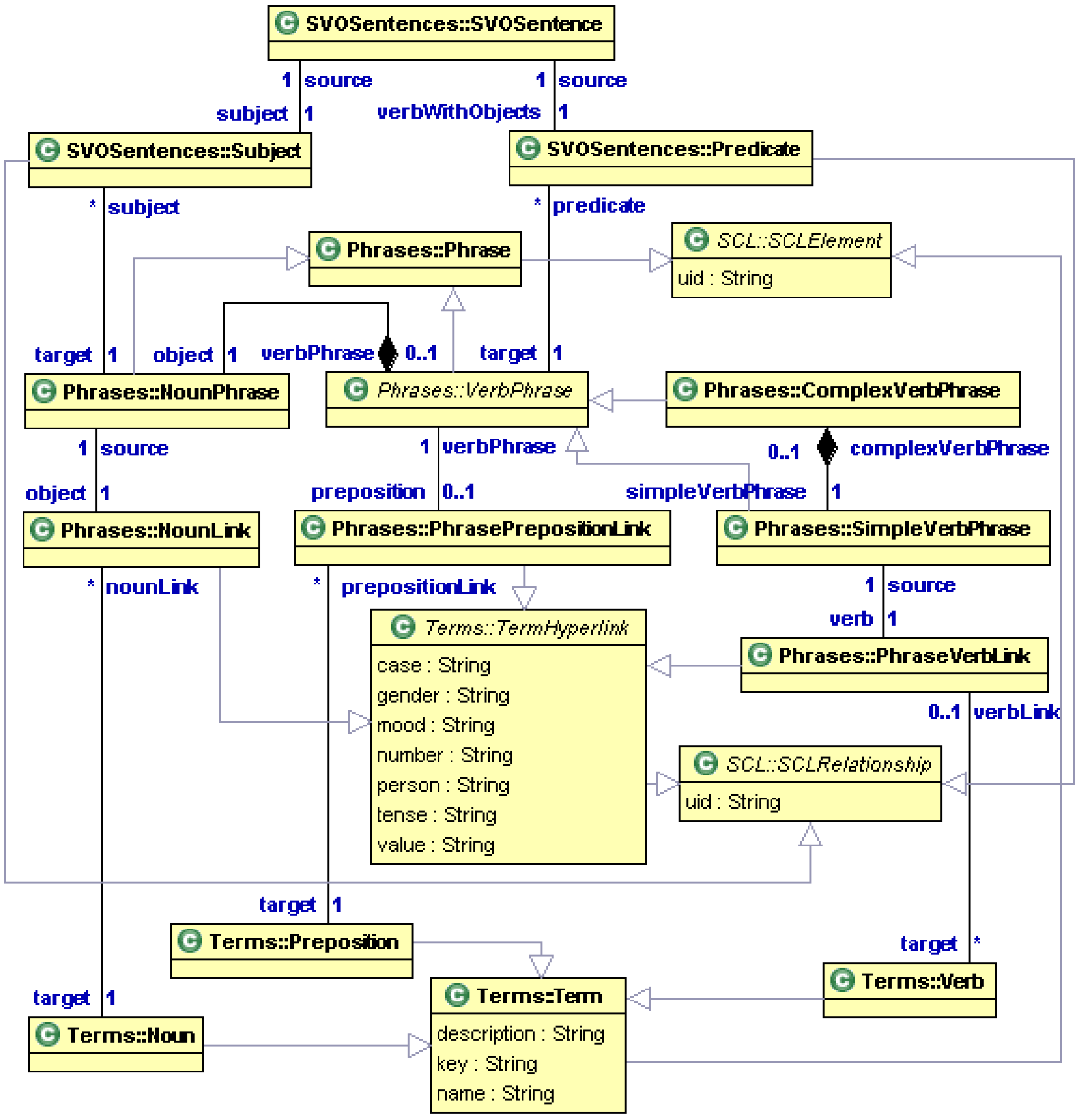


Figure 6.27: Phrases and terms used in MOLA-ready metamodel

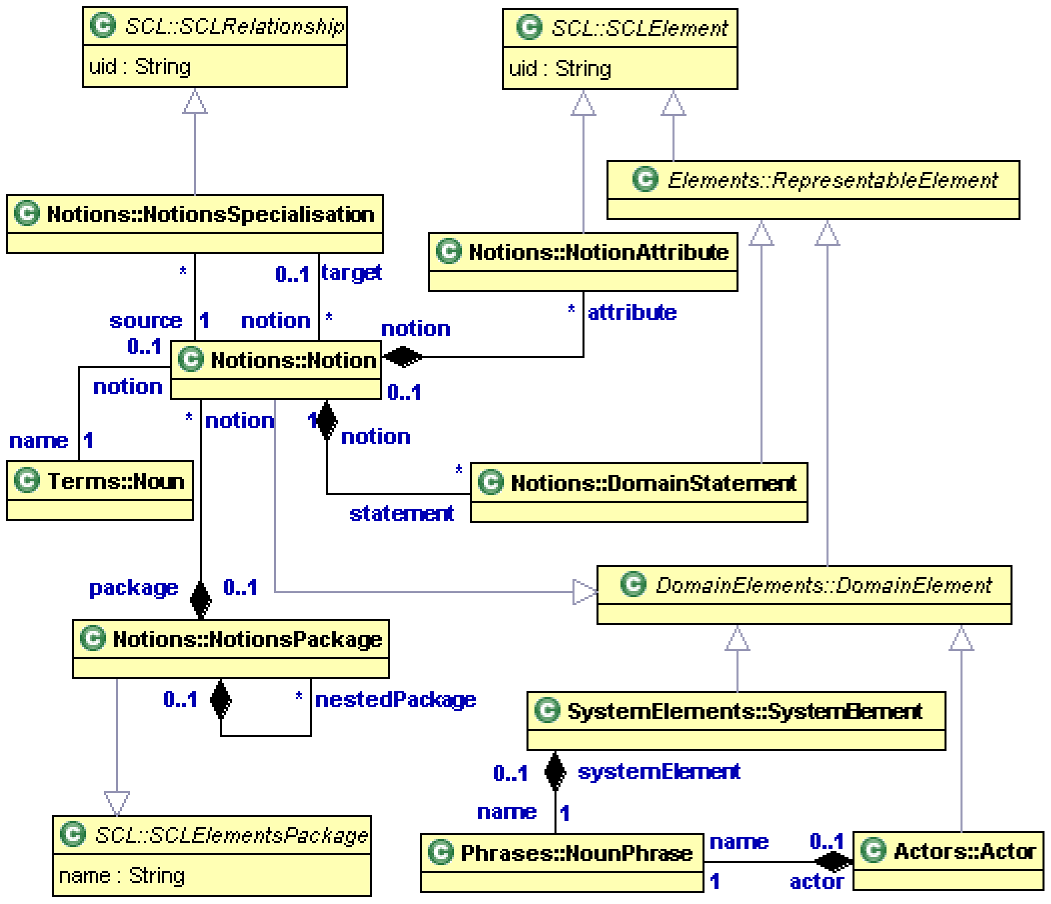


Figure 6.28: Notions and their relationships

It should be noted that for easy definition of transformations in MOLA, especially patterns in rules, it is convenient to use larger class diagrams. Then navigation possibilities between instances are more clearly visible. For example, the whole source metamodel of this example optimally could be shown in one diagram. The MOLA tool supports this easily, but unfortunately such a diagram cannot be shown in a printed page.

The target metamodel is the used subset of the UML 2.0 metamodel [Obj05b]. Since the algorithm in 6.2 uses only component, class and sequence diagrams only the UML elements usable in these diagrams are shown. Not all possible elements are even shown for these diagrams, only those which have a certain relevance for the architecture model style used in the example. There is the static structure part and the behaviour part in an architecture model. The static structure part uses packages, classes, components, interfaces and operations. All these elements are contained in the Kernel package of the UML metamodel. The included elements are completely equivalent to the original UML 2.0 metamodel, only some unused abstract metaclasses are "flattened out" from inheritance hierarchies.

The situation is more complicated for the behaviour part, which is described by means of sequence diagrams. The original UML 2.0 metamodel is unnecessary complicated for sequence diagrams (interactions, since the metamodel is common with communication diagrams). The version 2.1.1 has provided no improvements. There are so many metaclasses (and consequently instances) involved in building even a simple sequence diagram. It can hardly be seen there, where the messages go from and to, and what these messages actually represent. This fact has been noticed by several leading researchers in the metamodelling area, including R.B. France [Model-Driven Development Using UML 2.0: Promises and Pitfalls IEEE Computer 2006 02]. The metamodel in UML 1.4 for sequence diagrams (interactions) has another peculiarity. On the other hand, the Enterprise Architect tool uses a very simple, but completely custom metamodel for sequence diagrams. Therefore the decision was taken to use a significantly simplified version of UML 2.0 metamodel for defining sequence diagrams (interactions) in the example, in order to make the example readable. The real solution, what version of metamodel to use in ReDSeeDS to a great degree will depend on the selection of a tool chain to be used. But there is no doubt that any of these solutions can be used for MOLA transformations.

It should be reminded that the UML metamodel has no concept of diagram, it is only an abstract syntax. Therefore the target metamodel contains no diagrams (class, sequence, etc.), it depends on UML tool facilities how to visualise generated instances of metaclasses.

The next figures show the target metamodel. Figure 6.29 shows the used fragment of the Kernel package.

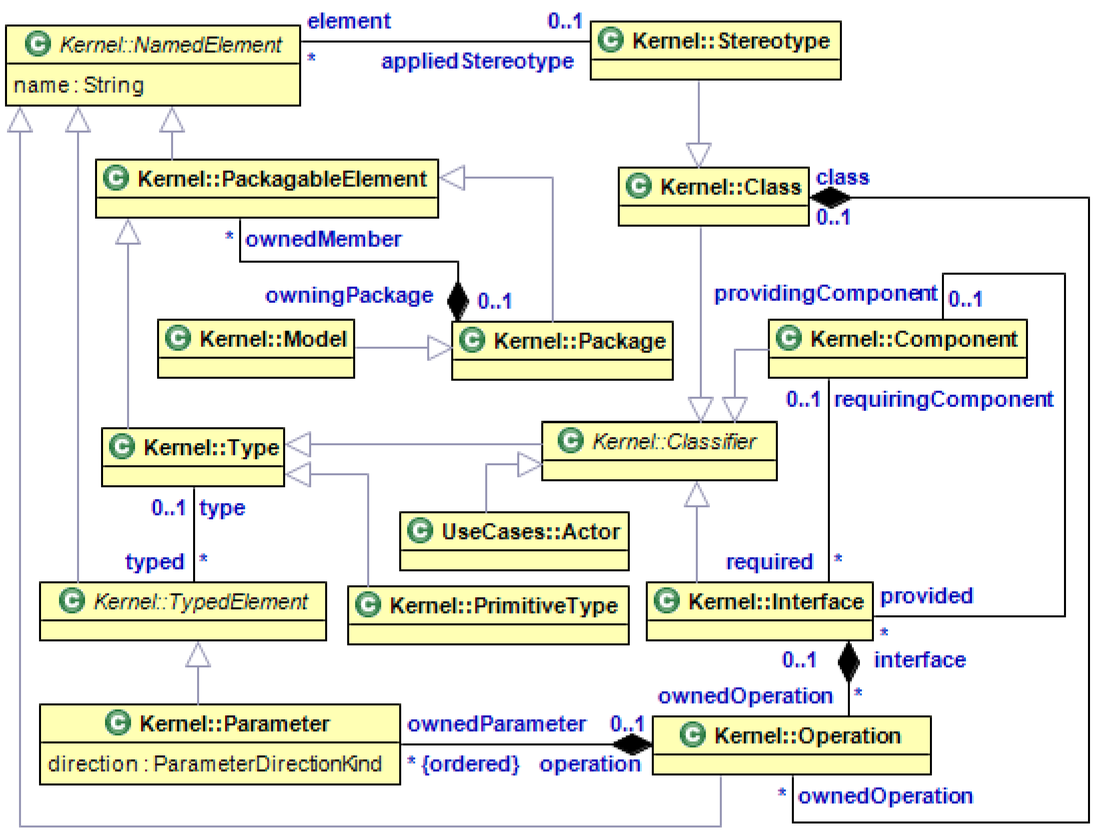


Figure 6.29: The used fragment of UML Kernel package

Figure 6.30 shows the simplified version of the Interactions package. Each message directly has its send and receive ends placed on the corresponding lifelines. The message signature can point either to a signal (unspecified message) or to an operation (operation invocation). A message can have arguments (simplified expressions) which must conform to the invoked operation parameters. A lifeline represents a named ConnectableElement, whose type can be a class, interface or component.

Finally, the figure 6.31 shows the UML and RSL elements used for traceability link kinds to be generated in MOLA transformations.

It should be noted that in MOLA traceability elements are used also for transformations themselves - to find in an easy way the results already obtained. Usually mapping associations are used for this purpose, but “mapping classes” necessary for ReDSeeDS serve this goal as well.

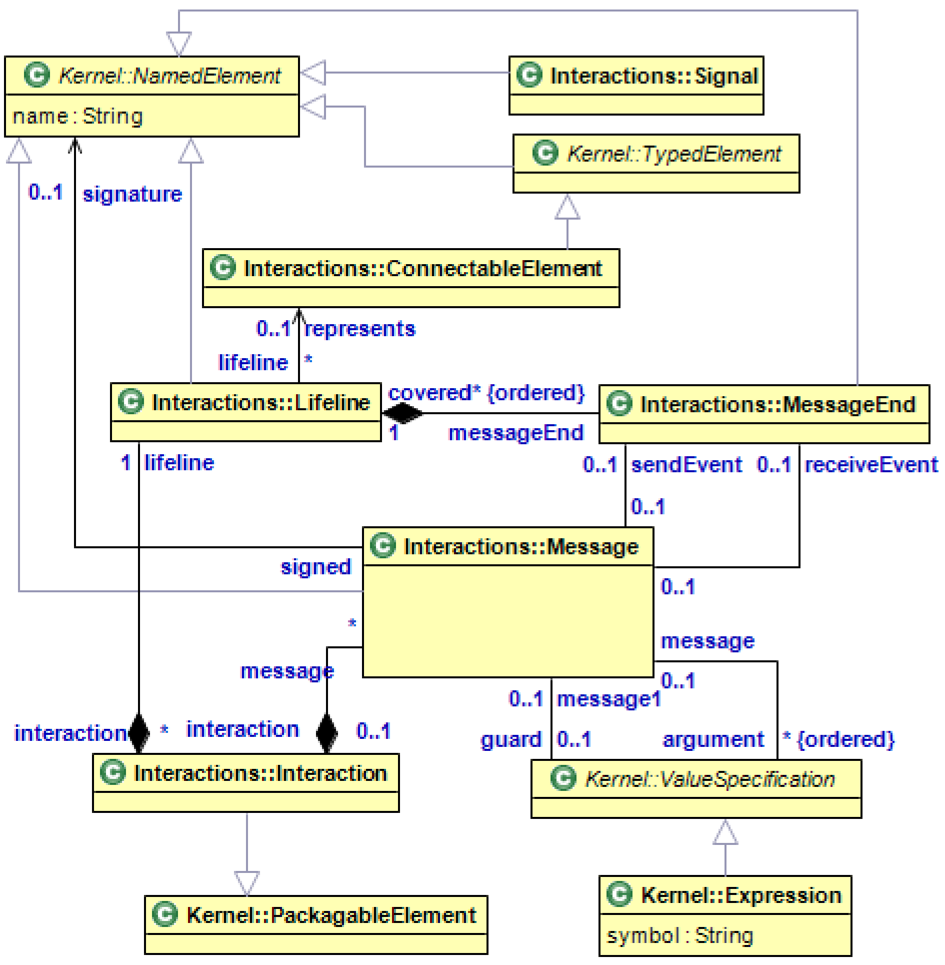


Figure 6.30: The simplified Interactions package

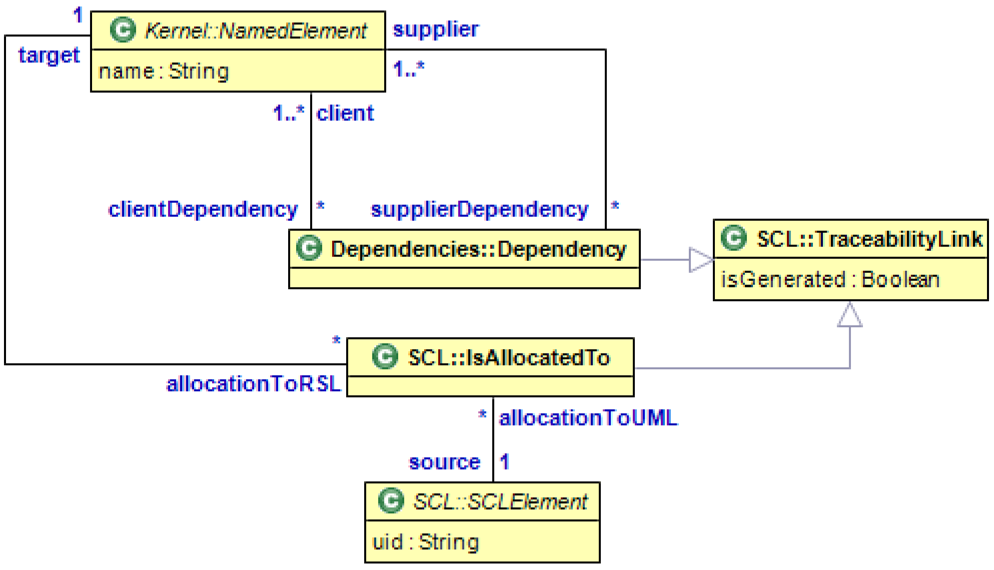


Figure 6.31: Traceability elements used for transformations

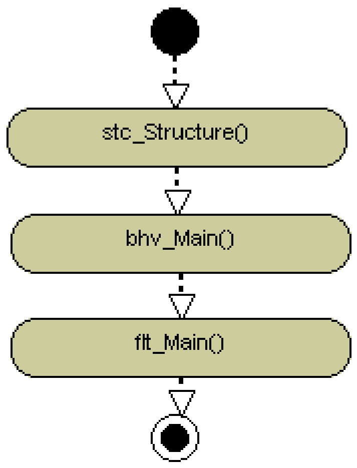


Figure 6.32: Main procedure of the transformation

6.5.2 Transformation procedures in MOLA

This section describes the MOLA procedures implementing the transformation. Since the informal algorithm in 6.2 has two parts - the static structure and behaviour, so has the set of MOLA procedures. The main procedure (figure 6.32) is very simple, it simply invokes the procedures implementing the two parts. The third part is added too, because more static structure elements have been built by the building procedures than required by the algorithm, the unused ones must be deleted at the end. Thus the static structure serves also as a kind of "compiler symbol table" in the transformation, making it easier to develop.

Processing of static structure

The static structure building has its own manager - the procedure stc_Structure, which invokes the real builders (the figure 6.32).

The next figure 6.35 shows the first procedure doing a real job - building the required packages and components with fixed names in the target architecture model. Its first rule contain only class elements for instance (Package, Component and Interface) creation. The second rule uses also a reference to the instance built by the first rule, namely to Interfaces package, in order to add new links to this instance. A similar thing is done with the interface UI. All these fixed packages actually have been defined in the full version of the Architecture model of the 4 layer Fitness Club example, mentioned in section 6.3. Figure 6.34 shows how this "fixed infrastructure" of the example looks like in the Enterprise Architect model tree browser.

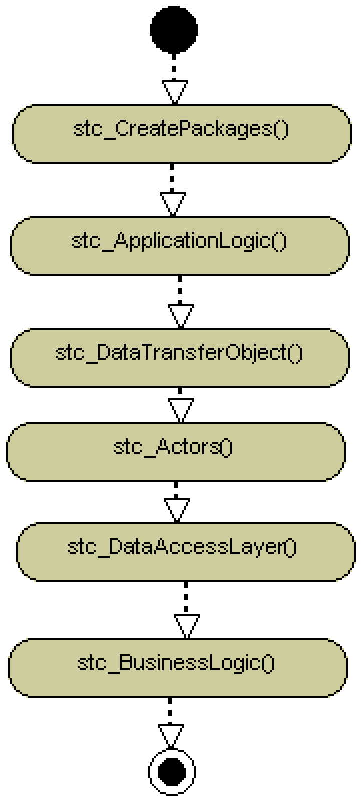
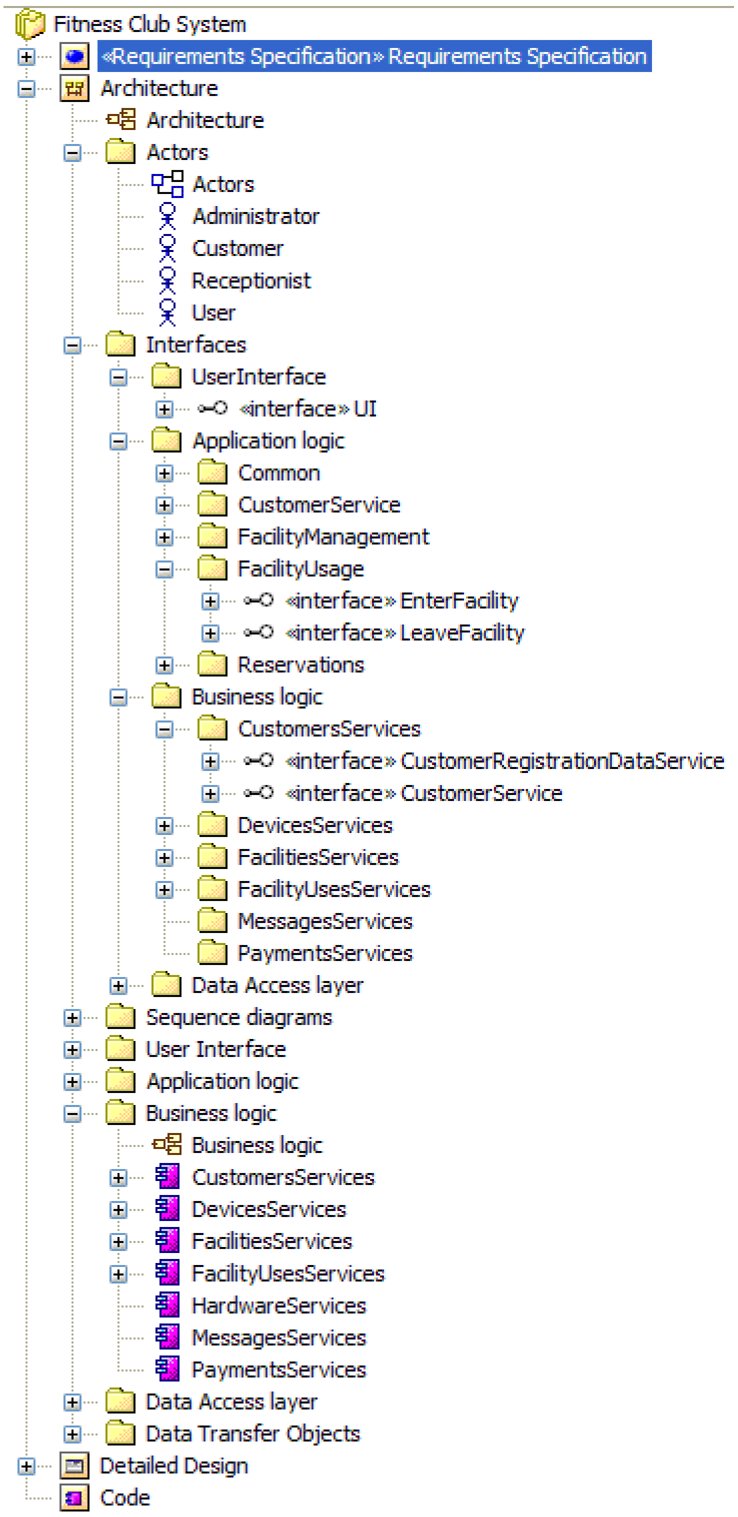


Figure 6.33: Main procedure for building the static structure

The next procedure to be invoked - stc_ApplicationLogic (figure 6.36) is a nontrivial one, which implements the first step of the informal algorithm. For each **requirements package** containing a use case it builds an application logic **component** (and a subpackage within the Application-Logic package under Interfaces), and for each **use case** in such a package it builds an **interface**, which is associated both to the component providing it and the package containing it. For each of the generated elements the naming conventions from 6.2.2 must be applied.

Since this is the first "typical" MOLA procedure, more comments are given to it in order to facilitate the understanding of the graphical syntax of MOLA. The first two rules locate the two instances of package with the name ApplicationLogic built by the previous procedure, each of which has a different context. The first rule has to locate the top level package (directly under the Architecture model), therefore the "anchor" for location is the single Model instance built, consequently, the {single} annotation may be used for locating the model instance itself. The owner instance, the ownedMember link and the name constraint uniquely determine the required package instance. In the second rule the "anchor" is a Package with the name Interfaces, and though actually there is only one such, the {start} annotation must be used because there are many Package instances. These two rules illustrate typical "design patterns" in MOLA how required instances can be located.



```
Fitness Club System
  «Requirements Specification» Requirements Specification
  Architecture
    Architecture
    Actors
      Actors
      Administrator
      Customer
      Receptionist
      User
    Interfaces
      UserInterface
        «interface» UI
      Application logic
        Common
        CustomerService
        FacilityManagement
        FacilityUsage
          «interface» EnterFacility
          «interface» LeaveFacility
        Reservations
      Business logic
        CustomersServices
          «interface» CustomerRegistrationDataService
          «interface» CustomerService
        DevicesServices
        FacilitiesServices
        FacilityUsesServices
        MessagesServices
        PaymentsServices
      Data Access layer
    Sequence diagrams
    User Interface
    Application logic
    Business logic
      Business logic
      CustomersServices
      DevicesServices
      FacilitiesServices
      FacilityUsesServices
      HardwareServices
      MessagesServices
      PaymentsServices
    Data Access layer
    Data Transfer Objects
  Detailed Design
  Code
```

Figure 6.34: The “fixed infrastructure” of architecture model to be generated

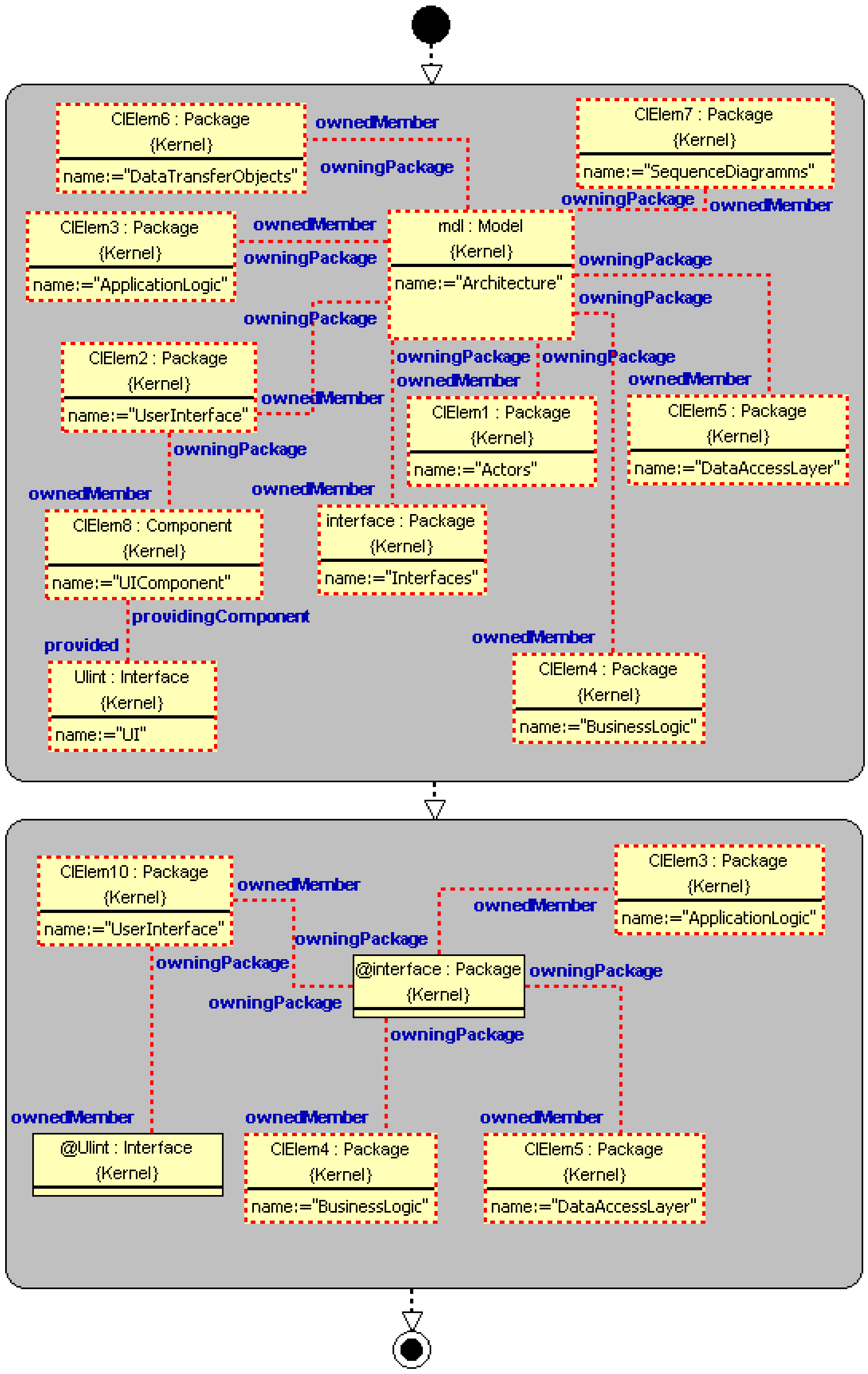

ClElem6 : Package {Kernel} name:="DataTransferObjects"
ownedMember
owningPackage
ClElem7 : Package {Kernel} name:="SequenceDiagramms"
owningPackage
ownedMember
ClElem3 : Package {Kernel} name:="ApplicationLogic"
ownedMember
owningPackage
mdl : Model {Kernel} name:="Architecture"
owningPackage
owningPackage
owningPackage
ClElem2 : Package {Kernel} name:="UserInterface"
ownedMember
owningPackage
owningPackage
ownedMember
ownedMember
ClElem1 : Package {Kernel} name:="Actors"
ClElem5 : Package {Kernel} name:="DataAccessLayer"
owningPackage
ownedMember
ownedMember
ClElem8 : Component {Kernel} name:="UIComponent"
interface : Package {Kernel} name:="Interfaces"
providingComponent
provided
ownedMember
UIint : Interface {Kernel} name:="UI"
ClElem4 : Package {Kernel} name:="BusinessLogic"
ClElem10 : Package {Kernel} name:="UserInterface"
ownedMember
ownedMember
ClElem3 : Package {Kernel} name:="ApplicationLogic"
owningPackage
owningPackage
@interface : Package {Kernel}
owningPackage
owningPackage
owningPackage
ownedMember
ownedMember
ownedMember
@UIint : Interface {Kernel}
ClElem4 : Package {Kernel} name:="BusinessLogic"
ClElem5 : Package {Kernel} name:="DataAccessLayer"

Figure 6.35: Static package creation

The next construct is a foreach loop which iterates over all RequirementsPackage instances, containing at least one UseCase. Note how the "existential quantifier" on UseCase is implemented in the loop head - there must be at least one UseCase instance, reachable via the requirements link. Before we can proceed, the appropriate name for the new component and UML package must be generated - 6.2.2 requires the source name (which can be a phrase) converted to UpperCamelCase. The "utility" procedure utl_toCamelCase does this, it has one in parameter of the type String, and one inout parameter for the transformed value (the String variable @str is used for this second argument). The next rule inside the loop builds the component and the package instances, assigns the name attribute and builds the required links. The next statement within the body is a nested loop, which iterates over all instances of the UseCase, contained in the current package. After obtaining the appropriate name, the rule in the nested loop body builds the Interface instance and links it accordingly. Both building rules build also the required ReDSeeDS **traceability** support - an instance of isAllocatedTo (with isGenerated=true), with source pointing to the relevant RSL element and target to the UML element. This pair of loops is a very typical "design pattern" in MOLA for building a nested object structure.

For the next procedures only new "design patterns" will be commented.

The next procedure (figure 6.37) stc_DataTransferObject builds a data transfer object for each notion. Its name is obtained by adding the suffix DTO to the "camelled" notion name. The informal algorithm says that only notions appearing in SVO sentences should be processed. But for transformations it is much easier to build initially DTOs for all notions and then clear those which have not been referenced by an SVO sentence in a scenario (these sentences must be processed in a later step anyway). This is the so-called "compiler symbol table" principle - initially unused variables are also allocated. The structure of the procedure is similar to the previous one, but with one loop only.

An even simpler procedure (figure 6.38) stc_Actors builds a UML actor for each RSL actor.

The next procedure stc_DataAccessLayer (figure 6.39) again processes notions, this time together with notion packages. Notion packages are transformed into Data Access components, but notions themselves - into interfaces of these components. Again, only used in SVO notions should be transformed, but initially all notions are processed, unused interfaces are dropped at the end. The structure of the procedure is quite similar to that transforming uses cases into application logic interfaces.

There is one more very similar procedure stc_BusinessLogic (figure 6.40) which builds business logic layer components and their interfaces. Their names are obtained by adding suffixes

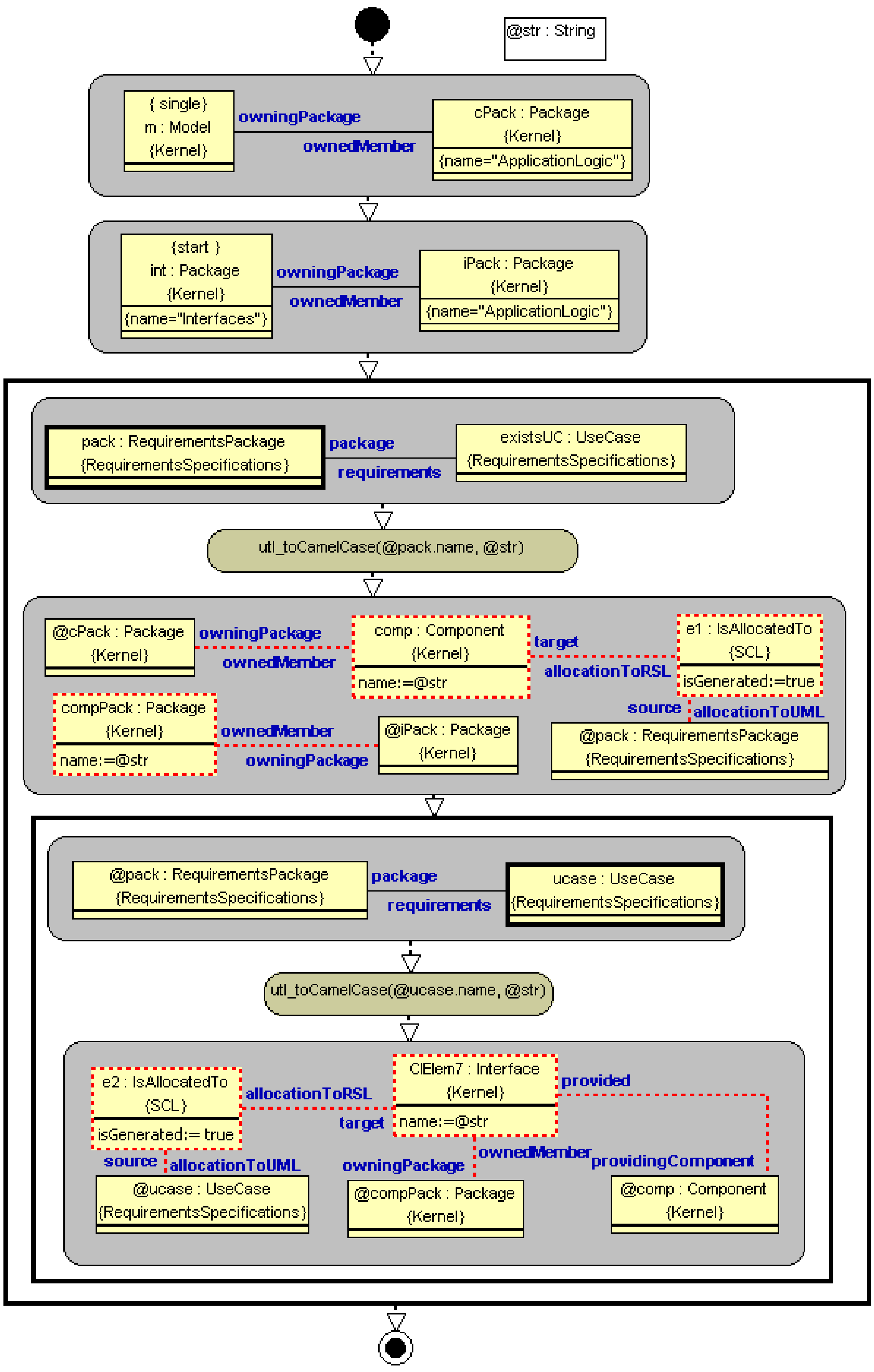


Figure 6.36: Procedure building the Application logic elements

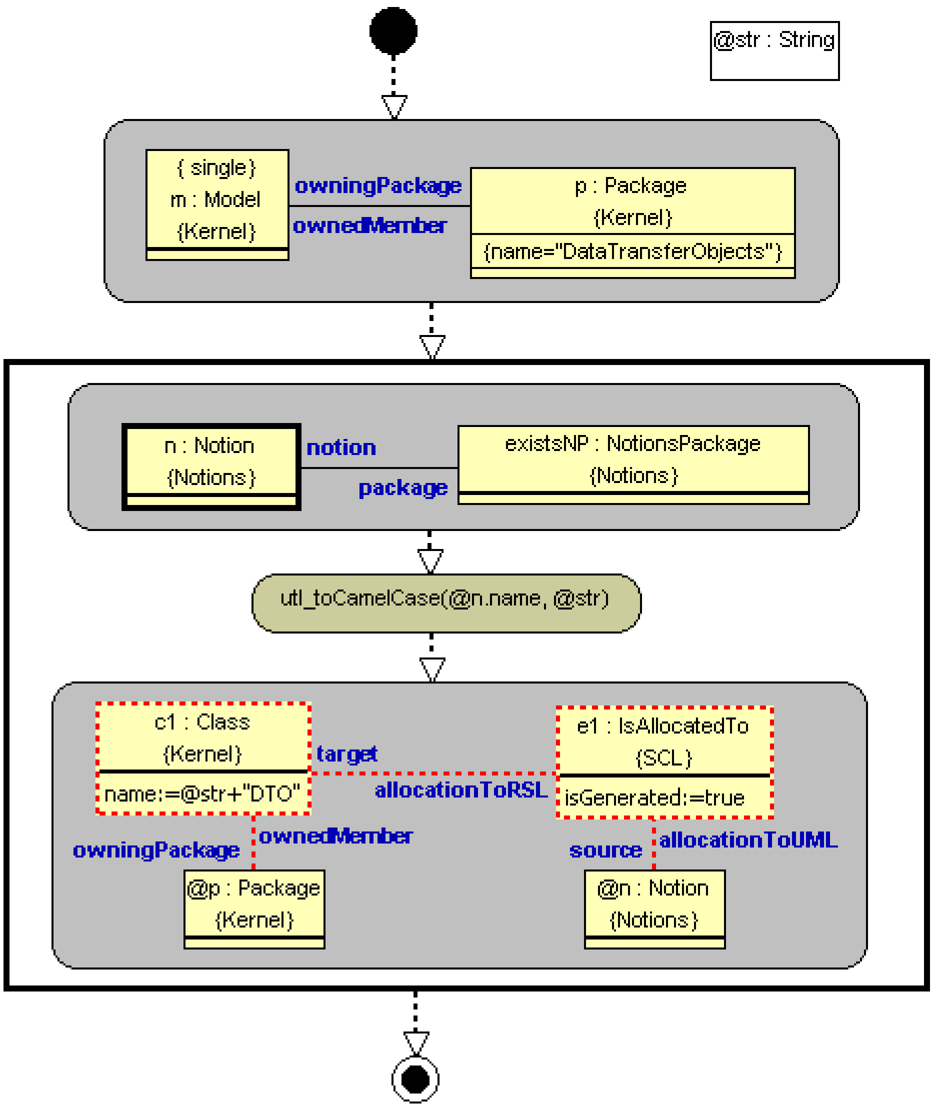


Figure 6.37: Procedure building the data transfer objects

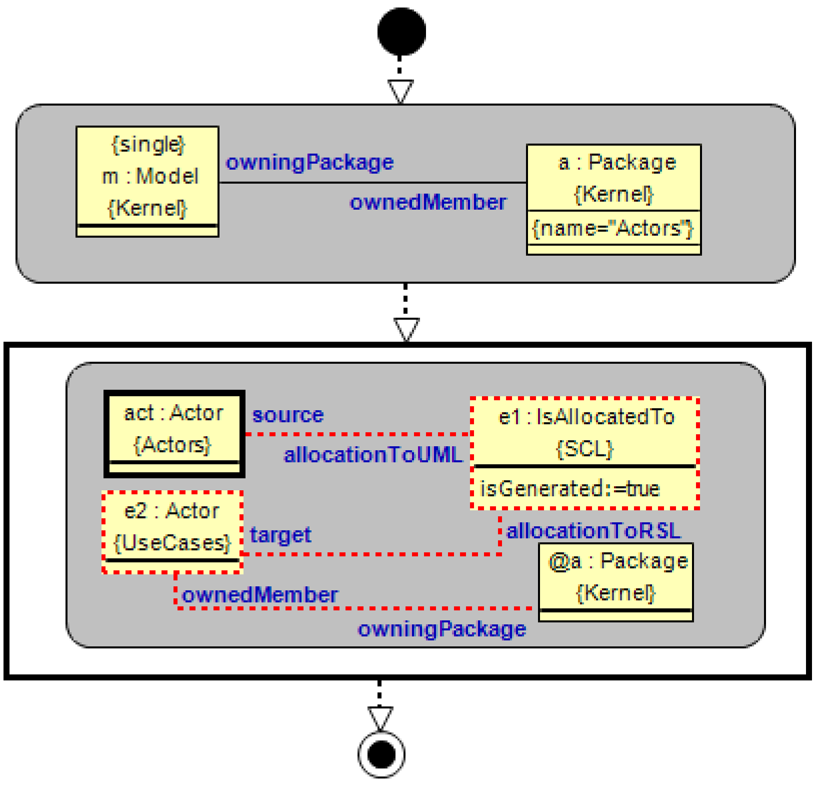


Figure 6.38: Procedure building UML actors

"Services" or "Service" respectively. Again, all are transformed, with unused ones dropped at the end. This completes the static structure building.

Processing of behaviour

The behaviour processing part has to process any use case having an activity scenario and build for it an **interaction** (sequence diagram at the UML abstract syntax level). These interactions are grouped into packages corresponding to use case packages. Other requirements representations are not processed in this example.

A very simple manager controls all this (figure 6.41).

The first non-trivial procedure is bhv_SequenceDiagramms (figure 6.42). This procedure finds Requirements packages, containing at least one use case (in a way similar to building application logic) and builds interaction packages for them. Then for each use case, having an activity scenario, an Interaction instance is built within the corresponding package. Finally, for this interaction the required lifelines and messages are built by the corresponding procedures, which

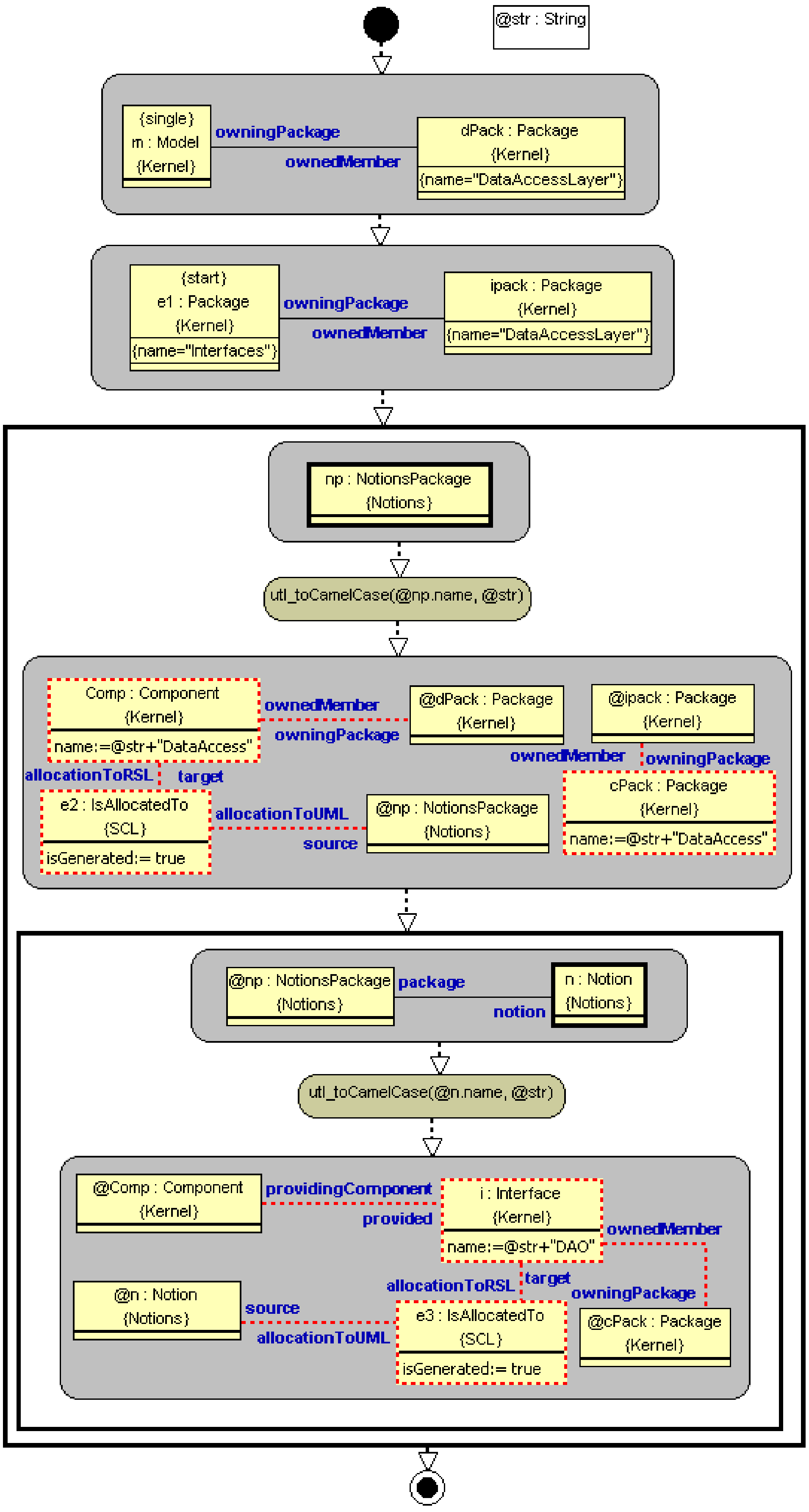


Figure 6.39: Procedure building data access objects

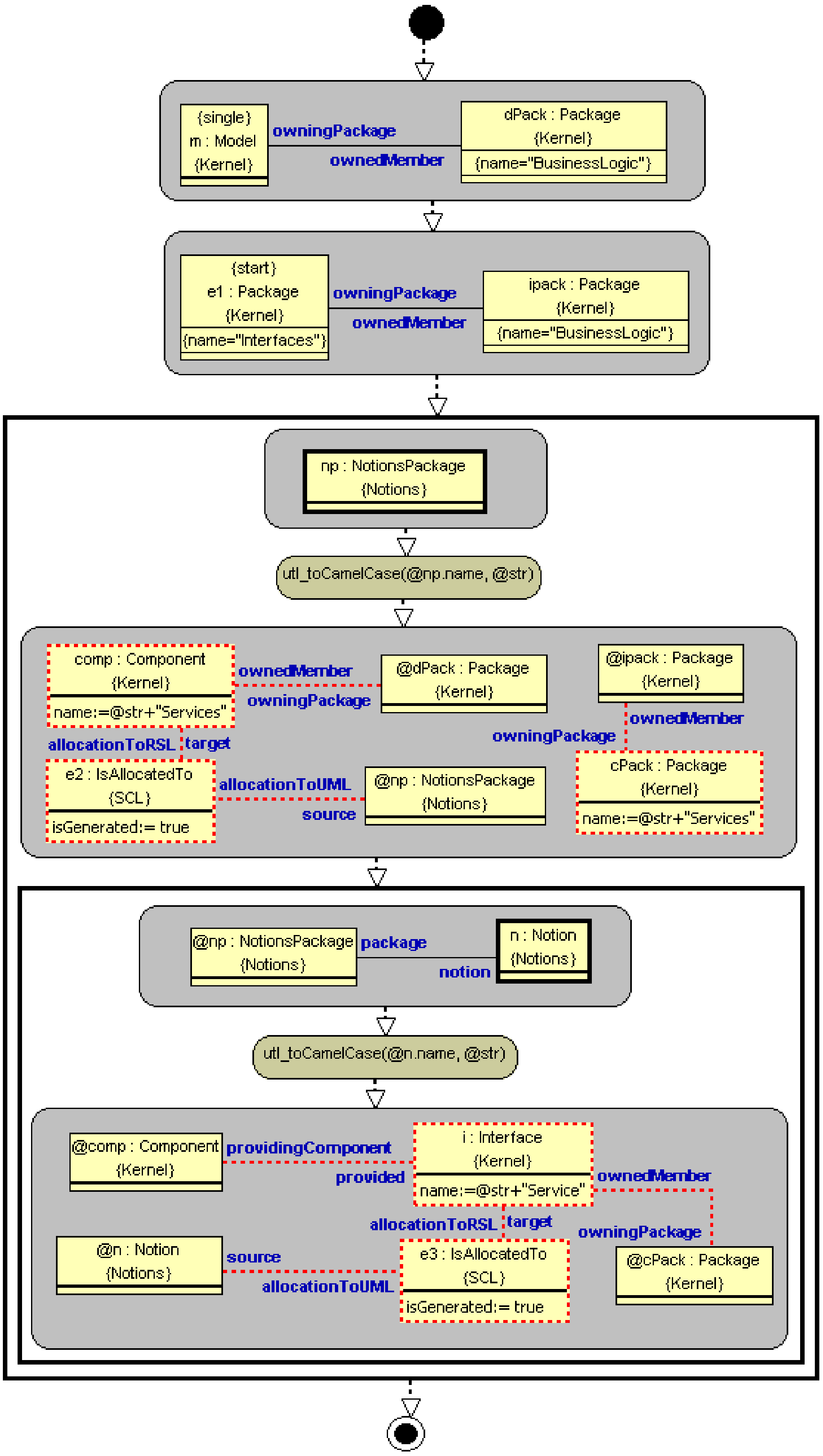


Figure 6.40: Procedure building business logic components

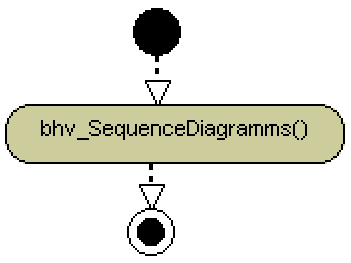


Figure 6.41: Procedure managing the behaviour

analyze the activity scenario. The invocations of these procedures have parameters - the activity scenario to be processed and the interaction to be built.

The next procedure to be explained is bhv_InteractionLifelines (figure 6.43). It builds the interaction lifelines on the basis of activity scenario provided by the first parameter, and adds these lifelines to the interaction, provided by the second parameter. Two of the lifelines are the same for all interactions, they correspond to the UI component and its UI interface (see figure 6.10). These fixed lifelines are built by the procedure bhv_StaticLifelines. The other lifelines are scenario specific. There is a lifeline for each actor appearing as a subject in an SVO sentence contained by activity node. The first loop in the procedure finds such actors and builds (in the second rule of the body) a lifeline representing the corresponding UML actor. Note how the **traceability** elements (instance of IsAllocatedTo and its links) built by the static structure builders are used to locate the UML actor corresponding to the RSL actor. One more lifeline represents the interface corresponding to the current use case (the use case owning this scenario). This interface again is located using the traceability. More lifelines can appear from invoked use cases, these are generated by the bhv_InvocationLifeline procedure (see figure 6.44).

The procedure for invocation lifelines is very similar, just the path how to locate the required interface starting from the invocation sentence contained by an activity node is longer. Note that still it can be implemented by one MOLA rule.

The most complicated part is the building of messages. The main message building procedure bhv_Messages (figure 6.45) just traverses the activity graph starting from the initial node. The first rule locates the initial node. The invoked procedure bhv_nextRSLActivityNode traverses one control flow edge and returns the next node in its second (inout) parameter. Note the use of class-typed ("pointer") variables pointing to nodes - current (now) and next. We remind that this example version can process only graphs without branching. If the next node is the final one, the job is done (currently postconditions are ignored). Otherwise the true message builder - the procedure bhv_GenerateMessages is invoked.

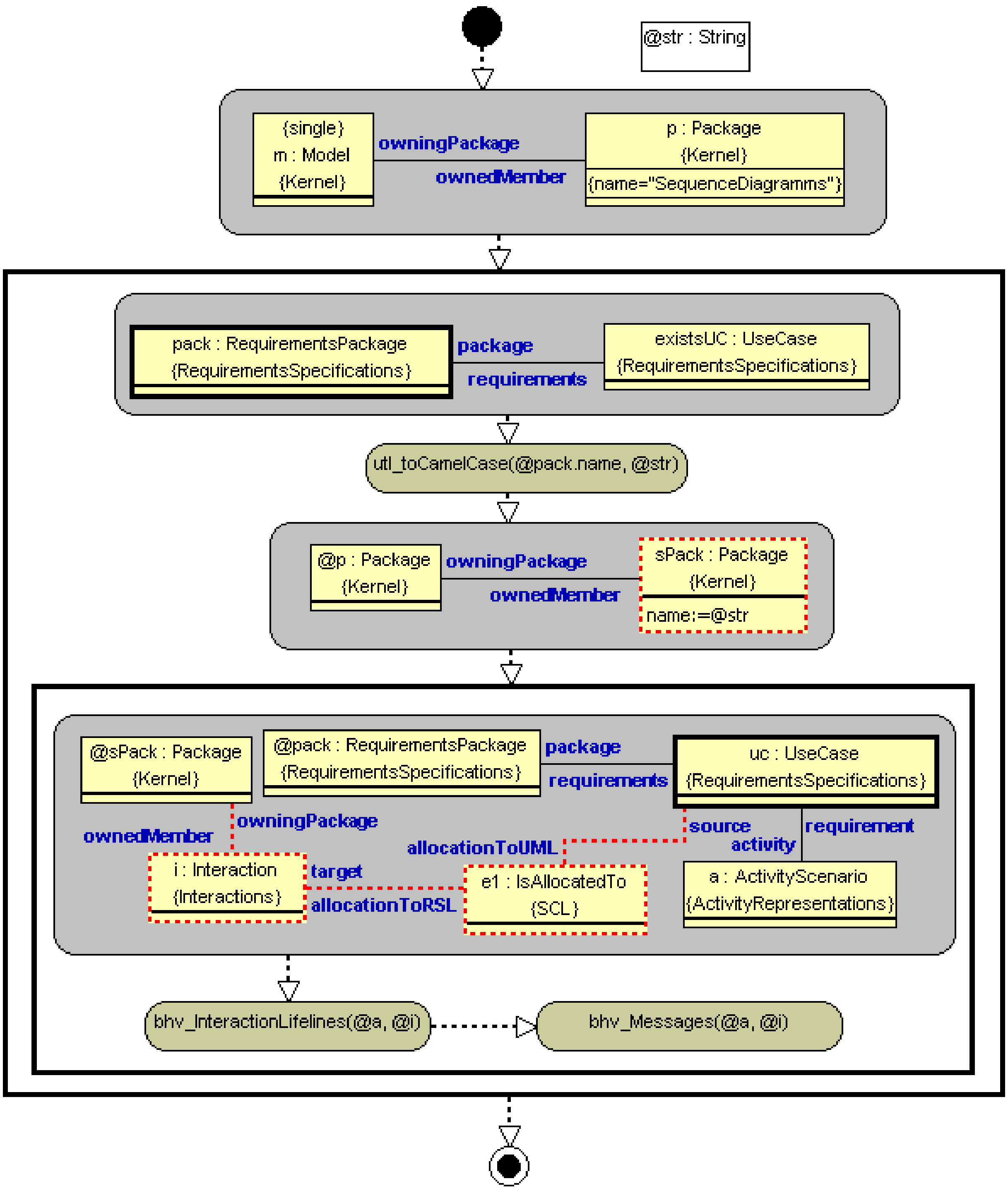


Figure 6.42: Procedure building interactions

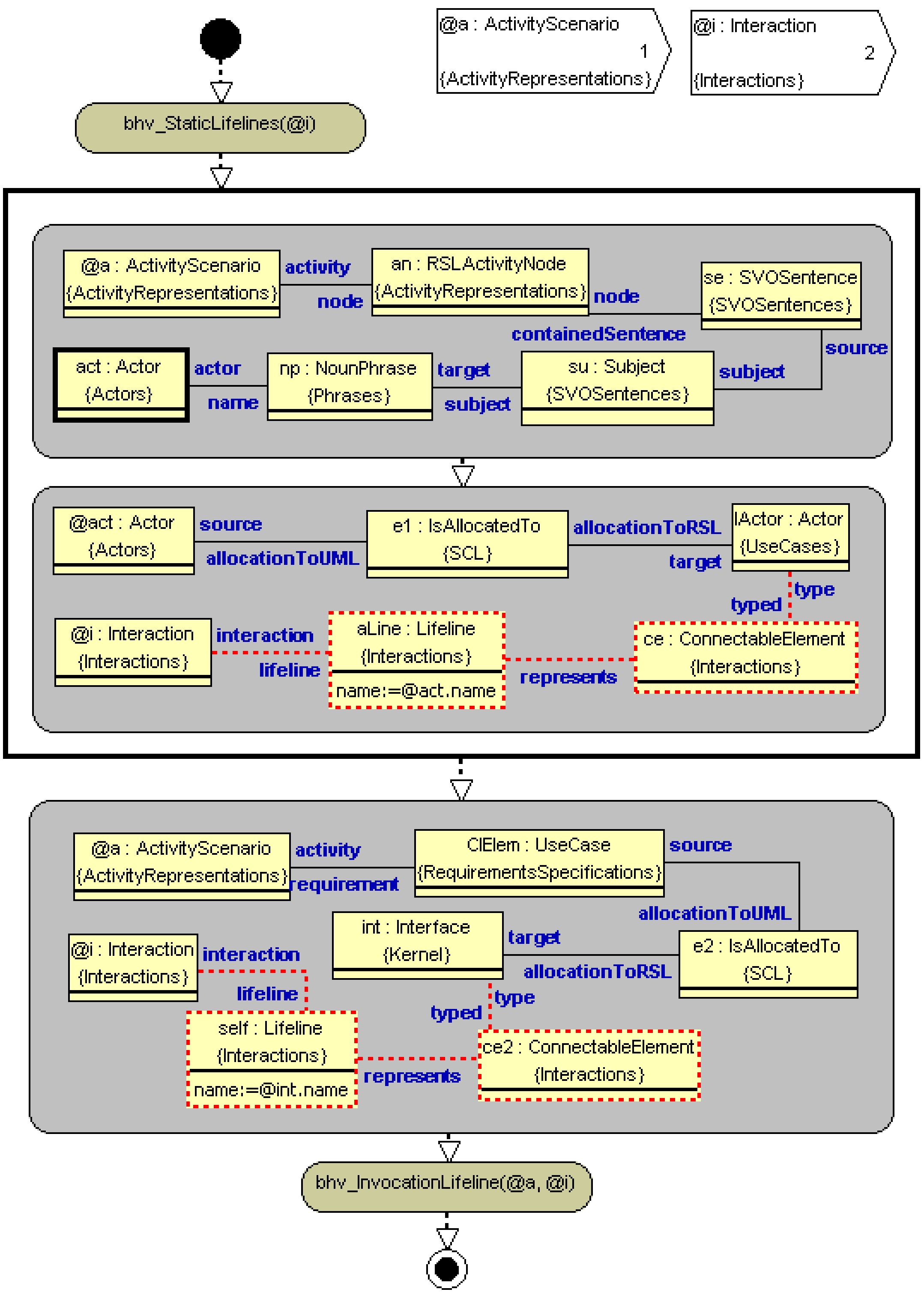


Figure 6.43: Procedure building lifelines

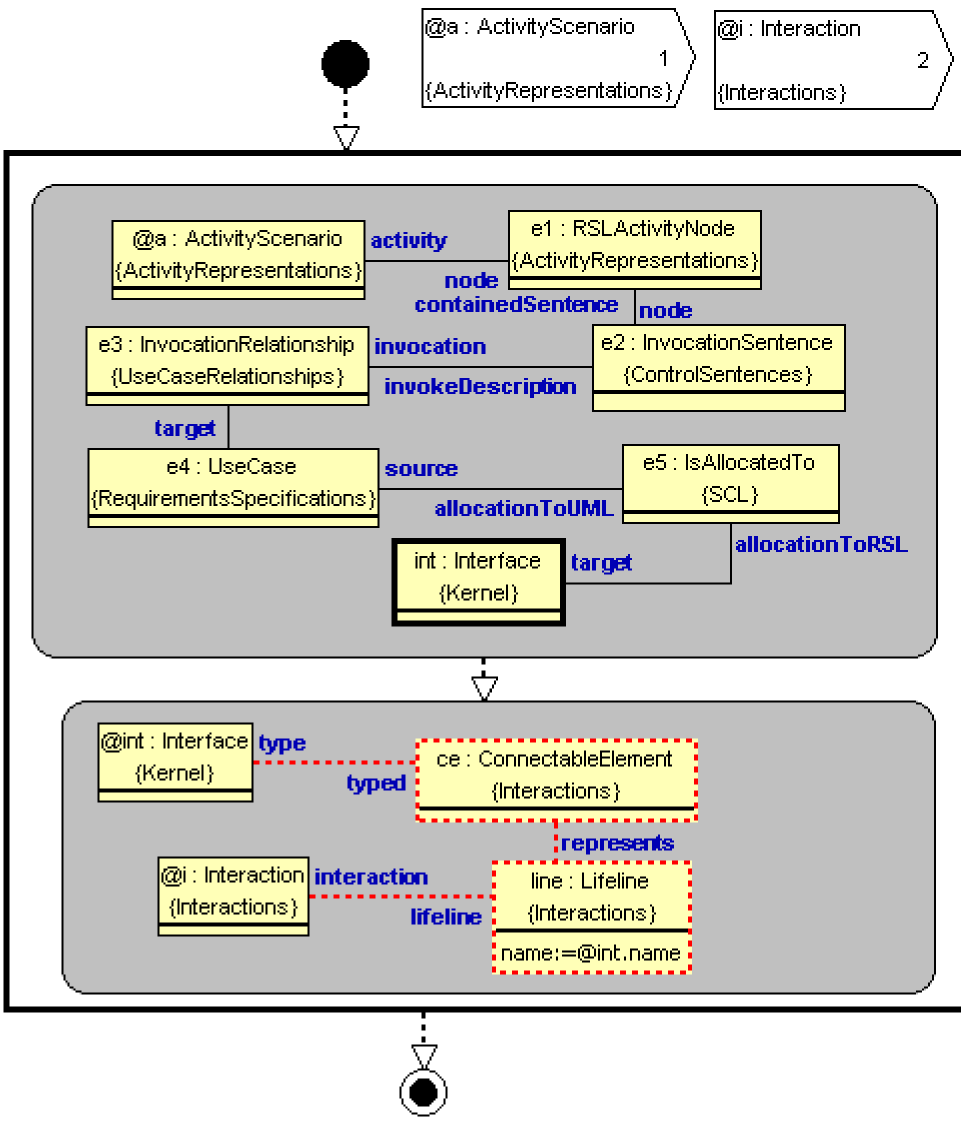


Figure 6.44: Procedure building invocation lifelines

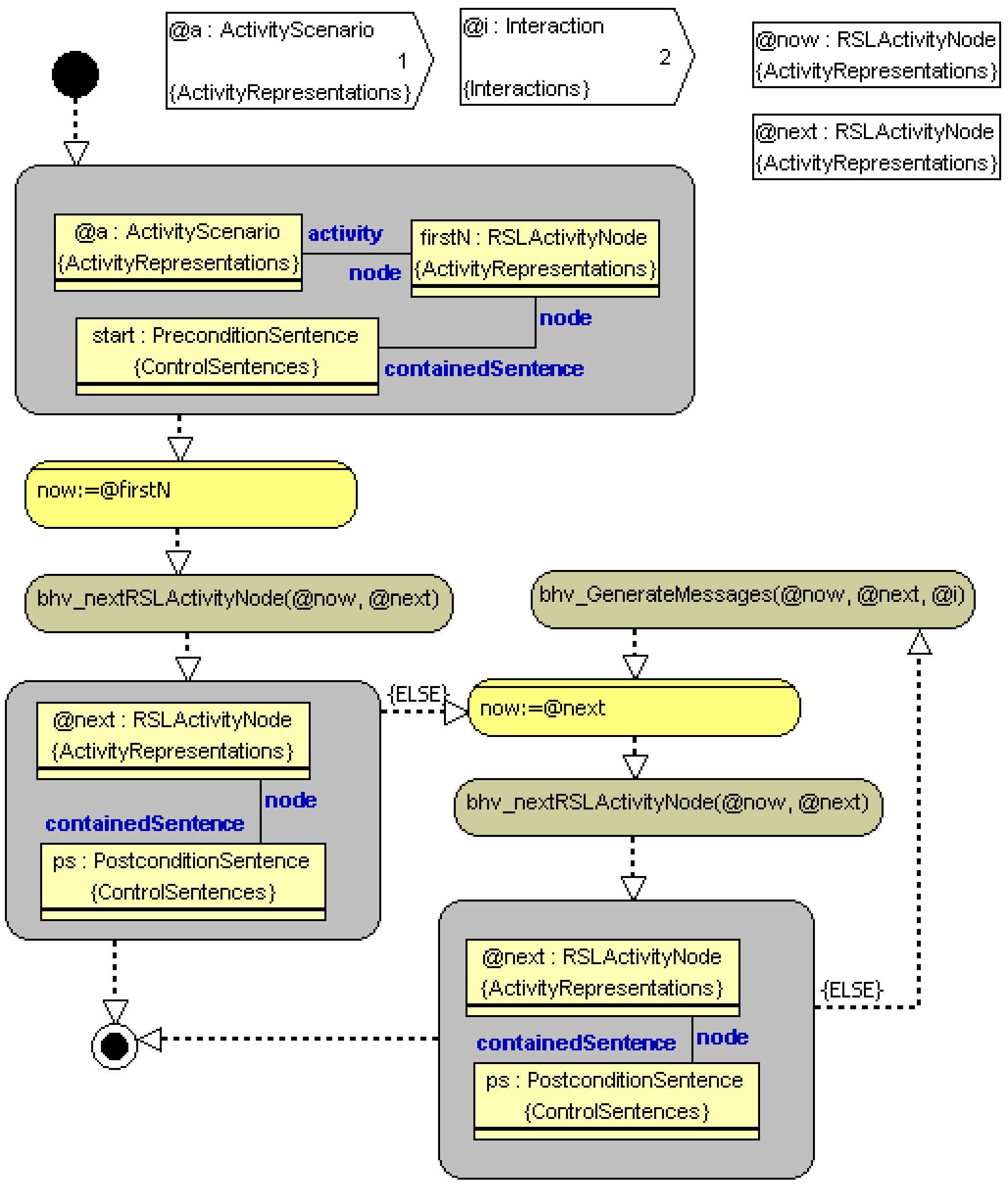

@a : ActivityScenario
1
{ActivityRepresentations}
@i : Interaction
2
{Interactions}
@now : RSLActivityNode
{ActivityRepresentations}
@next : RSLActivityNode
{ActivityRepresentations}
@a : ActivityScenario
{ActivityRepresentations}
activity
node
firstN : RSLActivityNode
{ActivityRepresentations}
node
start : PreconditionSentence
{ControlSentences}
containedSentence
now:=@firstN
bhv_nextRSLActivityNode(@now, @next)
bhv_GenerateMessages(@now, @next, @i)
{ELSE}
now:=@next
@next : RSLActivityNode
{ActivityRepresentations}
node
containedSentence
ps : PostconditionSentence
{ControlSentences}
bhv_nextRSLActivityNode(@now, @next)
@next : RSLActivityNode
{ActivityRepresentations}
{ELSE}
containedSentence
node
ps : PostconditionSentence
{ControlSentences}

Figure 6.45: Procedure traversing the activity graph

The procedure bhv_nextRSLActivityNode which traverses one control flow edge and returns the next node is shown in figure 6.46. This procedure includes facilities for processing invocation sentences, which are only partially implemented in the current example.

Now the message builder - the procedure bhv_GenerateMessages (figure 6.46) is explained. This procedure recognizes three kinds of messages to be generated, which are named actor-to-system, system-to-actor and system-to-system ("self-message") in the section 6.2.2. These kinds are distinguished on the basis of what is the subject kind (actor or system) in the current and next node. The three rules in the procedure serve as preconditions for these three cases. An appropriate message building procedure is invoked in each case.

The figure 6.48 shows the procedure bhv_ActorSystemProcedure, which processes the first case. In this case two messages must be generated - from actor (i.e., from the lifeline representing it) to UI component and from UI component to interface corresponding to the current use case. The first message is informal (has the signature of the type Signal). The second one is an operation invocation. This operation (without parameters) has to be generated and associated to the corresponding interface. The first rule locates the actor lifeline. The third rule locates the lifeline corresponding to "this" use case. Note the very long link chain in the pattern of this rule - this is implied by the given RSL metamodel, and such patterns are quite feasible for MOLA. The fourth and the fifth rules build the two messages and link their ends to the lifelines. Finally, the new operation is built (if it not already exists) and linked in an appropriate way. Two utility procedures are used in this procedure to analyze the verb phrase in the SVO sentence and to build the simple message or the CamelCase operation name, respectively (utl_GetPredicateText and utl_GetOperationText).

The next procedure (figure 6.49) is bhv_SystemActorProcedure, which processes the second case - system-to-actor. The procedure has to generate one message - from the current use case interface ("self") to the lifeline representing the fixed UI interface. The message is an operation invocation (for the UI interface). The operation must be generated (if it is not already there). The first rule locates the UI lifeline, the second - the "self" lifeline (using also traceability elements). The same utility procedure is used to build the operation name. Then the message is built and linked to the lifelines. Finally, the operation is built (if required) and linked appropriately.

The third and the most complicated system-to-system case is implemented by the procedure bhv_SystemSystemProcedure (figure 6.50). This procedure has to generate one message, however, in a very complicated way. The message goes from the "self" lifeline to a newly generated lifeline, which corresponds to an interface of a business component. This interface is the one which corresponds to the object of a simple verb phrase or the indirect object of a complex verb phrase. Fortunately, such a coding is used in RSL, that it is the object directly reachable

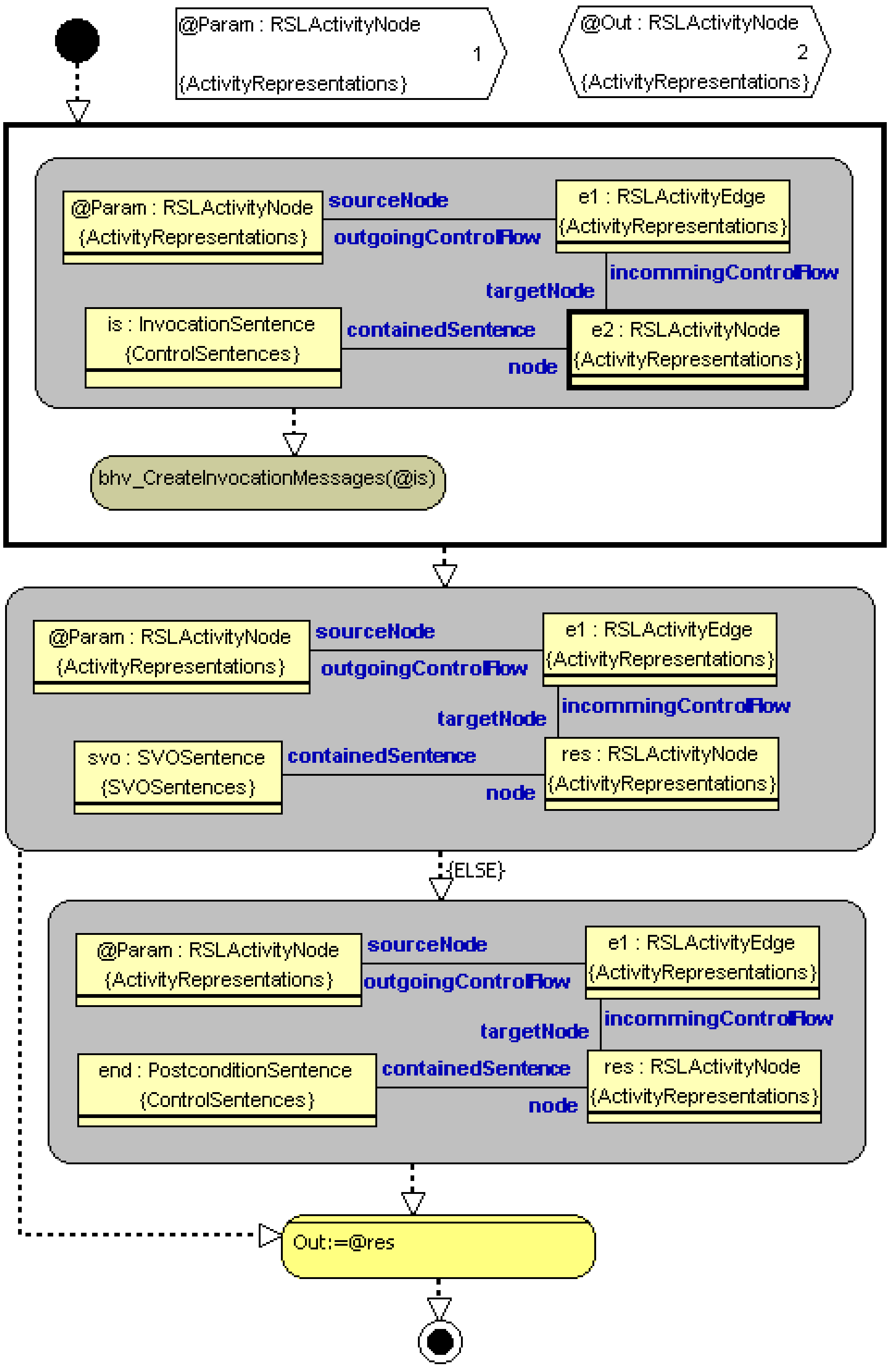

@Param : RSLActivityNode
1
{ActivityRepresentations}
@Out : RSLActivityNode
2
{ActivityRepresentations}
@Param : RSLActivityNode {ActivityRepresentations}
sourceNode
outgoingControlFlow
e1 : RSLActivityEdge {ActivityRepresentations}
incommingControlFlow
targetNode
is : InvocationSentence {ControlSentences}
containedSentence
node
e2 : RSLActivityNode {ActivityRepresentations}
bhv_CreateInvocationMessages(@is)
@Param : RSLActivityNode {ActivityRepresentations}
sourceNode
outgoingControlFlow
e1 : RSLActivityEdge {ActivityRepresentations}
incommingControlFlow
targetNode
svo : SVOSentence {SVOSentences}
containedSentence
node
res : RSLActivityNode {ActivityRepresentations}
{ELSE}
@Param : RSLActivityNode {ActivityRepresentations}
sourceNode
outgoingControlFlow
e1 : RSLActivityEdge {ActivityRepresentations}
incommingControlFlow
targetNode
end : PostconditionSentence {ControlSentences}
containedSentence
node
res : RSLActivityNode {ActivityRepresentations}
Out:=@res

Figure 6.46: Procedure traversing one edge

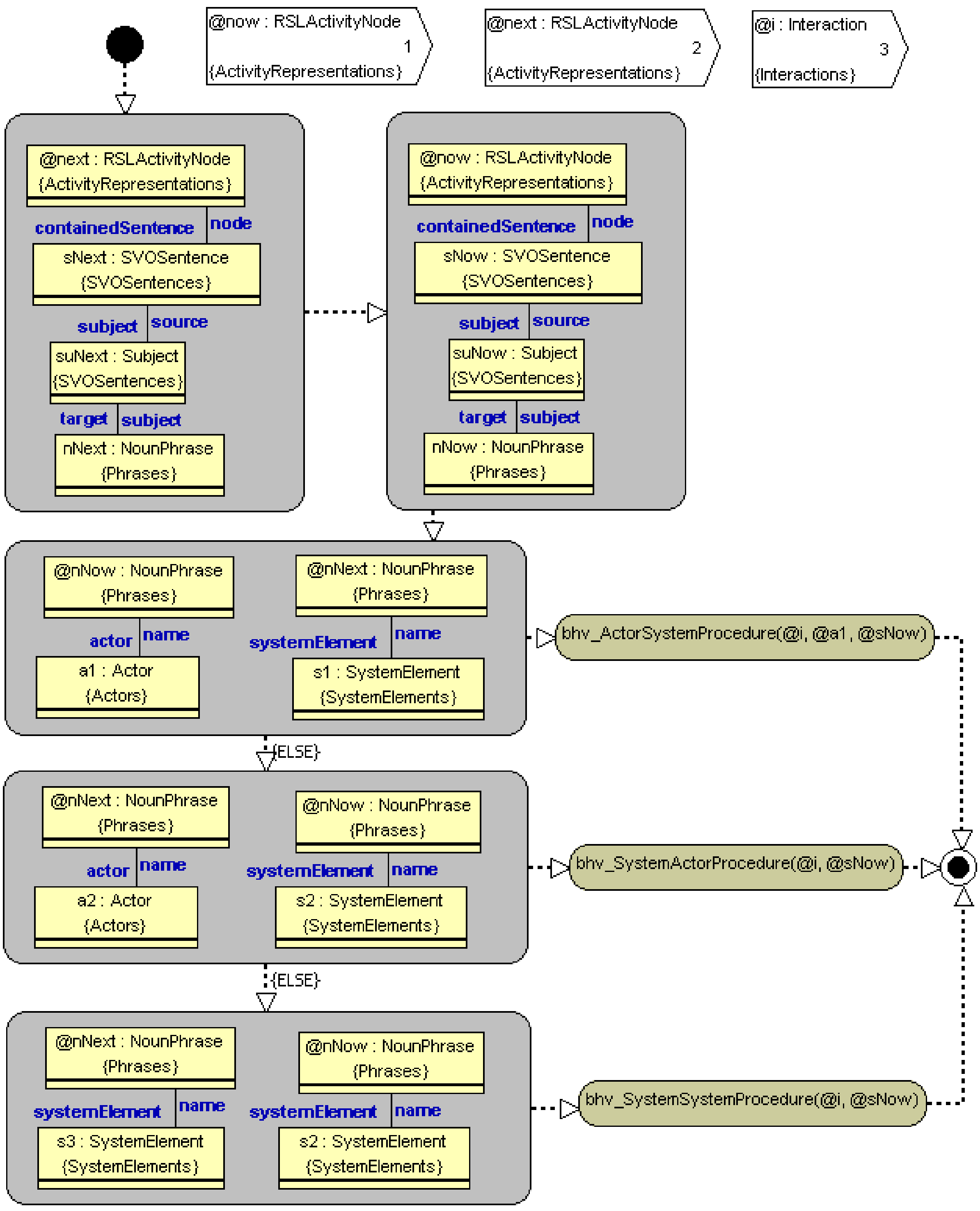

@now : RSLActivityNode 1 {ActivityRepresentations}
@next : RSLActivityNode 2 {ActivityRepresentations}
@i : Interaction 3 {Interactions}
@next : RSLActivityNode {ActivityRepresentations}
containedSentence
node
sNext : SVOSentence {SVOSentences}
subject
source
suNext : Subject {SVOSentences}
target
subject
nNext : NounPhrase {Phrases}
@now : RSLActivityNode {ActivityRepresentations}
containedSentence
node
sNow : SVOSentence {SVOSentences}
subject
source
suNow : Subject {SVOSentences}
target
subject
nNow : NounPhrase {Phrases}
@nNow : NounPhrase {Phrases}
@nNext : NounPhrase {Phrases}
actor
name
systemElement
name
a1 : Actor {Actors}
s1 : SystemElement {SystemElements}
bhv_ActorSystemProcedure(@i, @a1, @sNow)
{ELSE}
@nNext : NounPhrase {Phrases}
@nNow : NounPhrase {Phrases}
actor
name
systemElement
name
a2 : Actor {Actors}
s2 : SystemElement {SystemElements}
bhv_SystemActorProcedure(@i, @sNow)
{ELSE}
@nNext : NounPhrase {Phrases}
@nNow : NounPhrase {Phrases}
systemElement
name
systemElement
name
s3 : SystemElement {SystemElements}
s2 : SystemElement {SystemElements}
bhv_SystemSystemProcedure(@i, @sNow)

Figure 6.47: Procedure distinguishing message kinds

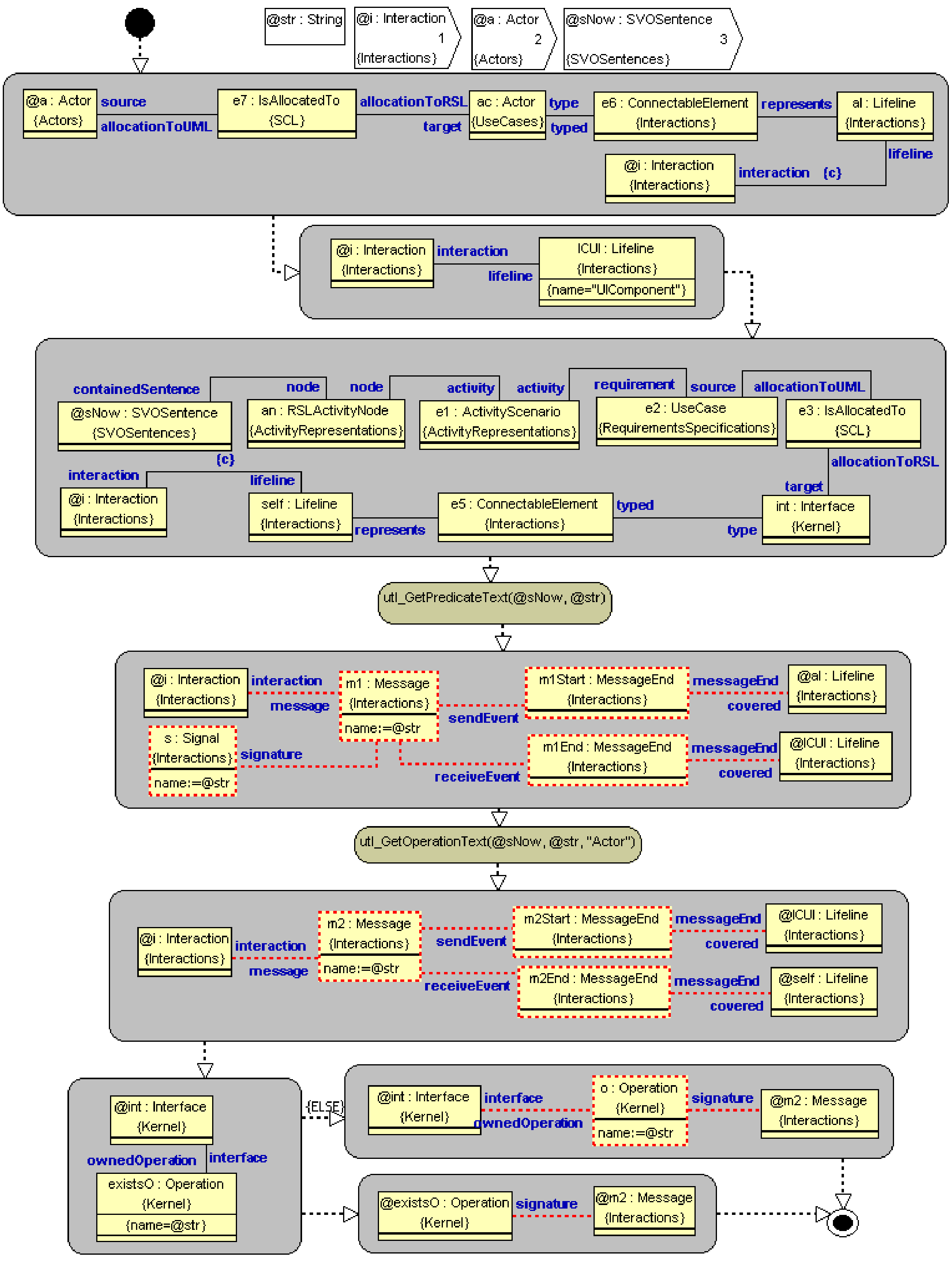

@str : String
@i : Interaction 1 {Interactions}
@a : Actor 2 {Actors}
@sNow : SVOSentence 3 {SVOSentences}
@a : Actor {Actors}
source
allocationToUML
e7 : IsAllocatedTo {SCL}
allocationToRSL
target
ac : Actor {UseCases}
type
typed
e6 : ConnectableElement {Interactions}
represents
al : Lifeline {Interactions}
lifeline
@i : Interaction {Interactions}
interaction {c}
ICUI : Lifeline {Interactions} {name="UIComponent"}
containedSentence
@sNow : SVOSentence {SVOSentences}
node
an : RSLActivityNode {ActivityRepresentations}
activity
e1 : ActivityScenario {ActivityRepresentations}
requirement
e2 : UseCase {RequirementsSpecifications}
e3 : IsAllocatedTo {SCL}
self : Lifeline {Interactions}
e5 : ConnectableElement {Interactions}
int : Interface {Kernel}
utl_GetPredicateText(@sNow, @str)
m1 : Message {Interactions} name:=@str
m1Start : MessageEnd {Interactions}
messageEnd
covered
@al : Lifeline {Interactions}
s : Signal {Interactions} name:=@str
signature
sendEvent
receiveEvent
m1End : MessageEnd {Interactions}
@ICUI : Lifeline {Interactions}
utl_GetOperationText(@sNow, @str, "Actor")
m2 : Message {Interactions} name:=@str
m2Start : MessageEnd {Interactions}
m2End : MessageEnd {Interactions}
@self : Lifeline {Interactions}
@int : Interface {Kernel}
ownedOperation
interface
existsO : Operation {Kernel} {name=@str}
{ELSE}
o : Operation {Kernel} name:=@str
@m2 : Message {Interactions}
@existsO : Operation {Kernel}

Figure 6.48: Procedure generating actor-to-system messages

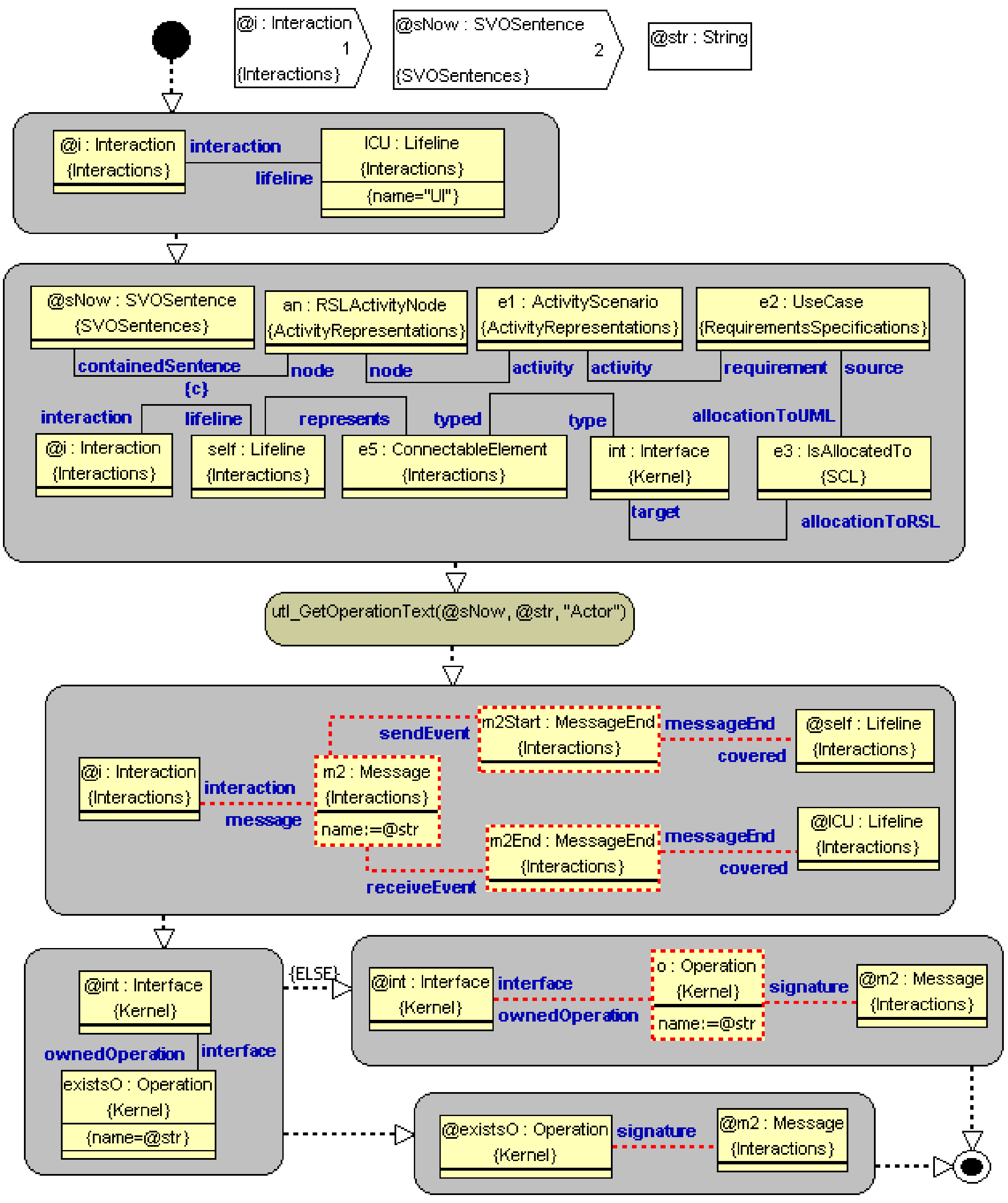

@i : Interaction 1 {Interactions}
@sNow : SVOSentence 2 {SVOSentences}
@str : String
@i : Interaction {Interactions}
interaction
lifeline
ICU : Lifeline {Interactions} {name="UI"}
@sNow : SVOSentence {SVOSentences}
an : RSLActivityNode {ActivityRepresentations}
e1 : ActivityScenario {ActivityRepresentations}
e2 : UseCase {RequirementsSpecifications}
containedSentence
{c}
node
node
activity
activity
requirement
source
interaction
lifeline
represents
typed
type
allocationToUML
@i : Interaction {Interactions}
self : Lifeline {Interactions}
e5 : ConnectableElement {Interactions}
int : Interface {Kernel}
e3 : IsAllocatedTo {SCL}
target
allocationToRSL
utl_GetOperationText(@sNow, @str, "Actor")
sendEvent
m2Start : MessageEnd {Interactions}
messageEnd
@self : Lifeline {Interactions}
covered
@i : Interaction {Interactions}
interaction
message
m2 : Message {Interactions} name:=@str
m2End : MessageEnd {Interactions}
messageEnd
@ICU : Lifeline {Interactions}
covered
receiveEvent
@int : Interface {Kernel}
ownedOperation
interface
existsO : Operation {Kernel} {name=@str}
{ELSE}
@int : Interface {Kernel}
interface
ownedOperation
o : Operation {Kernel} name:=@str
signature
@m2 : Message {Interactions}
@existsO : Operation {Kernel}
signature
@m2 : Message {Interactions}

Figure 6.49: Procedure generating system-to-actor messages

from the verb phrase in both cases. More precisely, we need not the object itself (which is a NounPhrase), but the Notion corresponding to this phrase. This notion has an already built business layer interface, which is used now. The message is an operation invocation, this time with parameters. Therefore the operation generation is also complicated - the operation name is built from the verb and the direct object, and both objects (or the sole for the simple case) correspond to parameters. More precisely, the operation parameter types are the DTO objects corresponding to the relevant notions (which means that this DTO object is used). The actual arguments of the message are simply the corresponding notion names.

The first rule locates the "self" lifeline. Then the new "business" lifeline is built (by the procedure bhv_CreateSystemLifeline) and the operation name is generated (by the same utility, but in a special "system-system" mode). After these preparatory actions the message is built and linked and the operation is completed with the corresponding parameters (one or two). The message is given its arguments (one or two). Note the reuse of the simple verb phrase. If the phrase in the current node is simple indeed, then the corresponding MOLA rules (the last two) build one operation parameter and one message argument. Otherwise, the rules specific to the complex case build the parameter and argument corresponding to the indirect object, then those for the direct one are added by reusing the "simple" case (fortunately, the required parameter order is namely such).

The figure 6.51 shows the "business" lifeline generation by the procedure bhv_CreateSystemLifeline.

Finally, figures 6.52, 6.53 and 6.54 show the utility procedures utl_GetPredicateText, utl_GetOperationText and utl_toCamelCase, respectively. No special comments are provided for them, just note how string processing is done in MOLA (in a way similar to programming languages).

Actually, some MOLA procedures are missing - those which clear the unused DTO objects, data access objects and business interfaces. In fact, they would be very simple - all used object instances get an appropriate link when the "using" procedure locates that instance. However, since object "using" is also not complete now, we do not provide these procedures.

Except for these explicitly stated omissions, the specified MOLA transformation is complete for the algorithm of 6.2.2 - but for the "single branch case" only, and no invocations. Certainly, there may be errors in the MOLA text. There was no possibility to test the transformation, since there is no RSL tool yet, which would provide instances according to the RSL metamodel. Building an artificial test driver was considered to be too expensive for this deliverable. The completing of this transformation, adapting it to real instance input from a tool and testing it are the issues for the next deliverables.

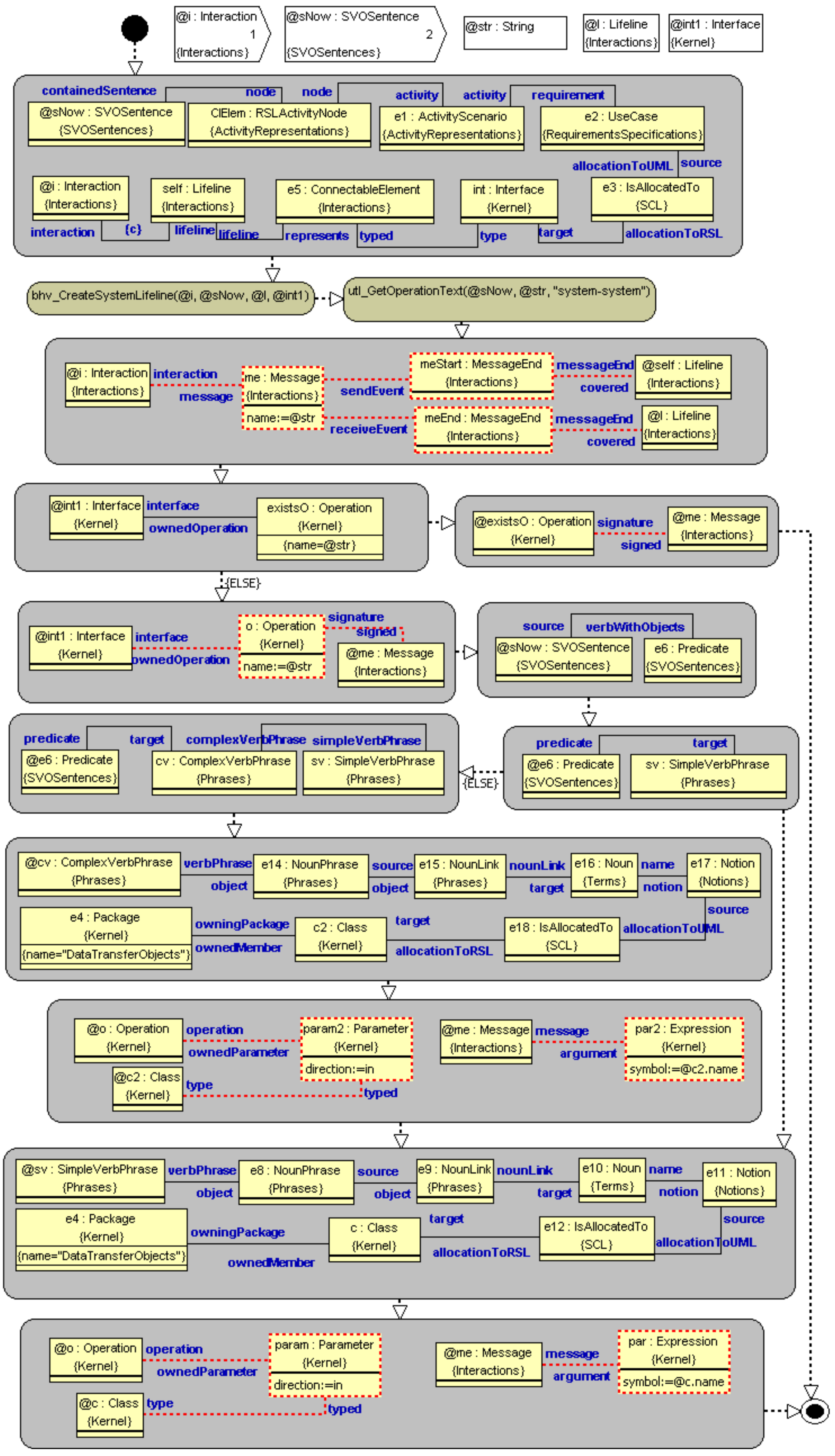

@i : Interaction 1 {Interactions}
@sNow : SVOSentence 2 {SVOSentences}
@str : String
@l : Lifeline {Interactions}
@int1 : Interface {Kernel}
bhv_CreateSystemLifeline(@i, @sNow, @l, @int1)
utl_GetOperationText(@sNow, @str, "system-system")
meStart : MessageEnd {Interactions}
meEnd : MessageEnd {Interactions}
me : Message {Interactions} name:=@str
existsO : Operation {Kernel} {name=@str}
{ELSE}
o : Operation {Kernel} name:=@str
cv : ComplexVerbPhrase {Phrases}
sv : SimpleVerbPhrase {Phrases}
e4 : Package {Kernel} {name="DataTransferObjects"}
param2 : Parameter {Kernel} direction:=in
par2 : Expression {Kernel} symbol:=@c2.name
param : Parameter {Kernel} direction:=in
par : Expression {Kernel} symbol:=@c.name

Figure 6.50: Procedure generating system-to-system messages

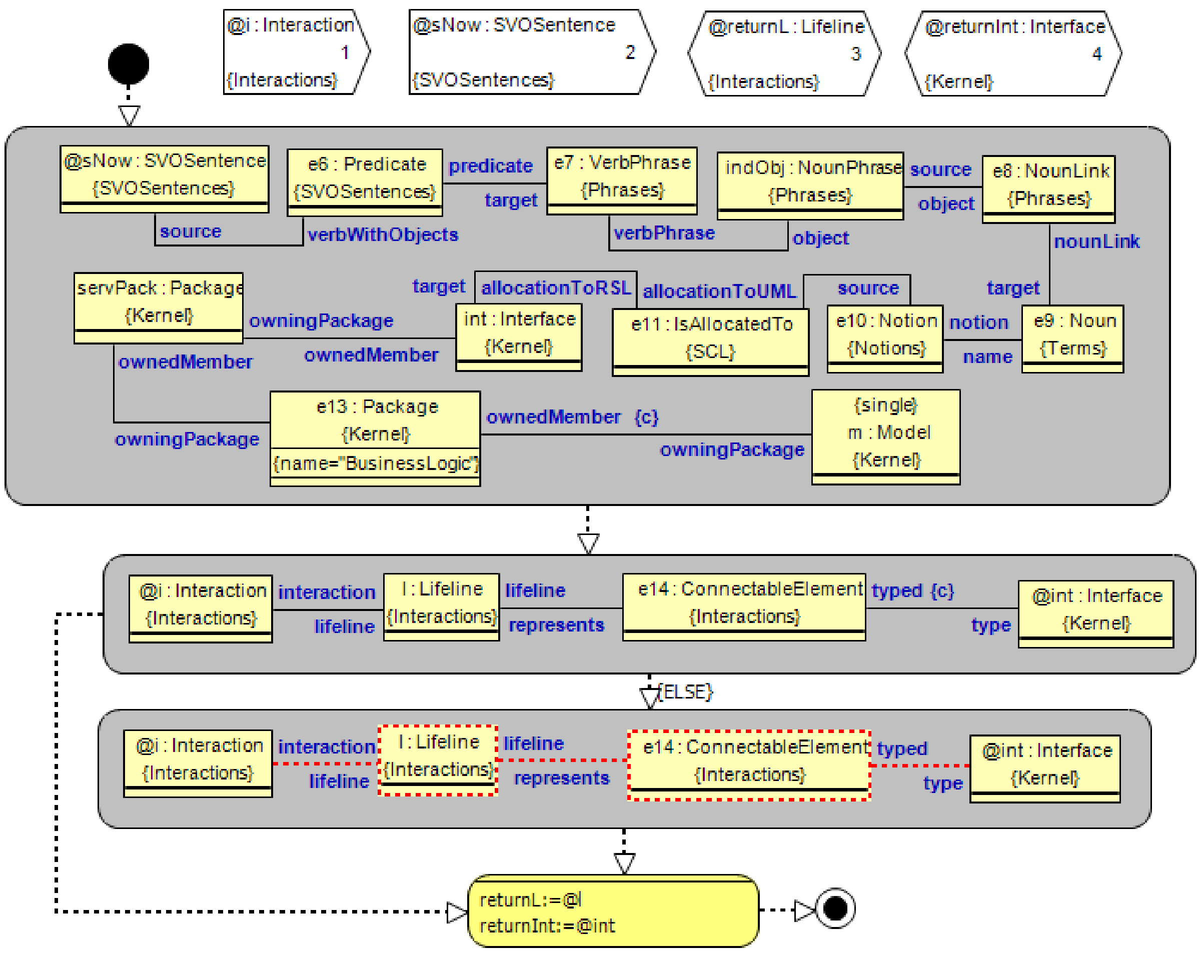

@i : Interaction 1 {Interactions}
@sNow : SVOSentence 2 {SVOSentences}
@returnL : Lifeline 3 {Interactions}
@returnInt : Interface 4 {Kernel}
@sNow : SVOSentence {SVOSentences}
e6 : Predicate {SVOSentences}
predicate
target
e7 : VerbPhrase {Phrases}
indObj : NounPhrase {Phrases}
source
object
e8 : NounLink {Phrases}
source
verbWithObjects
verbPhrase
object
nounLink
servPack : Package {Kernel}
target
allocationToRSL
allocationToUML
source
target
owningPackage
int : Interface {Kernel}
e11 : IsAllocatedTo {SCL}
e10 : Notion {Notions}
notion
e9 : Noun {Terms}
ownedMember
ownedMember
name
e13 : Package {Kernel} {name="BusinessLogic"}
ownedMember {c}
{single} m : Model {Kernel}
owningPackage
owningPackage
@i : Interaction {Interactions}
interaction
lifeline
l : Lifeline {Interactions}
lifeline
represents
e14 : ConnectableElement {Interactions}
typed {c}
type
@int : Interface {Kernel}
{ELSE}
@i : Interaction {Interactions}
interaction
lifeline
l : Lifeline {Interactions}
lifeline
represents
e14 : ConnectableElement {Interactions}
typed
type
@int : Interface {Kernel}
returnL:=@l
returnInt:=@int

Figure 6.51: Procedure generating system ("business") lifeline

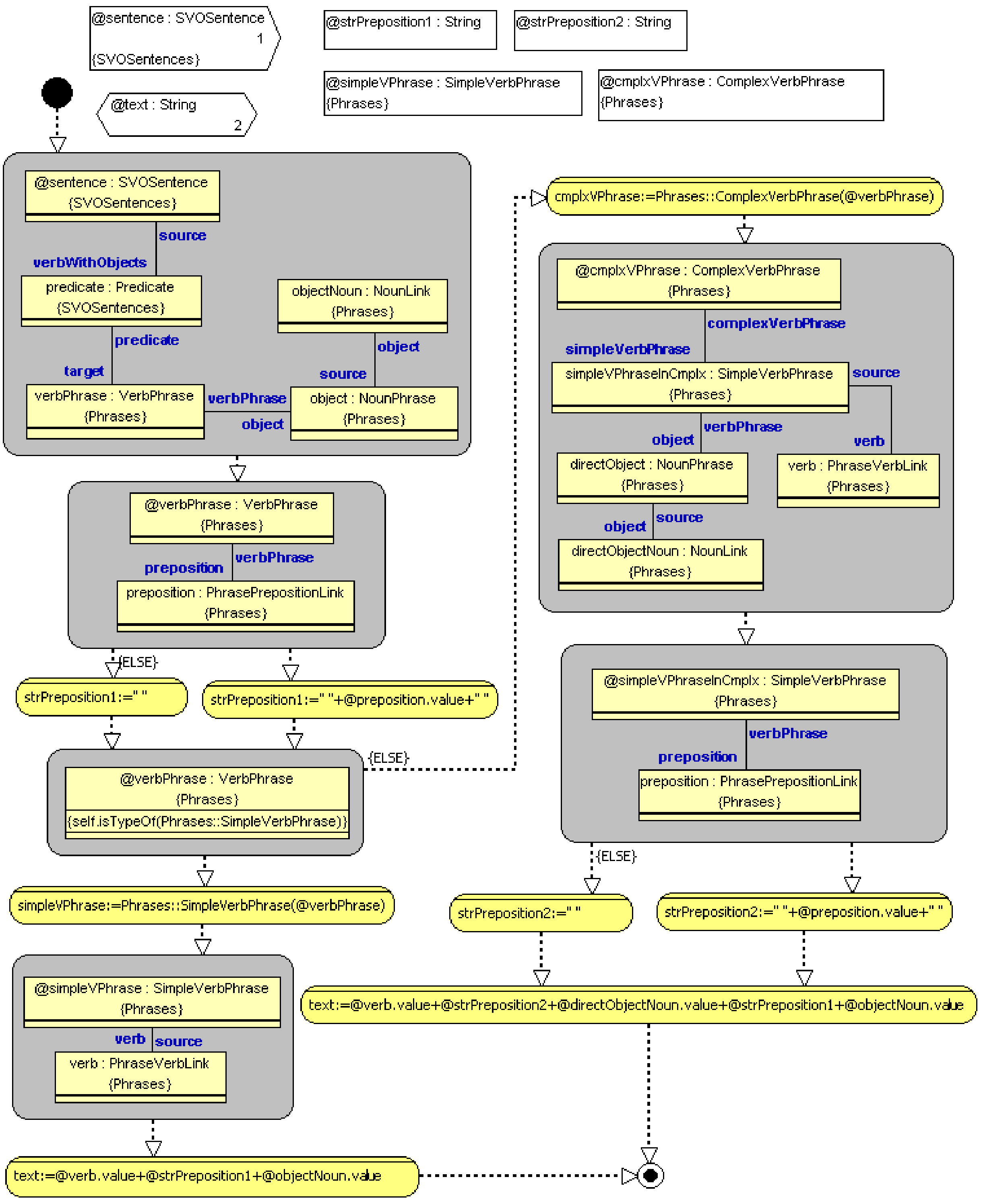

@sentence : SVOSentence
1
{SVOSentences}
@text : String
2
@strPreposition1 : String
@strPreposition2 : String
@simpleVPhrase : SimpleVerbPhrase
{Phrases}
@cmplxVPhrase : ComplexVerbPhrase
{Phrases}
@sentence : SVOSentence
{SVOSentences}
source
verbWithObjects
predicate : Predicate
{SVOSentences}
objectNoun : NounLink
{Phrases}
predicate
object
target
source
verbPhrase : VerbPhrase
{Phrases}
verbPhrase
object : NounPhrase
{Phrases}
object
@verbPhrase : VerbPhrase
{Phrases}
verbPhrase
preposition
preposition : PhrasePrepositionLink
{Phrases}
{ELSE}
strPreposition1:=" "
strPreposition1:=" "+@preposition.value+" "
{ELSE}
@verbPhrase : VerbPhrase
{Phrases}
{self.isTypeOf(Phrases::SimpleVerbPhrase)}
simpleVPhrase:=Phrases::SimpleVerbPhrase(@verbPhrase)
@simpleVPhrase : SimpleVerbPhrase
{Phrases}
verb
source
verb : PhraseVerbLink
{Phrases}
text:=@verb.value+@strPreposition1+@objectNoun.value
cmplxVPhrase:=Phrases::ComplexVerbPhrase(@verbPhrase)
@cmplxVPhrase : ComplexVerbPhrase
{Phrases}
complexVerbPhrase
simpleVerbPhrase
simpleVPhraseInCmplx : SimpleVerbPhrase
{Phrases}
source
object
verbPhrase
verb
directObject : NounPhrase
{Phrases}
verb : PhraseVerbLink
{Phrases}
object
source
directObjectNoun : NounLink
{Phrases}
@simpleVPhraseInCmplx : SimpleVerbPhrase
{Phrases}
verbPhrase
preposition
preposition : PhrasePrepositionLink
{Phrases}
{ELSE}
strPreposition2:=" "
strPreposition2:=" "+@preposition.value+" "
text:=@verb.value+@strPreposition2+@directObjectNoun.value+@strPreposition1+@objectNoun.value

Figure 6.52: Utility procedure providing verb phrase text

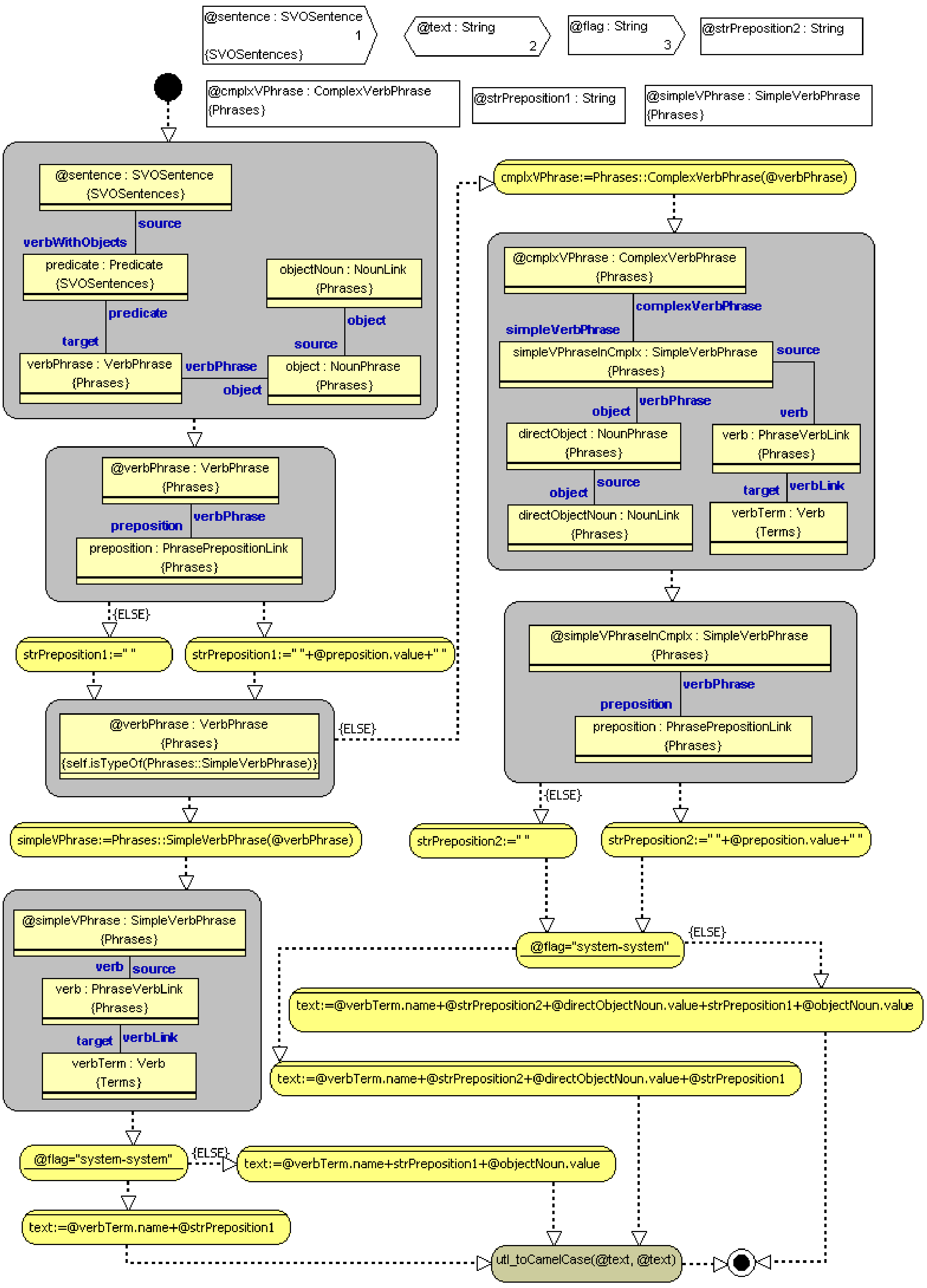

@sentence : SVOSentence 1 {SVOSentences}
@text : String 2
@flag : String 3
@strPreposition2 : String
@cmplxVPhrase : ComplexVerbPhrase {Phrases}
@strPreposition1 : String
@simpleVPhrase : SimpleVerbPhrase {Phrases}
@sentence : SVOSentence {SVOSentences}
source
verbWithObjects
predicate : Predicate {SVOSentences}
predicate
target
verbPhrase : VerbPhrase {Phrases}
verbPhrase
object
objectNoun : NounLink {Phrases}
object
source
object : NounPhrase {Phrases}
@verbPhrase : VerbPhrase {Phrases}
preposition
verbPhrase
preposition : PhrasePrepositionLink {Phrases}
{ELSE}
strPreposition1:=" "
strPreposition1:=" "+@preposition.value+" "
@verbPhrase : VerbPhrase {Phrases} {self.isTypeOf(Phrases::SimpleVerbPhrase)}
{ELSE}
simpleVPhrase:=Phrases::SimpleVerbPhrase(@verbPhrase)
@simpleVPhrase : SimpleVerbPhrase {Phrases}
verb
source
verb : PhraseVerbLink {Phrases}
target
verbLink
verbTerm : Verb {Terms}
@flag="system-system"
{ELSE}
text:=@verbTerm.name+strPreposition1+@objectNoun.value
text:=@verbTerm.name+@strPreposition1
cmplxVPhrase:=Phrases::ComplexVerbPhrase(@verbPhrase)
@cmplxVPhrase : ComplexVerbPhrase {Phrases}
complexVerbPhrase
simpleVerbPhrase
simpleVPhraseInCmplx : SimpleVerbPhrase {Phrases}
source
object
verbPhrase
verb
directObject : NounPhrase {Phrases}
verb : PhraseVerbLink {Phrases}
object
source
target
verbLink
directObjectNoun : NounLink {Phrases}
verbTerm : Verb {Terms}
@simpleVPhraseInCmplx : SimpleVerbPhrase {Phrases}
verbPhrase
preposition
preposition : PhrasePrepositionLink {Phrases}
{ELSE}
strPreposition2:=" "
strPreposition2:=" "+@preposition.value+" "
@flag="system-system"
{ELSE}
text:=@verbTerm.name+@strPreposition2+@directObjectNoun.value+strPreposition1+@objectNoun.value
text:=@verbTerm.name+@strPreposition2+@directObjectNoun.value+@strPreposition1
utl_toCamelCase(@text, @text)

Figure 6.53: Utility procedure providing operation name

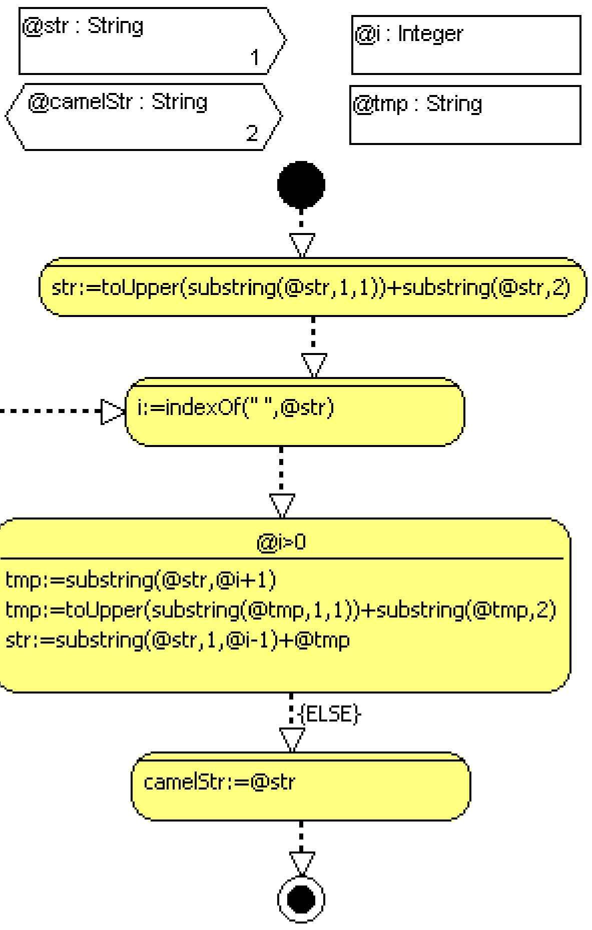

@str : String
1
@i : Integer
@camelStr : String
2
@tmp : String
str:=toUpper(substring(@str,1,1))+substring(@str,2)
i:=indexOf(" ",@str)
@i>0
tmp:=substring(@str,@i+1)
tmp:=toUpper(substring(@tmp,1,1))+substring(@tmp,2)
str:=substring(@str,1,@i-1)+@tmp
{ELSE}
camelStr:=@str

Figure 6.54: Utility procedure converting string to UpperCamelCase

Chapter 7

Transition to code in SCL

The ReDSeeDS Software Case Language (SCL) aims at combining all artefacts that comprise a software case, from requirements to code, in a structured and traceable manner. Obviously, as the result of automatic transformations, or by manual implementation, the *source code* realising the functionality of a software system is an important part of those artefacts.

A source code model in ReDSeeDS should be capable of representing programs at various levels of detail. For example, a coarse representation of a Java program could consist of source files, packages, classes and method signatures. On the other end of the scale is a detailed, fine grained, representation, where class and method bodies, declarations, statements, and even expressions and their parts are represented in the model.

Source code models can be built by transformations of design models in a *forward engineering* step, but also, they can be extracted from existing programs by *reverse engineering* techniques.

Forward engineering: When used in forward engineering, the detail level of the source code model has to be sufficient to generate real source code. For example, from a UML class diagram in a detailed design model, a *transformation* might *create representations* of Java source files. During the transformation, traceability links can be established. In a next step, this source code model can be turned into physical Java files containing source skeletons for classes, fields, and methods.

Reverse engineering: When existing source code is to be imported into a ReDSeeDS software case, a so-called *fact extractor* analyses the files and *builds a source code model* on the required level of detail. The conventions used to represent traceability links in the programming language also influence the rules of the fact extractor.

The reverse engineering approach is especially useful during the elaboration of a software case. Automatically generated source code is likely to be incomplete: Only source code fragments, or skeletons, are generated. Also, the code might contain pseudo statements which can not be compiled or executed in the target programming language. Therefore, the generated code must be completed manually. This requires changes and extensions in the source code model of a software case.

A fact extractor, usually a parser based on the grammar of the analysed language, creates an *abstract syntax graph* (ASG) of the examined program. Thus, it is a straightforward approach to represent source code in SCL as abstract syntax graphs.

ASGs can easily be annotated with source position information. When source positions are available in the model, each model element can be located in the source files. As a conseqence, a solution marking mechanism can visualize the relevant parts of software cases, for example those parts which are to be reused, directly in the code.

Traceability links

Independent from forward or reverse engineering approaches, the *traceability information* which links source code to the various SCL parts must be created and maintained. This is covered by SCL's traceability facilities, which are explained in detail in chapter 8.

Additionally, traceablity links should be *represented inside the source files*. This can be done in a programming language dependent way. In Java, for example the built-in facility for meta information, the so-called annotations, can be used to store the links in a structured way. A special annotation class defined in ReDSeeDS could be used to store unique identifiers (uid) of SCL model elements. Program elements tagged with such annotations can then be traced back to their corresponding model elements. In other languages, for example C or C++, no built-in annotation mechanism exists. When such languages are used, links could be represented by specifically formatted comments.

A quite different approach could be *not* to represent traceability links inside the source files themselves, but to keep them only in the model. Then, if existing source code is to be integrated into a software case, the links must be re-computed from source code models by queries or transformations. In this case, another set of conventions which determine the transformation rules is needed.

Source code representation in SCL

SCL is an open language which is designed to be extendable and adaptable to the specific needs of organisations and companies. Therefore, the integration of source code representations into the SCL metamodel is shown by means of an example: In section 7.1, a metamodel of the *Java 5 programming language* is presented and fitted into SCL.

The *rules governing such an integration* are made explicit in chapter 7.2 to enable software engineers to add new programming and implementation languages to SCL.

7.1 A metamodel for the Java 5 programming language

This section gives an overview of a metamodel for the Java 5 programming language[1]. The metamodel was developed by the reverse engineering group at the University of Koblenz in a reverse engineering context. It is based on a publicly available Java 5 grammar which comes along with the ANTLR[2] (Another Tool for Language Recognition).

Models corresponding to this metamodel are essentialy abstract syntax graphs (ASG) representing the source programs. The models are on the most detailed level of abstraction, where every aspect of the syntax is represented. The Java 5 metamodel also allows to build more coarse representations. This can be achieved by leaving out the packages for statements and expressions. Then, only method signatures and field declarations are represented. If the members package is also omitted, an even coarser view of the program is possible. The result would be a kind of summary model containing only packages and class names.

At the University of Koblenz, a fact extractor was implemented which analyses Java programs and generates TGraphs[Ebe81] according to the metamodel. The edges in those graphs are attributed with source position information, so that the model elements can easily be visualised, for example by highlighting regions in the original source files.

In addition to the syntax analysis, the Java 5 fact extractor also contains a *semantic analysis* component. During semantic processing, the types of expressions and the identities of various program elements are computed. As a result, the subgraphs representing different source files are linked together according to the type and visibility rules of Java.

[1]http://java.sun.com/javase/downloads/index_jdk5.jsp
[2]http://www.antlr.org

The following sections depict the general ideas of the Java 5 meta model. A detailed specification of the 90 classes of the model would go beyond the scope of this document.

7.1.1 Overview over the Java 5 metamodel

Figure 7.1 displays the package structure of the Java 5 metamodel. The main package Java5 imports the SCL package to enable access to the SCL infrastructure classes SCLElement and SCLRelationship.

The partitioning of the Java 5 metamodel into the various subpackages reflects the basic concepts of the Java programming language.

programs: The package programs at the bottom of figure 7.1 is used to represent the Java program, the source files themselves, and their connection to Java packages. The programs is part of a common infrastructure which is capable to describe the source file relationships of various programming languages. Refer to section 7.1.2 for a detailed description.

basiclanguageelements: Basic language elements of Java are combined in this packaged. The most important classes are Identifier and QualifiedName whose instances are used to describe the names of program elements. While identifier stands for a simple name, QualifiedName is the representation of the dotted combination of names. An example for a QualifiedName is `java.lang.String`.

types: The main components of Java programs are the *types* defined in the source files. Java 5 programs are a collection of ClassDefinitions, InterfaceDefinitions, EnumDefinitons, and AnnotationDefinitions.

members: *Member definitions* constitute the body of classes and interfaces in Java. The meta classes describing the various members are grouped in the members package. The primary classes are MethodDefinition and FieldDefinition which represent the respective Java concepts.

statements: This package comprises the different kinds of *Java 5 statements*, from Assert to the While loop. Statements make up the body of methods and are always contained in a, possibly nested, block.

expressions: Since the Java 5 metamodel is capable to represent every aspect of Java 5, there has to be a way to deal with *expressions*. In this package, the valid Java 5 expressions are modelled. In order to keep the model readable, the various operator expressions are

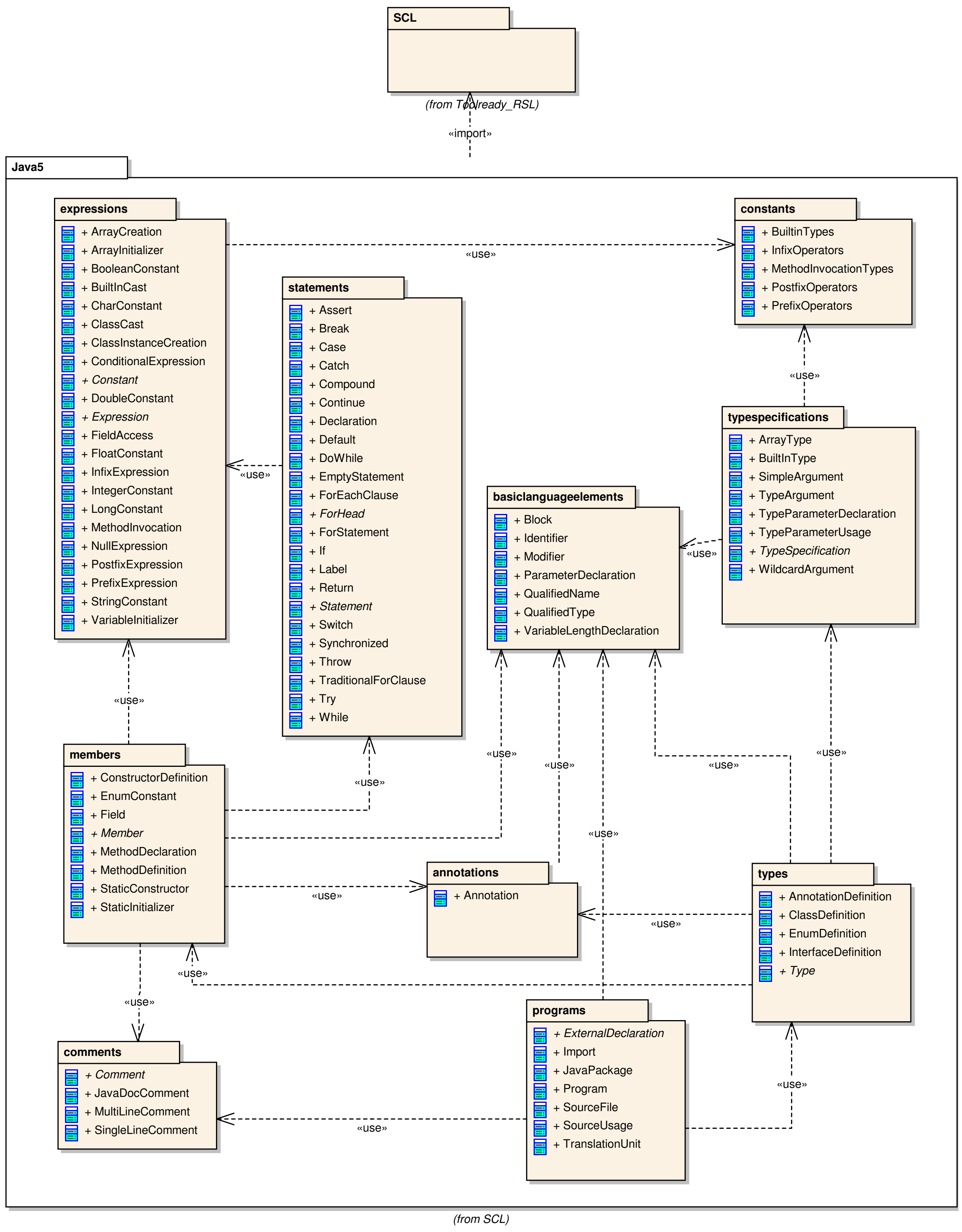


Figure 7.1: Package structure of the Java 5 metamodel

represented by only three classes: PrefixExpression, PostfixExpression, and InfixExpression. The concrete operator is stored as an attribute with a corresponding enumerated type defined in the constants package.

annotations: Meta information for Java 5 elements (classes, fields, and methods) can be stored in so-called *annotations*. Java 5 defines several annotation types already in the language specification, for example the `@Deprecated` and the `@SuppressWarning` annotations. The meta class Annotation is used to represent the concrete occurrences of annotations in a Java program. User defined annotation occurrences are handled the same way; the types package provides the class AnnotationDeclaration to record their definitions.

comments: The Java standard defines three kinds of *comments*. Single line comments start with // and reach to the end of the line, while block comments `/*...*/` can span multiple lines. The third kind is the so-called *JavaDoc comment*, a specifically formatted block comment which holds documentation tags. All comment types can be found also in the comments package of the Java 5 metamodel.

typespecifications: The classes of the typespecifications package allow to represent, among others, the generic types (type parameters as well as actual values) of the Java 5 language.

constants: Finally, the constants package contains enumeration definitions which are not directly part of the Java 5 language, but which are needed to define specific attribute domains for the metamodel. Using those enumerations keeps the metamodel simpler. Otherwise, a meta class per enumeration value would be needed, and the model would be crowded by lots of nearly identical classes.

The high level of detail of the Java 5 metamodel leads to rather big models, even for representations of apparently simple programs. To depict such a model in this document is impossible. The next section explains the representation of some top-level concepts in more detail.

7.1.2 Representaion of Java 5 programs

Java 5 programs are physically distributed over many source files. The source files contain type definitions (classes, interfaces etc.) which are logically grouped into Java packages. Figure 7.2 shows how programs are represented in the Java 5 metamodel.

In order to enable links between the source code model and the other parts of a software case in SCL, the base classes in the Java 5 model are specialisations of SCL::SCLElement[3]. Consequently, each code model element contains a unique identifier and can be referenced unambigously. Also, SCL::SCLRelationship links can be established and traceability information can be recorded.

[3]See section 8.2 for a detailed description of the SCL traceability infrastructure

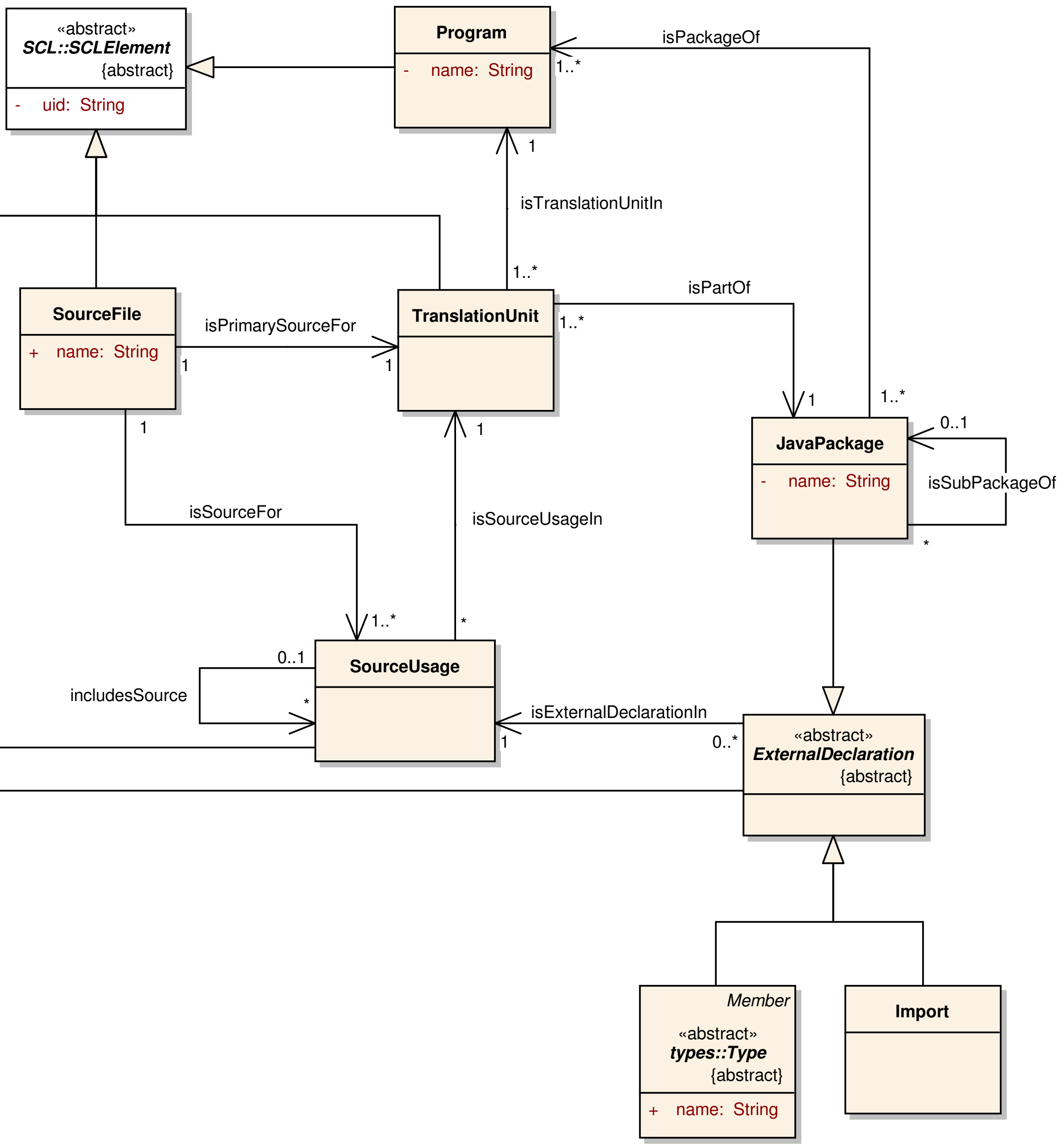


Figure 7.2: Representation of programs in the Java 5 metamodel

The classes Program, TranslationUnit, SourceFile, and SourceUsage in figure 7.2 reflect the physical structure of a program. A Program consists of TranslationUnits, which again contain so-called SourceUsages. A source usage represents the processing of a source file by the compiler.

If SCL would only be designed to contain Java models, source usages were not necessary - a Java file is a self-contained entity. However, the concepts presented emerged from experiences in reverse engineering projects, where different programming languages are considered. The source usage concept allows for representation of for example C/C++ programs, where a translation unit is combined by a preprocessor. In those languages, a source file can include many others by means of a preprocessor directive, and each source file may be included by many others - even multiple times in a single translation unit. The links isTranslationUnitIn, isPrimarySourceFor, isSourceFor, and includesSource represent those relations.

In each SourceUsage, there are a number of ExternalDeclarations. An ExternalDeclaration is a program element that is somehow 'visible' to external entities, for example Java classes, interfaces, or enumerations. The link isExternalDeclaraionIn connects those elements to their corresponding source usage. In the Java 5 metamodel, there are two kinds of external declarations: Imports and Types. Every other element, like fields, methods, etc., is contained in a Type.

In Java, a Type is not only located in a specific source file, but is also contained in a JavaPackage. Java programs consist of Java packages. Java packages are a logical grouping of types, and each package can be spread over many different source files[4]. The language allows for nesting packages.

The relation between a program, its Java packages with their content, and the corresponding source files is represented by isPackageOf and isPartOf links. The nesting of JavaPackages is mirrored by isSubPackageOf links.

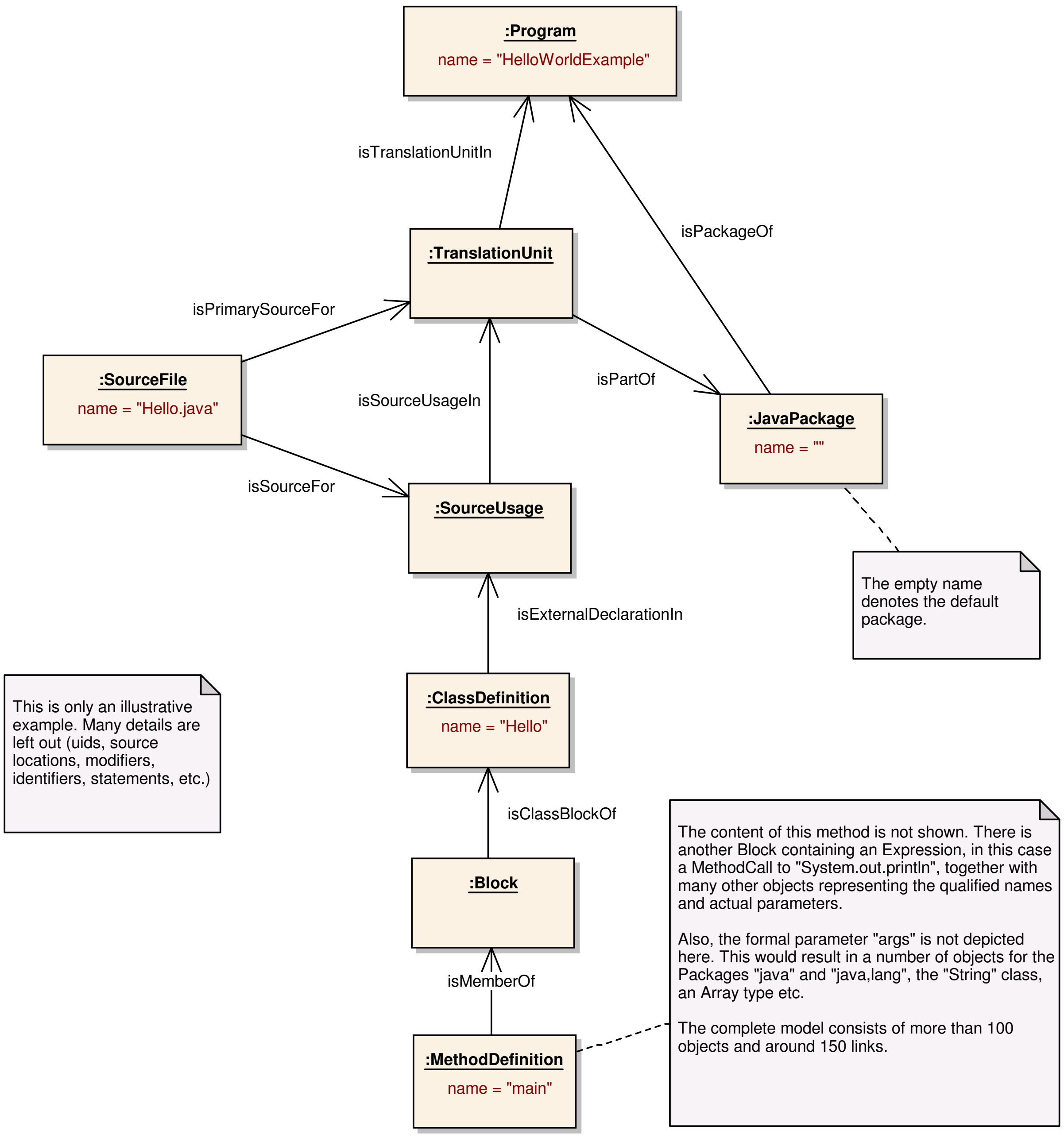
To illustrate the Java 5 metamodel, figure 7.3 shows the basic structure of a corresponding model for the well-known 'Hello world' program.

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

The complete model can not be shown in a reasonable way, since it comprises more than 100 objects and around 150 links. Of course, this level of detail might be too fine-grained for a software case repository. By configuration of the fact extractor, the granularity can be reduced to extract only a skeleton of the program, which is of similar complexity as figure 7.3.

The next section defines the rules for integrating other implementation languages into SCL.

[4]By convention, all source files making up a package should be located in one directory, and the directory structure should reflect the package structure. But this is not required by the language. It is also possible to store the source files of several packages in a common folder.



The content of this method is not shown. There is another Block containing an Expression, in this case a MethodCall to "System.out.println", together with many other objects representing the qualified names and actual parameters.

Also, the formal parameter "args" is not depicted here. This would result in a number of objects for the Packages "java" and "java,lang", the "String" class, an Array type etc.

The complete model consists of more than 100 objects and around 150 links.

Figure 7.3: Example model for the ‘Hello world’ program

7.2 Rules for the integration of source code models into SCL

Similar to the Java 5 metamodel presented in section 7.1, any other programming language can easily be integrated into the ReDSeeDS Software Case Language.

As shown by the Java example, a source code model can be developed along the grammar of the required language. Possibly, EBNF notations could even be converted automatically into an equivalent metamodel for abstract syntax graphs. But also already existing metamodels for arbitrary purposes can be used.

There are only three basic rules for language metamodels to be integrated in SCL:

1. The metamodel must adhere to the metamodelling style rules defined in [RBS+07]. This is required because the code model must be stored and processed together with all other SCL parts. The style rules ensure that the various ReDSeeDS components like repositories, querying technology, and transformation machinery will seamlessly work with the new language models.
2. Any entity that must be referenced by traceability links must be a specialisation of the meta class SCL::SCLElement. This ensures that the element gets a unique identifier, and that links can be connected from or to the other parts of the software case.
3. Any relationship inside the metamodel for the new language that must be identifiable or referenced must be a specialisation of SCL::SCLRelationship.

However, it is recommended that the structure used to represent programs and source files should be shared by all code metamodels. As a reference, the classes and associations defined in the Java 5 model in section 7.1.2 can be used.

In the following chapter, the SCL traceability features are explained in detail.

Chapter 8

Traceability facilities in SCL

Traceability refers to "the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match" [IEE90]. A traceability link constitutes the manifestation of such a relationship. In *ReDSeeDS*, they connect two instances of SCLElement (see section 2.1).

This chapter describes in detail the traceability facilities provided by the Software Case Language. The first section gives a general overview of the role traceability plays in the reuse of Software Cases. Section 8.2 extends the basic model for traceability links introduced in section 2.1 by specific link types. Possible visualisations for traceability links are discussed in section 8.3.

8.1 Traceability as enabler of reuse

In [ACC$^+$02], Antoniol et al. give a sample of areas of applications for traceability, e.g. program comprehension, impact analysis or reuse of existing software.

The main goal of traceability links in *ReDSeeDS* is to promote reuse: A Software Case contains links connecting not only requirements to their satisfying architectural or design artifacts, but also links between related requirements, between design elements and source code etc. Starting from requirements of an archived Software Case which were identified to be similar to those of a currently active project, it is possible to traverse these links. The reached elements are candidates for reuse. These candidates could be, for instance, components, single classes, or even snippets of code.

Traceability links in *ReDSeeDS* also contribute to impact analysis. Since there exist different link types, each one with special semantics, developers are able to identify possible effects of changes more precisely.

ReDSeeDS does not only allow for the manual insertion of traceability links by requirements engineers, software architects, programmers etc., but also their automatic generation due to the application of model transformations. Consider a functional requirements package which is transformed to a component of the application logic layer, according to section 6.2. This transformation leaves a trace of type IsAllocatedTo (see section 8.2.3) documenting that the component is intended to satisfy the requirement.

8.2 Integration of traceability links in SCL

This section gives descriptions of the various traceability link types contained in the SCL. Metamodel excerpts serve to depict the links' abstract syntax. The semantics are given in natural language enhanced by concrete examples for their usage. Most of the link types are based on the work of Ramesh and Jarke [RJ99], Ebner and Kaindl [EK02] and the UML Specification [Obj07] by the OMG.

Figure 8.1 shows TraceabilityLink, the common superclass of every traceability link type. It is a specialisation of SCLRelationship and therefore inherits the uid attribute and the associations connecting it to its source and target SCLElements. Two instances of SCLElement linked by a TraceabilityLink are supposed to be related in some way. The exact nature of this relationship is determined by the applied specialisation of TraceabilityLink. Every TraceabilityLink is contained in exactly one Software Case.

The isGenerated attribute of TraceabilityLink indicates whether a link is automatically generated.

As the SCL is extendable by new sublanguages, so it has to be possible to extend the traceability facilities. This can be accomplished by the inclusion of new traceability link types in the specialisation hierarchy of TraceabilityLink. Such a link type must then be associated to its source and target SCLElements, subsetting the source and target inherited from SCLRelationship.

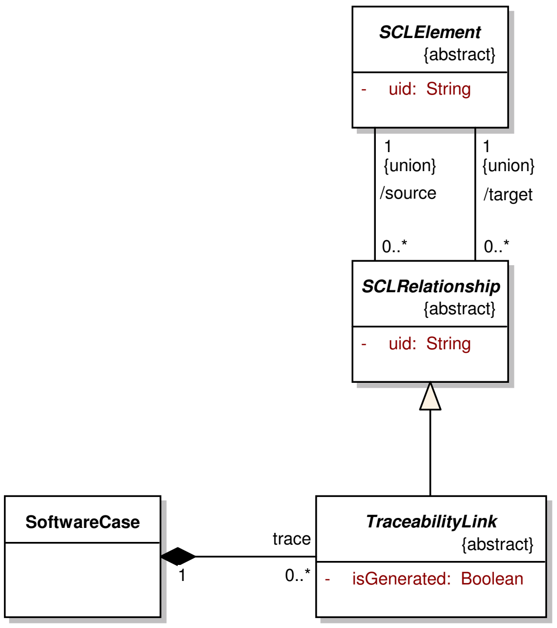


Figure 8.1: TraceabilityLink as a subclass of SCLRelationship

The existing specialisations of 8.1 are discussed in the following sections. While section 8.2.1 presents link types inherent in the RSL, section 8.2.2 deals with the UML's traceability link types. The last section 8.2.3 shows link types associated with elements of two different sublanguages of the SCL.

8.2.1 RSL traceability links

The RSL offers a variety of different traceability link types in order to model relationships between requirements (see figure 8.2). Their semantics is given below. All of these link types are specialisations of TraceabilityLink and RSL :: RequirementRelationship. Therefore they connect to one Requirement in the role of source and to one Requirement in the role of target. The link types Fulfils and MakesPossible are exceptions: while the former connects to an UseCase as source, the latter has an UseCase as target.

ConflictsWith

A ConflictsWith link connecting two Requirements points to some contradiction existing between them.

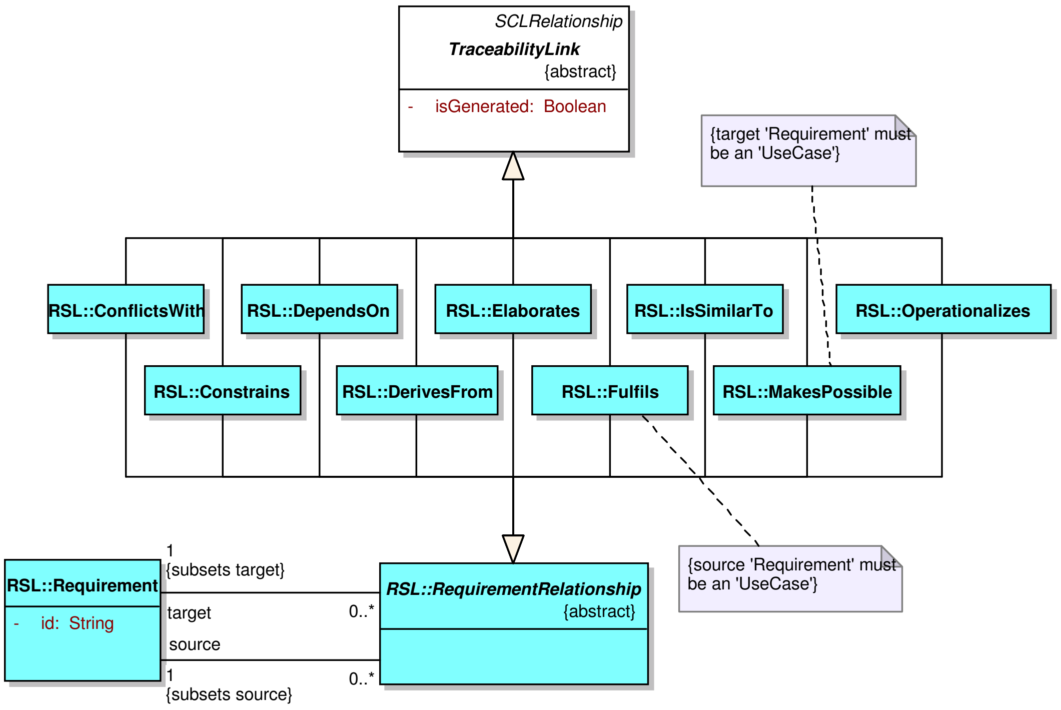


Figure 8.2: RSL traceability links

Example

Source: Fitness Club System shall register every activity.
ConflictsWith
Target: Employees may use facilities without registration.

Constrains

A Constrains link states that the source Requirement puts some restrictions upon the target Requirement. In most cases, the source is a ConstraintOnSystem or a ConstraintOnProcess.

Example

Source: User data transmission shall be encrypted with SHA.
Constrains
Target: Reservations should be accessible through the internet.

DependsOn

This link type conveys that a change of the target Requirement may cause a change of the source Requirement. Often the target Requirement constitutes a prerequisite for the source Requirement.

Example

Source: Customers shall be able to reserve facilities.
DependsOn
Target: Fitness Club System shall allow for reservations.

DerivesFrom

DerivesFrom indicates that the conception of the source Requirement has its origins in the target Requirement.

Example

Source: Fitness Club System shall register payments.
DerivesFrom
Target: Fitness Club System shall automatically bill customers.

Elaborates

The source Requirement of an Elaborates link extends the target Requirement by adding more details or offering explanations.

Example

Source: Fitness Club System shall print bills on the basis of customer activities.
Elaborates
Target: Fitness Club System shall automatically bill customers.

Fulfils

A link of this type connects a source UseCase to a target Requirement. The UseCase is supposed to show how the system's feature described by the target Requirement can be accessed by a user.

Example

Source: Browse an offer and reserve.
Fulfils
Target: Customers shall be able to reserve facilities.

IsSimilarTo

Links of this type show that the connected Requirements point to the same issue and differ only slightly, possibly with regard to their wording.

Example

Source: Fitness Club System shall register every entry to a facility.
IsSimilarTo
Target: If a customer enters a facility, the Fitness Club System shall register the entry.

MakesPossible

MakesPossible denotes a link between a source Requirement and a target UseCase: fulfilment of the source enables the system's behaviour described by the target.

Example

Source: System shall show timetables for facilities.
MakesPossible
Target: Browse timetables.

Operationalizes

The source Requirement of an Operationalizes link adds some concrete information to a rather general target Requirement. Typically, the target is a ConstraintOnSystem or ConstraintOnProcess.

Example

Source: Data storage server shall be protected with firewall certified by ICSA.
Operationalizes
Target: Customer data shall be stored securely.

8.2.2 UML traceability links

Traceability links in UML are modelled via the class Dependency, its subclasses and associated stereotypes. In SCL, a generalisation relationship is created from UML :: Dependency to SCL :: TraceabilityLink, as shown in figure 8.3. Furthermore, only binary dependencies are allowed when using the SCL.

See the specification of the UML Superstructure [Obj07] for further details on syntax and semantics of UML traceability link types.

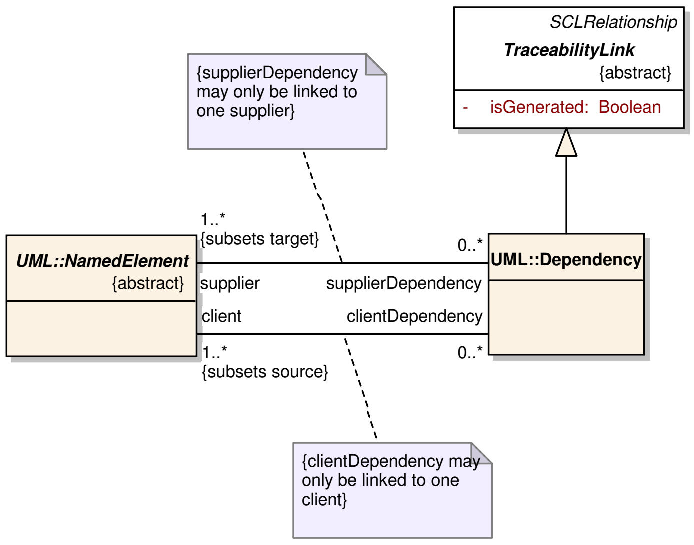


Figure 8.3: UML traceability links

8.2.3 Inter-language traceability links

The traceability link types specified in this section facilitate the connection of elements originating from different sublanguages of the SCL. These link types are direct specialisations of TraceabilityLink (see figure 8.4). See below for more details.

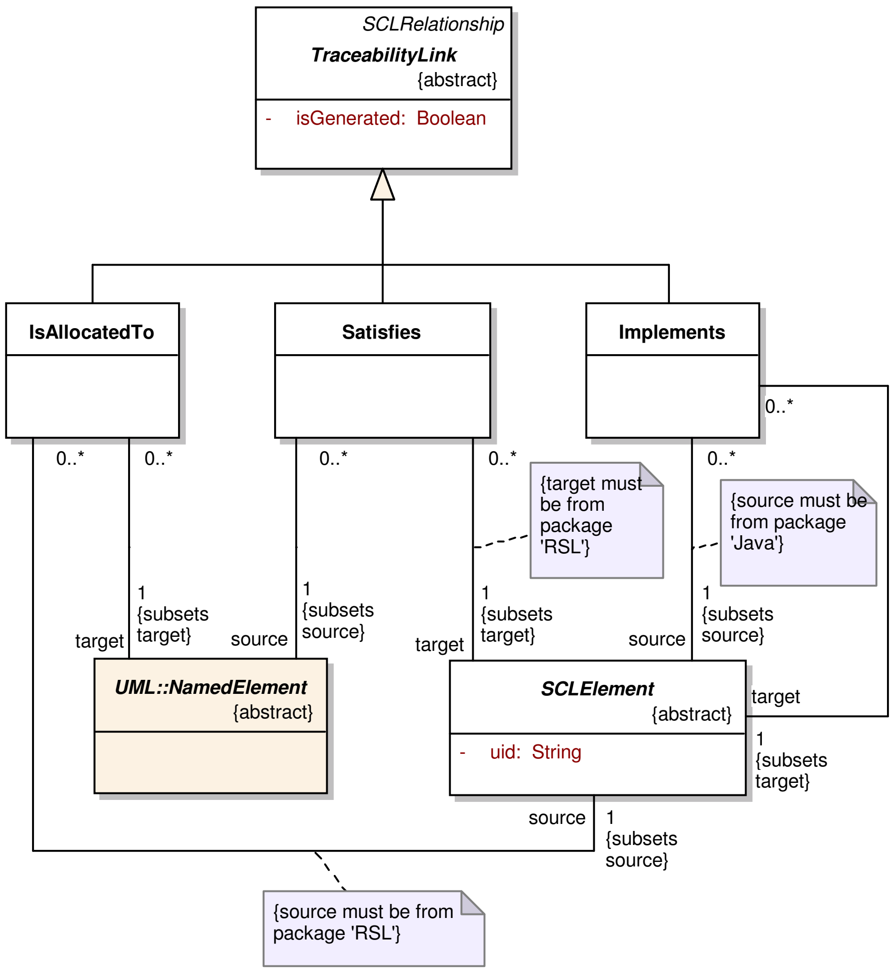


Figure 8.4: Inter-language traceability links

Implements

An Implements link connects a piece of source code with an arbitrary target SCLElement, stating that this piece of code is the representation of the SCLElement in terms of programming language. Upon the integration of other programming languages into SCL, the constraint stating that the SCLElement must originate from the Java5 package could be extended in order to include these new languages.

IsAllocatedTo

This link type serves to allocate a source SCLElement, originating from the RSL, to a target UML :: NamedElement which, as a part of the system's architecture or design, is supposed to realize the SCLElement.

These links shall be automatically generated by the transformations given in chapter 6, with the transformation's origin as source and the transformation's result as target of the link. Manual insertion is permitted, though.

See figure 8.5 for an example of this link type's usage.

Satisfies

A Satisfies link with a UML :: NamedElement as source and a SCL :: Element, originating from the RSL, as target implies that the former contributes to the realisation of the SCL :: Element.

As an example of its usage, a Satisfies link could be manually created as inversion of every IsAllocatedLink, if the automatically generated architectural or design element is ensured to realise the linked RSL :: Element. Additional Satisfies links with the same target may exist if other elements contribute to the RSL: Element's realisation, too.

For an example, refer to figure 8.5.

8.3 Possible visualisation of traceability links

In general, it is conceivable to visualise traceability links in two different forms. In a model representing a single software case, i.e. an instance of the SCL's meta-model, links are depicted as lines connecting related artifacts. Automatically generated links have to be marked. Figure 8.5 shows a possible notation.

Another possibility would be to present traceability links as a condensed table listing all links together with their source and target artifacts.

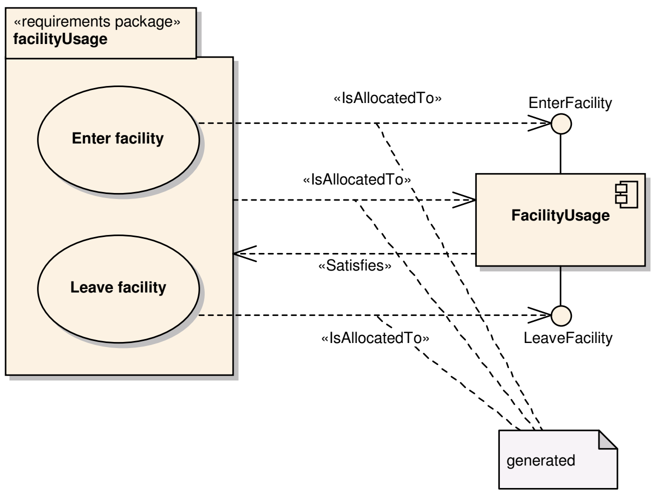


Figure 8.5: Traceability link visualisation example

Bibliography

[ACC$^+$02] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[BRJ05] G Booch, J Rumbaugh, and I Jacobson. *Unified Modeling Language User Guide, The (2nd Edition)*. Addison-Wesley, 2005.

[DP03] Asa G. Dahlstedt and Anne Persson. Requirements interdependencies - moulding the state of research into a research agenda. In *Proceedings of the Ninth International Workshop on Requirements Engineering: Foundation for Software Quality, Klagenfurt/Velden, Austria (REFSQ'03)*, pages 71–80, 2003.

[Ebe81] Jürgen Ebert. *Effiziente Graphenalgorithmen*. Akademische Verlagsgesellschaft, 1981.

[EK02] Gerald Ebner and Hermann Kaindl. Tracing all around in reengineering. *IEEE Software*, 19(3):70–77, 2002.

[IEE90] IEEE. *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990*, 1990.

[KBC04] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language MOLA. In *Lecture Notes in Computer Science, Springer, v. 3599, 2005. Proceedings of MDAFA 2004*, 2004.

[KWW03] A G Kleppe, J B Warmer, and Bast W. *MDA Explained, The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, 2003.

[Let02] Patricio Letelier. A framework for requirements traceability in uml-based projects. In *Proceedings of 1st International Workshop on Traceability in Emerging Forms of Software Engineering (In conjunction with the 17th IEEE International Conference on Automated Software Engineering), Edinburgh, U.K.*, pages 173–183, September 2002.

[MM03] Joaquin Miller and Jishnu Mukerji, editors. *MDA Guide Version 1.0.1, omg/03-06-01*. Object Management Group, 2003.

[MSUW04] Stephen J Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model Driven Architecture*. Addison-Wesley, 2004.

[Obj05a] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification, ptc/05-11-01*, 2005.

[Obj05b] Object Management Group. *Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04*, 2005.

[Obj06] Object Management Group. *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*, 2006.

[Obj07] Object Management Group. *Unified Modeling Language: Superstructure, version 2.1.1 (non-change bar), formal/07-02-05*, 2007.

[Pol72] Solomon L. Pollack. *Decision tables*. John Wiley and Sons Inc, 1972.

[RBS+07] Volker Riediger, Daniel Bildhauer, Hannes Schwarz, Audris Kalnins, Agris Sostaks, Edgars Celms, Michal Śmiałek, and Ambroziewicz. Software case meta-model definition. Technical report, ReDSeeDS (Requirements-Driven Software Development System), 2007.

[RJ99] Bala Ramesh and Matthias Jarke. Towards reference models for requirements traceability. Technical report, Georgia State University, Atlanta, 1999.

[Ś05] Michał Śmiałek. *Zrozumieć UML 2.0. Metody modelowania obiektowego*. Helion, 2005.