Requirements Specification Language Definition

Defining the ReDSeeDS Languages

Deliverable D2.4.1, version 1.00, 28.02.2007

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

Requirements Specification Language Definition

Defining the ReDSeeDS Languages

Workpackage	WP2
Task	T2.4
Document number	D2.4.1
Document type	Deliverable
Title	Requirements Specification Language Definition
Subtitle	Defining the ReDSeeDS Languages
Author(s)	Hermann Kaindl, Michał Śmiałek, Davor Svetinovic, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Hannes Schwarz, Daniel Bildhauer, John Paul Brogan, Kizito Ssamula Mukasa, Katharina Wolter, Thorsten Krebs
Internal Reviewer(s)	Michał Śmiałek, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, John Paul Brogan, Hermann Kaindl, Sevan Kavaldjian, Roman Popp
Internal Acceptance	Project Board
Location	https://svn.redseeds.eu/svn/redseeds/1_DeliverablesSpace/WP2_Requirements_specification_language/D2.4.01/ReDSeeDS_D2.4.1_Requirements_Specification_Language_Definition.pdf
Version	1.00
Status	Final
Distribution	Public

28.02.2007

History of changes

Date	Ver.	Author(s)	Change description
13.02.2007	0.01	Hermann Kaindl (TUW)	Proposition of ToC
15.02.2007	0.02	Hannes Schwarz (UKo)	Added sections Constrained language representations, Representation sentences, SVO sentences, Phrases, Terms
15.02.2007	0.03	Hermann Kaindl (TUW)	Added content for executive summary
16.02.2007	0.04	Hermann Kaindl (TUW)	Added content for Requirements on RSL
16.02.2007	0.05	Albert Ambroziewicz (WUT)	Transferred and updated UIBehaviour description; transferred sections 1.2&1.5
16.02.2007	0.06	Albert Ambroziewicz (WUT)	Added Profiles appendix and profile description for UI devices
16.02.2007	0.07	Michał Śmiałek (WUT)	Proposition of content for Chapter 6, update in Chapter 7
16.02.2007	0.08	Albert Ambroziewicz, Jacek Bojarski (WUT)	Changed document structure
16.02.2007	0.09	Daniel Bildhauer (UKo)	Added sections Natural language representations, Interaction representation, System, Actors, Basic domain elements
16.02.2007	0.10	John Paul Brogan (HWU)	Added content for Document Scope
16.02.2007	0.11	John Paul Brogan (HWU)	Added content for Related work and relations to other documents
16.02.2007	0.12	John Paul Brogan (HWU)	Added content for Structure of Document
16.02.2007	0.13	Daniel Bildhauer (UKo)	Restructuring of Interaction representations
16.02.2007	0.14	Kizito Ssamula Mukasa (Fraunhofer)	Added profile for user interface elements
16.02.2007	0.15	Hermann Kaindl (TUW)	Added content for Domain representations using conceptual models

Date	Ver.	Author(s)	Change description
16.02.2007	0.16	Hermann Kaindl (TUW)	Added content for Thesaurus (based on text by Markus Nick)
16.02.2007	0.17	Hermann Kaindl (TUW)	Added content for Conclusion
17.02.2007	0.18	Jacek Bojarski (WUT)	Added content for Activity sentences and Activity sentence constructs
19.02.2007	0.19	Albert Ambroziewicz (WUT)	Added content for Kernel package description
19.02.2007	0.20	John Paul Brogan (HWU)	Added content for Relations to UML and SysML
20.02.2007	0.21	Tomasz Straszak (WUT)	Added content for TermsRelations section and updated Terms section
20.02.2007	0.22	Kizito Ssamula Mukasa (Fraunhofer)	Added content for Chap 15
21.02.2007	0.23	Daniel Bildhauer (UKo)	Finished all sections dealing with interaction representations
21.02.2007	0.24	Albert Ambroziewicz, Tomasz Straszak (WUT)	Major changes in User interface elements section
21.02.2007	0.25	John Paul Brogan (HWU)	Updated content for Relations to UML and SysML and added relevant references
21.02.2007	0.26	Hannes Schwarz (UKo)	Added section 12.1
21.02.2007	0.27	Wiktor Nowakowski (WUT)	Added content for Chapter 11
22.02.2007	0.28	Daniel Bildhauer (UKo)	Major changes to actors and system elements sections.
22.02.2007	0.29	Jacek Bojarski (WUT)	Finished all sections dealing with activity representations
23.02.2007	0.30	Daniel Bildhauer (UKo)	Removed deprecated content and replaced figures
23.02.2007	0.31	Albert Ambroziewicz (WUT)	Added content for Notions section
23.02.2007	0.32	Jacek Bojarski (WUT)	Added content for Scenario sentences section
23.02.2007	0.33	Tomasz Straszak (WUT)	Added content for Domain elements section
23.02.2007	0.34	Hannes Schwarz (UKo)	Added content for Requirement representations section
23.02.2007	0.35	Daniel Bildhauer (UKo)	Updated and replaced figures

Date	Ver.	Author(s)	Change description
23.02.2007	0.36	Katharina Wolter (UH)	Small additions and corrections in Chapter 7,14
23.02.2007	0.37	John Paul Brogan (HWU)	Added examples
23.02.2007	0.38	Wiktor Nowakowski (WUT)	Revised and updated descriptions in "Overview" sections for Chapters 11-13
24.02.2007	0.39	Michał Śmiałek (WUT)	Typographical, language and content (slight) changes made throughout the document
24.02.2007	0.40	Wiktor Nowakowski (WUT)	Updated description in "Overview" sections for Chapter 14
26.02.2007	0.41	Davor Svetinovic (TUW)	Completed all TUW sections
26.02.2007	0.42	Hannes Schwarz (UKo)	Added list of abbreviations (based on input from Sevan Kavaldjian and Roman Popp)
26.02.2007	0.43	Hermann Kaindl (TUW)	Cleaning-up conceptual Requirements Model chapter
26.02.2007	0.44	Hermann Kaindl (TUW)	Added text to Requirements Representation Model chapter
26.02.2007	0.45	Hermann Kaindl (TUW)	Cleaning-up Discussion chapter
27.02.2007	0.46	Albert Ambroziewicz (WUT)	Corrections in interaction sentences sections
27.02.2007	0.47	Tomasz Straszak (WUT)	Introducing Terminology instead of Dictionary and Thesaurus
27.02.2007	0.48	Jacek Bojarski (WUT)	Revision and small corrections in document
27.02.2007	0.49	Kizito Ssamula Mukasa (Fraunhofer)	Revised chap 8 and 15
27.02.2007	0.50	John Paul Brogan (HWU)	Correct english revision and small document corrections
27.02.2007	0.51	Kizito Ssamula Mukasa (Fraunhofer)	Revised UI Profile
27.02.2007	0.52	Daniel Bildhauer (UKo)	Small corrections and updates
27.02.2007	0.53	Hermann Kaindl (TUW)	Clean-up
28.02.2007	0.54	Katharina Wolter & Thorsten Krebs (UH)	Improvements in Chapter 7 (e.g. Terminology)
28.02.2007	1.00	Hermann Kaindl (TUW)	Finalisation

Summary

Requirements specification languages are abundant in the field of Requirements Engineering. However, most of them focus on formal representation only and are not used much in practice. Others provide a subset of natural language only and do not provide means for conceptual modelling. So, natural language is still the most widely used language for writing requirements specifications in practice. Generally, requirements specification languages do not integrate user-interface specifications, although requirements and user interfaces have a lot to do with each other.

Therefore, we defined a new language, the *ReDSeeDS Requirements Specification Language (RSL)*. Our approach is intended to be comprehensive for practical use and includes, therefore, even unconstrained natural language. RSL integrates descriptions — constrained and unconstrained —, conceptual modelling — based on object-oriented ideas — and even user-interface specifications. RSL is, however, not simply an aggregation of existing concepts and language constructs. It has several distinguished and even unique features.

The behavioural part of RSL distinguishes between Functional and Behavioural Requirements. While the former specify the required effects of some system, the latter specify required behaviour across the system border, in the form of Envisioned Scenarios. Functional Requirements are further specialised into Functional Requirements on Composite System and Functional Requirements on System to be built. The former are fulfilled by an Envisioned Scenario, while the functions of the latter will make its execution possible. Related Envisioned Scenarios together make up a Use Case.

The structural part of RSL deals with models and descriptions of objects existing in the domain (environment) of the software system to be built — *domain objects*. These objects are part of a conceptual Domain Model (to-be). In addition, the concepts can (and should) be described in a defined vocabulary with phrases, containing terms which are organised in a terminology representation that integrates a dictionary with a thesaurus. RSL is the first language that integrates

conceptual modelling with thesaurus features. The descriptions facilitate a better understanding of the concepts, which in turn facilitates a better understanding of the requirements.

We distinguish strictly between requirements and *representations* of requirements. Strictly speaking, only the latter can actually be reused. Requirements representations can be *descriptive* or *model-based*, and our RSL language makes this distinction explicit. The former describe the needs of certain requirements, while the latter represent models of the system to be built. A requirement is then to build a system like the one modelled.

The user-interface part of RSL contains language features for specifying user-interface elements and their dynamics. It deals with descriptions of various user-interface elements that can express various graphical or other types of elements existing in a user interface. It also includes user-interface storyboards that show dynamic change in the user interface.

Based on the previous deliverables D2.1, D2.2 and D2.3, this deliverable contains a comprehensive description of RSL. First, it gives a conceptual overview and explanation of the approach and the language. In the second part, it provides a complete language reference including concrete syntax.

Table of contents

List of figures

Chapter 1

Scope, conventions and guidelines

1.1 Document scope

This document provides a conceptual overview, and defines coherent syntax and semantics for the Requirements Specification Language (RSL). This definition is required to aid the construction of accurate requirements specifications in the form of descriptive or model-based representations.

The conceptual overview of the RSL explains the approach taken to allow for describing functional, behavioural, structural and user interface requirements and how functional, behavioural, structural and user interface requirements can be represented in the language. Structural requirements are meant as vocabularies and thesauruses or ontologies containing domain elements, including terms used in the domain and their descriptions. User Interface requirements are meant as a means to specify user interface elements such as menus or screens which can be determined from acquired user requirements. Furthermore, the user interface requirements language provides constructs for specifying the intended behaviour of user interface elements.

This document then presents the detailed RSL Reference which covers the definitions for Requirements, Requirements Representations, Requirement Representation Sentences and Domain Elements with User Interface Elements. This reference explains the syntax of the language in its abstract form (using a meta-model) and in its concrete form (using concrete examples of language usage). The semantics of all the RSL language constructs are also defined. The definitions for Requirements, Domain Elements and User Interface Elements describe the required language constructs that allow for depicting individual requirements and elements of the domain vocabulary. This explains how to structure requirements, domain elements and user in-

terface elements into full requirements specifications and full vocabularies respectively. It also defines possible relationships between requirements domain elements, including the system under development, actors and user interface considerations. Such relationships are presented graphically through appropriate diagrams.

Moreover, the reference for Requirements Representations, Domain Elements and User Interface Elements defines all the representations of requirements possible to be expressed in the RSL. These include textual, descriptive representations in natural or constrained language and schematic, model-based representations mostly derived from UML. The reference for Domain Elements defines top-level, general representations for all the elements' constructs of the language. It also defines how to express phrases and terms that can be used for representing Domain Elements. This part of the language is mostly textual but also includes some graphical variants.

Within its given definition, the RSL uses hyperlinks as basic facilitators of coherence. This allows for building a requirements specification where behavioural and quality requirements are based on the domain vocabulary, thus greatly enhancing the possibility to reuse it in the future. The Representation Sentences define the smallest "building blocks" of the RSL, ie. sentences. These sentences allow and usually necessitate for extensive use of hyperlinks to the domain vocabulary. Apart from natural language sentences, several types of controlled, structured language sentences are defined. These are mostly based on the Subject-Verb-Objects SVO(O) grammar. Finally, The description of user interface elements and associated behaviour representation constructs is shown to demonstrate a need for a compromise between the user interface elements abstract and concrete syntax as well as the user interface elements semantics language construct components.

1.2 Approach to language definition and notation conventions

1.2.1 Meta-modelling

The Requirements Specification Language is defined using a meta-model. The meta-model is a model of models, where a model of a system is a description or specification of that system and its environment for some known task. A model is often presented as a combination of drawings and text. The text may be in a modelling language or in a natural language (adapted from [MM03]).

The meta-model can be treated as a definition of a language in which models can be expressed properly. A Meta-model sets well-formedness rules for models. A model has to comply with the meta-model of the language it uses. For instance, a UML model [Obj05b] has to comply with the UML meta-model.

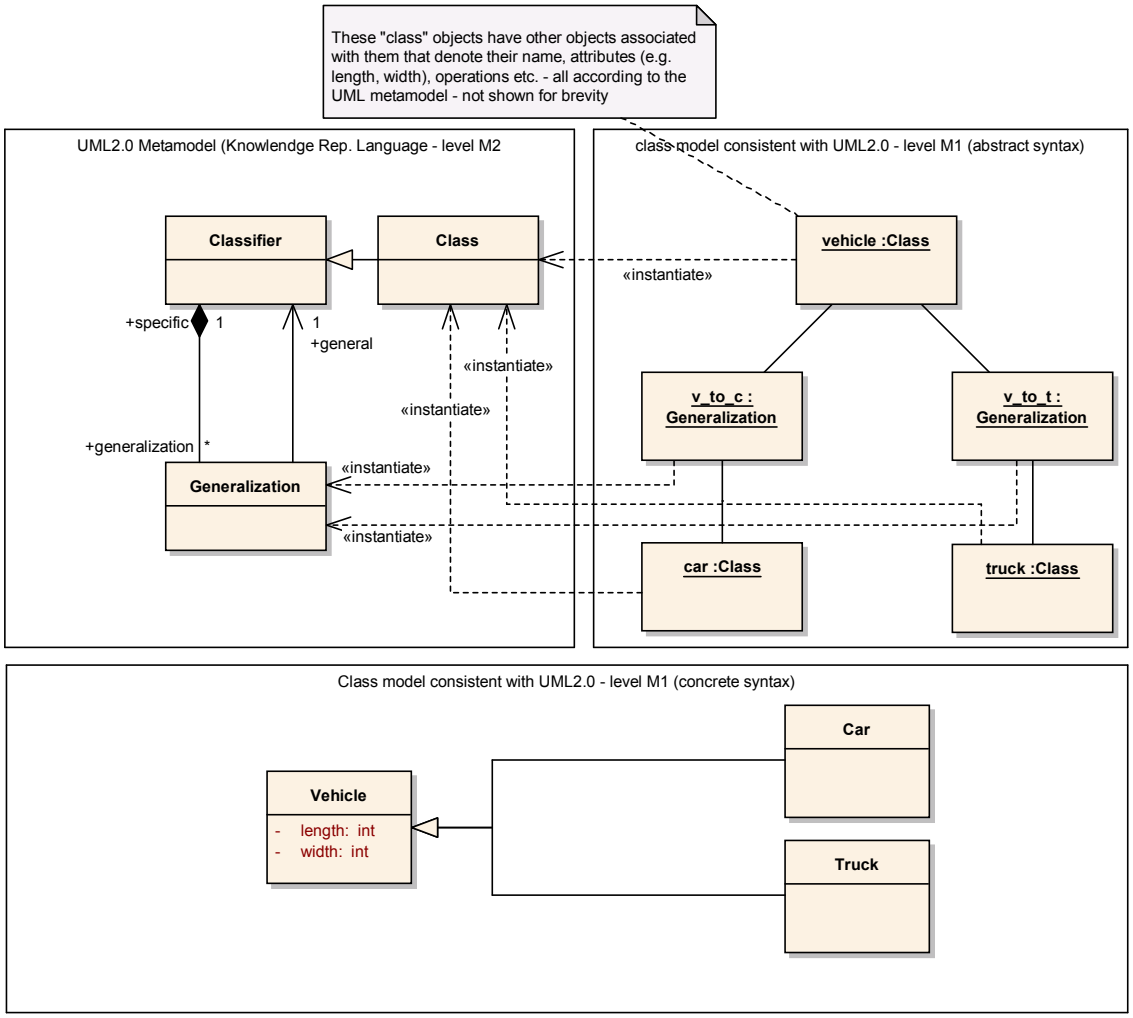


Figure 1.1: UML meta-modelling example

Meta-models and models define two levels of meta-modelling. In fact we can have four levels: M0 - model instance level, M1 - model level, M2 - meta-model level, M3 - meta-meta-model level. The model instance level contains all the objects (real time instances or real world objects) of classifiers (classes) included in the model level. The meta-meta-model level defines the language to represent a meta-model (a meta-modelling language). In defining the whole Requirements Specification Language we use MOF[1] [Obj03a] as a meta-modelling language. From the perspective of MOF, UML and RSL both can be viewed as user models based on MOF as language specification. From the perspective of RSL, requirements specification is a model of requirements expressed by the RSL.

[1]MOF is similar to UML but it is reduced to simplified class diagrams with embedded OCL [Obj03b] constraint expressions (expressions in curly brackets "{}"). These special class diagrams have their semantics defined for language construction.

The most common role of a meta-model is to define the semantics for how language tokens from a language specification can be used. As an example, consider figure 1.1, where the meta-classes Classifier, Generalisation and Class are defined as part of the UML meta-model. These are instantiated in a user model in such a way that the classes Car, Truck and Vehicle are all instances of the meta-class Class. The generalisation between Car and Vehicle (or Truck and Vehicle) classes is an instance of the Generalisation meta-class (based on [Obj05a]).

1.2.2 Defining languages using meta-modelling

In languages defined with MOF we define tokens of a language, their relationships and meaning. Every token has to be described in terms of its syntax (abstract, concrete) and semantics.

The abstract syntax defines the tokens of the language and their relationships and integrity constraints available in the language. Relationships and constraints determine a set of correct sentences that can be created in the language (its grammar). Note that abstract syntax should be independent from graphical or textual representation of the language elements it is defining. In the RSL, abstract syntax is expressed by MOF diagrams and natural language descriptions.

The concrete syntax is a description of specific notation used in representing a language's elements. In other words it is a mapping from notation to the abstract syntax. If an element of the meta-model is marked as abstract it does not have any concrete syntax (because it cannot be instantiated). In the RSL, its concrete syntax is expressed by natural language descriptions and illustrated with examples of the language's usage. The figure 1.2 is an example of a definition of the abstract and concrete syntax for the RSL.

The semantics of meta-model elements expresses the meaning of properly formulated constructs of a language (according to its abstract syntax). In the RSL, its semantics are represented by natural language descriptions.

The RSL is not an extension to UML, though we use certain UML packages (for those parts of the RSL that derive from UML). Those packages were merged into the RSL definition (using «merge» or «include») from "UML Superstructure" [Obj05b] packages. This merger is done on the package level. Inside a package that is part of the RSL's definition, meta-classes from the merged UML package are directly specialised.

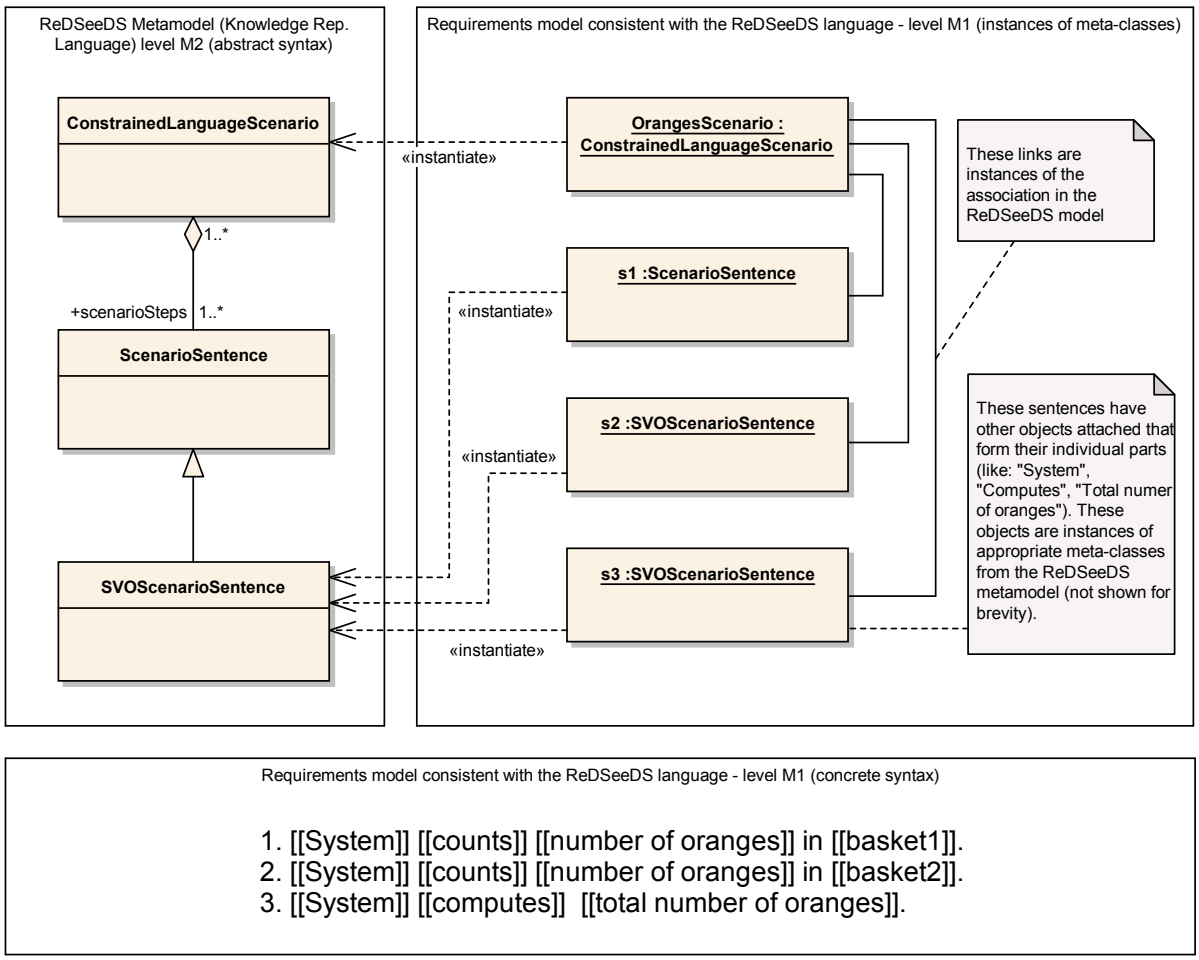


Figure 1.2: ReDSeeDS meta-modelling example

1.2.3 Relations to UML and SysML

As described in the previous section the RSL as a modelling language is not an exact extension to UML or indeed a replacement. However UML is utilised under specialised circumstances where it is only individually unique parts (sub-packages, meta-classes, etc) of the RSL which require the reuse of the already defined UML packages. This reusability feature but non-extension of the UML concept is facilitated through the utilisation of «merge» or «include» from “UML Superstructure” [Obj05b] packages. The merger activity is conducted at the RSL package level. Since the RSL is not deriving any previously defined OMG UML packages but is instead merging parts of UML that are deemed of necessary use to the RSL’s formation then it is safe to point out that the ReDSeeDS RSL is not extending UML in any way. The RSL under the context of software case and requirements reusability is in effect a highly constrained specialisation of UML which falls out of the bounds of being a subset of or an extension to UML itself but UML definitions and theories are incorporated into the RSL’s overall design .

In contrast, SysML [Obj06b] as a domain specific modelling language was purposefully created as an open source project to aid in supporting the specification, analysis, design, verification and validation of an ever broadening range of systems and systems of systems which are not software centric or indeed software oriented in any way. See http://www.sysml.org/. SysML is defined to be an extension to a subset of UML using the UML profiling mechanism. This allows for robust modelling to be conducted within the generalised field of systems engineering. As such SysML is an enhancement to UML which takes away some of UML's software centric specialisation features and replaces them with generality or generic features special to the entire engineering field as a whole.

One extension of SysML of potential interest for our RSL is a part explicitly dedicated to requirements in textual form. We even considered to base RSL on this part of SysML. However, what is defined there only covers a minor part of what we can express for requirements. In particular, there is no classification of requirements, and they can be related only in predefined ways. So, the reuse effect would have been very small.

While it was intended by the developers of SysML to keep this part small and to rather have it extended, such extensions would have been very laborious, if not also very hard to do for everything we had in mind. In particular, the textual representation of a requirement is inherently mixed up with the corresponding requirements entity itself. In contrast, we strive for a clean distinction between requirements and their representation.

Considering the view of SysML as now defined it is plausible to say that the ReDSeeDS RSL is intended to be similar to SysML's portrayal in that the RSL is effectively to become the SysML for software centric requirements reuse but without the need to extend UML or any subsets of UML.

1.2.4 Structure of the language reference

Part II of this document contains the RSL definition. It has been divided into sections according to the logical structure of packages and subpackages of the RSL.

The RSL is divided into five main packages:

- Kernel (abstract elements of the language forming the common basis for other elements)
- Requirements (requirements as such with their relationships)

- Requirements Representations (definitions of individual requirements in various notations)
- Requirement Representation Sentences (basic "building blocks", i.e. sentences that form the representations as above)
- Domain elements (elements that allow for forming a vocabulary of phrases based on common terminology)

Each of theses packages is described briefly with an overview section (including a package diagram), which is followed by description of its subpackages. Every subpackage is presented in an overview explaining general ideas behind a package, a meta-model diagram for this package and two sections which describe the abstract syntax with semantics of language constructs and the concrete syntax.

1.2.5 Notation conventions

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 emphasized 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).

1.3 Related work and relations to other documents

External research work conducted in compliance with formulating a good understanding of the Requirements Specification Language included researching and reasoning about such ar-

eas as software case representations, query procedure pragmatics, Domain representation, Domain mapping with or without hyperlinking, Domain access methods via taxonomies and similarity measures concerning domain constructs (vocabulary items) and requirement dependencies/interdependencies leading to possible upgrades of the domain or industry specific terminology including terms.

From the domain elements part of RSL, a means for modelling domain entities are introduced, such as domain entity types, vocabulary, phrases, and terms. Terms are organised in a terminology. Domain entities are used for representing requirements of *each* new software development project forming a requirement model of the developed software case. For performing case retrieval on the basis of similarity measures, this requirement model is mapped and included in a so-called *software knowledge model*. This software knowledge model contains the knowledge known for *all* software cases of an organisation i.e. the software vendor that implements the different software cases.

For reuse purposes, we strive in this project for finding software cases based on similarity measures. In principle, this can be done by text-based approaches, where our terminology as defined in the language will be very useful. For including more semantics into similarity measures, an *ontology* of the given application domain should be available. While our requirements language does not yet include any means for knowledge representation and reasoning, the user may still use it to represent a domain model using object-oriented means (in the form of a domain element diagram, as derived from a UML class diagram). Such models can be used as a simple form of ontology. These issues will be worked out in Workpackages 3 and 4 of ReDSeeDS and discussed in the related deliverables.

One form of representing requirements in ReDSeeDS is to write them down in constrained language. This constrained language uses the so-called *SVO(O)-Grammar*, recently researched at WUT [SBNS05b, SBNS05a]. Other significant work concerning some kind of restriction to natural language was and is still done in the *Attempto* research project conducted by the Department of Informatics and the Institute of Computational Linguistics at the University of Zurich [FHK+05].

The *Attempto Controlled English (ACE)* language developed in the course of the *Attempto* project is currently available in its fifth version. Although ACE closely resembles natural English, the syntax of a text written in ACE is based on a defined abstract grammar which avoids ambiguity in language constructs [Hoe04][2]. Furthermore, ACE can be automatically translated into first-order logic and consequently be read by humans as well as by machines.

[2]The cited article contains abstract grammar for ACE 4.0. The grammar for version 5.0 has not been published yet.

The results of search engines in the internet for "user interface" is a long list including conferences, workshops, symposia etc. This incudes conferences like the International Conference on Intelligent User Interfaces (IUI), Computer Human Interaction (CHI), International Conference on HumanComputer Interaction (INTERACT), The International Conference on Ubiquitous Computing (UbiComp) etc. They all show a number of on going research activities in this field. A short analysis of these activities will show that most research is concerned with the design of user interfaces as one stage in the software development process, and not with the elicitation of user interface requirements in the analysis phase. Consequently, re-use of user interface requirements becomes impossible. Nevertheless significant research work in the field of user interface development that serve as a basis for the RSL has been done.

1.3.1 Model Based User Interface Development

Following the Model Based User Interface Development (MB-UID), the user interface is specified through different declarative models at different abstraction levels (see [Sze96])

- at the highest abstraction level is the application model. This defines the objects in the domain as well as the user tasks. Some of these objects can be manipulated by users on the user interface.
- at the following abstraction level is the abstract user interface specification. Abstract interaction objects (AIO) are defined at this level. they include groups of objects for information representation as well as for the user interaction and the dialog.
- at the third level the concrete user interface is defined by using concrete interaction objects (CIO). These are concrete objects, e.g., button, window, checkbox, text field etc., from the object library of a selected toolkit.

Other models include the user model and the platform model (see Figure 1.3).

The distribution of user interface aspects over several models allows the separation of concerns, an important requirement especially for complex user interfaces. An extensive overview of MB-UID can be found in [Mol04].

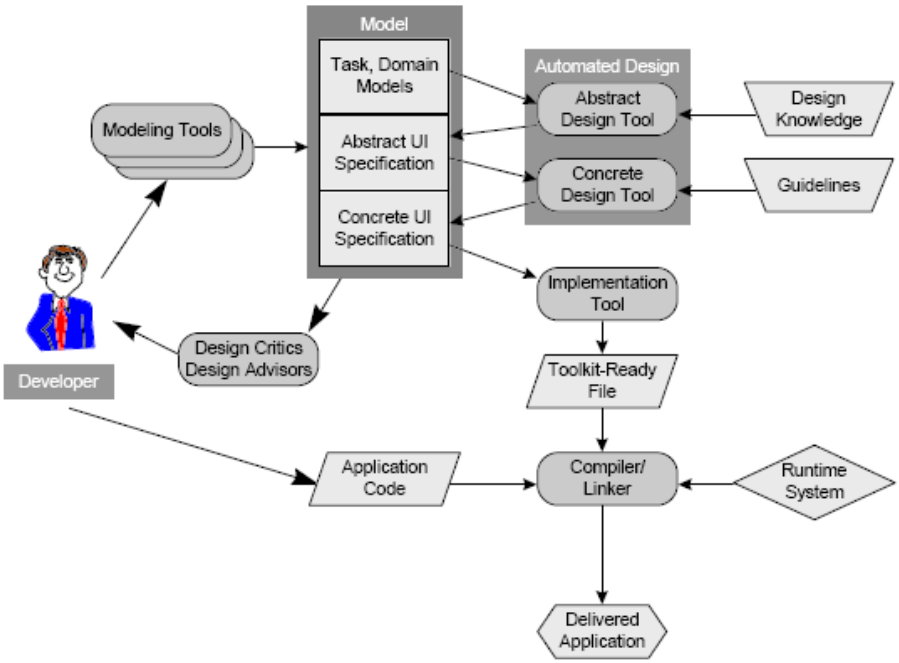


Figure 1.3: The MB-UID Architecture

1.3.2 User Interface Description Languages

Since the existence of XML as a universal data format, several languages for the description of user interfaces models (UIDL) for predefined systems have been developed. They can be classified according to

1. The abstraction levels of their elements with respect to the elements of the implementation toolkit,
2. Their application domains and
3. The abstraction levels of the resulting models with respect to modality.

The first category includes UIDLs with abstract elements like UIML [AH02] and those that provide concrete elements like XUL [Moz02].
In the second category, there are general purpose UIDLs, e.g., UIML [AH02], and those for specific application domains, e.g., XUL [Moz02] for web applications.
The third category includes UIDLs following on the approach of first defining the abstract user interface with modality independent elements and then transforming the resulting description into a concrete user interface by providing modality dependent elements, e.g., UsIXML [Lim04], or integrating modality specific elements of different modalities in one model, e.g,

useGUI [Muk06]. An extensive survey of most UIDLs can be found in [Lim04].
As mentioned previously, the goal of these UDLs is to support the design phase of the software development process. In the contrary, the UI descriptive part of the RSL should support the analysis phase.

1.3.3 Task and Object Oriented Requirement Engineering

The success of both approaches above depends on the proper elicitation of user requirements. The requirements analysts should know which data to collect and which decision to take. The Task and Object Oriented Requirements Engineering Framework (TORE) [PK03] gives an answer. It defines decision points to be made at different (abstraction) levels (see Figure 1.4). The levels conform to a certain kind of pattern: At the domain level, the interaction level, the application core and the GUI, there are always decisions concerning behaviour chunks like activities, functions or actions as well as decisions concerning data. Interaction and dialog put these chunks into a sequence. UI-structure, architecture and screen structure group data and behaviour chunks together. In this way, it implicitly guides the analyst towards the required data. The specified decision points in TORE can be defined as follows. More details can be found in [PK03].

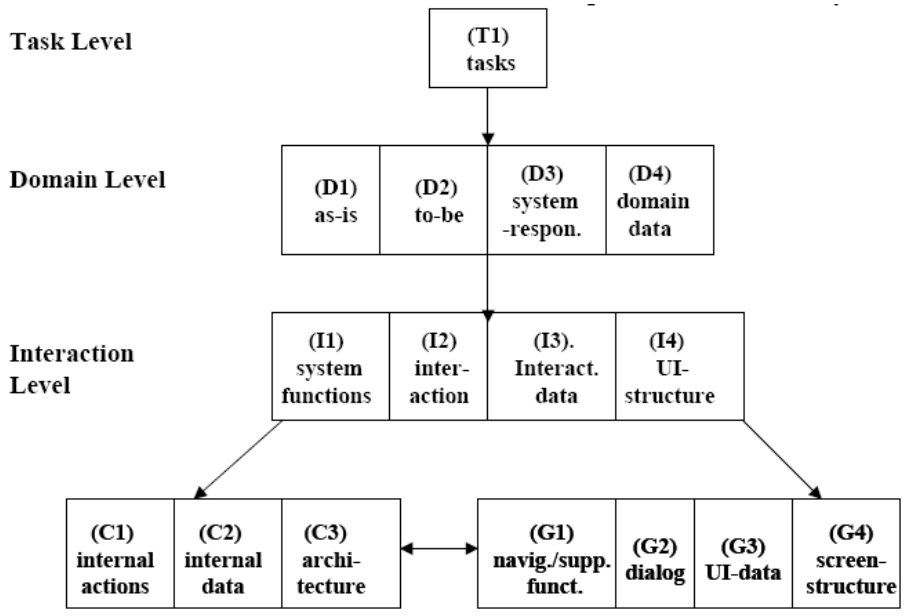


Figure 1.4: The TORE Framework

- (T1) Decisions about the user tasks:
  User roles and the tasks of these roles to be supported by the system are determined.

- (D1) Decisions about the as-is activities:
  Decision on which activities users currently perform and whether these are relevant for the system. The activities are derived from the tasks. Hence user tasks are being refined here.

- (D2) Decisions about the to-be activities:
  Decision on new tasks as the improvement of the as-is activities.

- (D3) Decisions about the system responsibilities:
  Which activities should be done automatically by the system and which should be left to the user?

- (D4) Decisions about the domain data relevant for a task:
  System responsibilities of UIS manipulate data. Decisions have to be made on which domain data are relevant for the system responsibilities.

- (I1) Decisions about the system functions:
  System responsibilities are realised by system functions. The decision about the system functions determines the border between user and system.

- (I2) Decisions about user-system interaction:
  It has to be decided how the user can use the system function to achieve the system responsibilities. This determines the interaction between user and system.

- (I3) Decisions about interaction data:
  For each system function the input data provided by the user as well as the output data provided by the system have to be defined.

- (I4) Decisions about the structure of the user interface (UI-structure): Decisions about the grouping of data and system functions in different workspaces have to be made. Through the UI-structure, the rough architecture of the user interface is defined. This structure has a big influence on the usability of the system.

- (C1) Decisions about the application architecture:
  The code realising the system functions is modularised into different components. In the decision about the component architecture, existing components and physical constraints as well as quality constraints such as performance have to be taken into account. Only a preliminary decision concerning the architecture is made during requirements.

- (C2) Decisions about the internal system actions:
  Decisions have to be made regarding the internal system actions that realise the system functions. The system actions define the effects of the system function on the data.

- (C3) Decisions about internal system data:
  The internal system data refines the interaction data to the granularity of the system actions. The decisions about the internal system data reflect all system actions. In OO, system data is grouped within classes.

- (G1) Decisions about navigation and support functions:
  It has to be decided how the user can navigate between different screens during the execution of system functions. This determines the navigation functions.

- (G2) Decisions about dialog interaction:
  For each interaction the detailed control of the user has to be decided. This determines the dialog. It consists of a sequence of support and navigation function executions. These decisions also have a strong influence on the usability of the system.

- (G3) Decisions about detailed UI-data:
  For each navigation and support function, the input data provided by the user as well as the output data provided by the system have to be defined. These decisions determine the UI-data visible on each screen.

- (G4) Decisions about screen structure:
  The separation of workspaces as defined in (I4) into different screens that support the detailed dialog interaction as described in (G2) has to be decided. The screen structure groups navigation and support functions as well as UI-data. The decisions to separate the workspaces into different screens are influenced by the platform of the system.

1.4 Structure of this Document

Part I gives a conceptual overview of the behavioural, structural and user interface parts of the Requirements Specification Language. The behavioural part introduces the requirements language and the representation language and deals with functional and behavioural requirements and presents their conceptual model. Then outlined are possible representations of the requirements without going too deeply into detail. Newly introduced concepts affecting the behavioural aspects are also discussed. The structural part of RSL gives a conceptual overview of domain entities and dictionaries. It describes different types of entities existing in a domain and the conceptual model of the domain's vocabulary, respectively. the user interface part communicates the purpose of modelling user interfaces within requirements specifications, contains a rationale and outlines the approach taken in development of this part of the language. Also described are the basics of modelling elements and behaviour within the user interface environment.

Part II initially defines the metamodel of the RSL's behavioural part, again dealing with the subject of requirements itself and different possibilities for requirement representation. It is divided into four chapters, each of them dealing with a part of the metamodel. Every chapter has a short overview, defines abstract syntax and semantics and then gives a short example of the concrete syntax. It defines the part of the metamodel containing the requirements themselves and their arrangement. It explains the part of the metamodel that deals with different kinds of representations, especially textual representations and schematic representations, which can be displayed as UML-like models. It then defines the grammar for the semi-formal textual representation.

Furthermore part II, after discussing the benefits and consequences of using a domain vocabulary, defines the domain elements (structural) part of the language. This major package contains several subpackages. These subpackages cover basic domain entities, the actors in the system's environment and the representations of the system and the entities. Furthermore, this package contains phrases and other, more fine-grained elements which compose phrases. Phrases constitute the names of the entities as well as the parts of a sentence in constrained language. Finally, the package comprises the individual terms which can occur in a phrase. Each section concerning one of the packages has an overview, defines abstract syntax and semantics, and then gives a short explanation of the concrete syntax using the Fitness Club case study as a running example.

Finally, partII specifies the meta-model for the user interface aspects of the Requirements Specification Language. Here, there are addressed the structural and behavioural aspects of UI specification on the requirements level.

1.5 Usage guidelines

The ReDSeeDS Requirements Specification Language (RSL) definition should be used as a book that guides the reader through the structure, syntax and semantics definitions of the RSL, as part of the complete ReDSeeDS Software Case Specification Language (to be defined in Workpackage 3). It should be used mainly by creators of appropriate software CASE tools (with reusability features) that would allow handling of the language by the end users (analysts, etc.) to express behaviour of the system under development. It can be used by advanced end users of the language as a reference for the language's syntax and semantics. Examples of RSL elements' concrete syntax have illustrative character and should be treated only as support in understanding of a given element's occurrence.

Users of the RSL Specification are expected to know the basics of metamodelling and MOF (Meta Object Facility) specification [Obj06a]. Knowledge of UML ([Obj05b] and [Obj05a]) could be helpful as some elements of RSL are extensions, constraints or redefinitions of UML elements.

Part I

Conceptual Overview of the Coherent Requirements Language

Chapter 2

Introduction

The ReDSeeDS Requirements Specification Language (meta-)model consists of 3 parts, which are linked to an extra model of the Reuse Domain:

1. Requirements Language
2. Requirements Representation Language
3. Application Domain Language

The primary reason for separation of the overall language model into several parts is the separation of concerns. In particular, the main separation is between Requirements Language and Requirements Representation Language. This is a crucial innovation, which is important since we are not reusing requirements themselves but rather requirements representations.

The separation of Requirements Language and Requirements Representation Language allows separation and simplification of the Reuse Domain. Avoiding the separation between the former two would force integration of the latter and mixing of the different concerns and lead to higher complexity of the overall language specification. In addition, requirements are not reusable directly, and this fact should be reflected in the language specification.

It is also important that the representation of the Application Domain is distinct from the Requirements Representation, while they will be linked, of course. The requirements may not be understood without the links to the Application Domain, but the content of the application domain is not requirements.

Hyperlinks between Application Domain and Requirement Representation are a crucial element of our language, and greatly facilitate keeping the specification coherent.

Chapter 3

Requirements for the requirements language

Before actually designing our requirements language RSL as a joint effort of several partners, it was deemed useful to gather requirements for RSL itself. Unfortunately, RSL has obviously not yet been available for specifying its own requirements. So, we simply used natural language for their representation. Still, we used the keywords *shall* and *should* (in *italics*) for clearly distinguishing mandatory from optional requirements. We also separated functional from constraint requirements. The sections below list the results of this informal requirements acquisition.

3.1 Functional Requirements

- RSL *shall* allow describing any kind of requirement in (free) natural language text.
- RSL *shall* allow assigning a unique identifier to each requirement, which becomes a requirement entity in this way.
- RSL *shall* allow specifying whether a requirement is mandatory or optional.
- RSL *shall* allow including attributes to each requirement entity.
- RSL *shall* allow to formulate scenarios in textual form according to a given grammar.
- RSL *shall* allow to formulate use cases and to link scenarios to them.
- RSL *shall* allow to explicitly link a scenario with functional requirements for the system to be built in such a way, as to specify which functions will have to be available to execute this scenario.

- RSL *shall* allow defining a glossary of terms in natural language, which serves as a defined vocabulary for formulating the requirements.
- RSL *shall* allow to model requirements using a selection of UML 2.0 diagrams.
- RSL *shall* allow building hierarchical structures of the requirements entities.
- RSL *shall* allow cross-linking of the requirements entities.
- RSL *should* allow linking each entry of the glossary of terms with a corresponding class in UML diagrams.
- RSL *shall* allow defining a user interface of the system to be built.
- RSL *should* allow specifying a task model.
- RSL *should* allow linking to any kind of descriptions or models in any other language.
- RSL *should* allow defining stakeholders.

3.2 Constraint Requirements

- RSL *shall* have a mandatory core part that is being used for finding similar cases.
- RSL *shall* have optional parts that are not being used for finding similar cases.
- RSL *shall* be defined based on the UML 2.0 metamodel, both through restrictions and extensions.
- RSL *shall* have compatible representations of requirements and user interface.
- RSL documents *shall* be semi-formal in the sense of including both formal and informal representation.
- RSL *should* be extensible.

Chapter 4

Requirements Model

Requirements engineering (RE) is the essential activity in assuring that one builds computer-based systems that will satisfy stakeholders' goals. As the need for a systematic method of requirements elicitation and specification first became obvious for very large and complex systems, most RE research focused on discovery and development of requirements techniques and artefacts that are tailored to support the development of these very large systems in those environments with relatively large amounts of resources. Developers of a small system, on the other hand, traditionally used an *ad hoc* approach to RE due to the system's small size and the developers' unsystematic approach to development. The importance of systematic RE increases dramatically, even for small systems, with the introduction of product-line approaches, customisable software, etc. So, over time, we have gone from *ad hoc* approaches to requirements management to more formal ways of capturing and managing requirements. This trend is what led to our project and the goal of building systems based on requirements reuse. That is, if we are systematically capturing and managing requirements for one project then we should be able to reuse some of these requirements on other similar projects.

The other important impact on RE techniques is due to the nature of the development of large systems. Traditionally, a typical Computer-Based System (CBS) is developed in-house, where developers work with relatively stable domains, are responsible for the development of the system from scratch, and have relatively stable production teams and a large amount of resources. This way of development has led to the dominance of the requirements specification that focuses mainly on product-level requirements such as features [Lau02] and subsequently on low-level requirements and design. Designs of different systems have already been successfully reused either through reuse such as using design pattern or through larger scale reuse such as using reference architectures for the specification of the new systems. The reuse of these design elements implied the reuse of some of the background requirements that led to these designs, but there

was no intentional and systematic reuse of the requirements as such. The main contribution of this project will be in pushing the reuse effort even further, i.e., beyond design reuse — all the way to systematic requirement reuse. The primary target, for facilitating reuse are product-level and low-level requirements.

Product-level and low-level requirements are very well studied and widely applied in industrial settings, but the main difficulty is in ensuring that they fulfil essential business goals. Product-level requirements form a set of features that, combined, are used to achieve the organisation's business goals. The success of the overall goal depends on every single one of the features and on the particularities of their interactions. The problem of achieving goals is exacerbated as a result of their frequent changes over time, which cause a chain reaction of changes in product-level and low-level requirements.

Lauesen has observed that product-level and low-level requirements management is straightforward and changes to them are relatively easy to deal with in practice [Lau02]. Developers can usually sense when these requirements are not correct and do not fit with each other. This ability usually does not work at the higher levels of abstractions, and it is the responsibility of the business analyst to deal with these higher level requirements. In an occasional case, it is not even possible to estimate how changes in the product-level requirements effect overall goals until the changes are implemented [Lau02]. Therefore, besides attempting to reuse product-level and low-level requirements, our project is also dealing with the reuse of higher-level requirements in order to ensure the traceability and the fulfilment of all the requirements at the later design stages for the system that is built through the requirements reuse.

Requirements are specified either directly or indirectly for many different purposes and as part of many different engineering activities. One example classification is [Lau02]:

1. *Business-level requirements specification* — Business-level requirements are most often specified indirectly as part of business re-engineering activities. The most common concepts that appear at this level are business goals, processes, resources, and rules. For example, for an elevator system, a business-level requirement is: "The elevator shall transport passengers and goods from any floor to any other floor."

2. *Domain-level requirements specification* — Domain-level requirements, as mentioned previously, are most often indirectly specified in the traditional requirements specifications [DvLF93]. Newer, more systematic versions of domain-level RE have received a lot of attention recently [BPG+01, CKM01, MCL+01]. Most explicit domain-level requirements are captured and specified for domains, which are becoming increasingly complex and difficult to adequately support by systems [CKM02, GMP01, GPS01, MC00]. The

most common concepts that appear at this specification level are user goals, user tasks, domain input, and domain output. A more recent trend is the incorporation of agent-based analysis as part of domain modelling [MKG02, KGM02, MKC01, GPM+01]. For an elevator system, an example domain-level requirement is: "The elevator shall be accessible from each floor."

3. *Product-level requirements specification* — Product-level requirement specifications are the most common type of requirement specifications. There is an extensive body of knowledge about them, and most previous research focused on improving the different techniques used to elicit, specify, and validate this type of requirement. The common artefacts and concepts that occur as parts of product-level specifications are features, use cases, functional lists, data input, data output, etc. For an elevator system, an example product-level requirement is: "The elevator shall accept elevator calls only while stationary."

4. *Design-level requirements specification* — Design-level requirements specification are the requirements that directly constrain the design of a system. Much effort has been invested into its standardisation through the Unified Modelling Language (UML) [Lar04, Fow04]. UML artefacts and underlying techniques present the most common types of concepts and techniques used to capture requirements at this level. This level acts as a transition phase between product-level specification and code-level requirement specifications. For an elevator system, an example design-level requirement is: "A queue data structure shall be used to store the data for elevator calls."

5. *Code-level requirements specification* — Code-level requirements are usually specified as part of the programming activity and describe details of low-level algorithm and data structures. This type of specification is that with which most programmers are familiar, as it is inseparable from coding. Code-level requirements focus mainly on implementation-related issues and constraints and are probably the best understood form of requirements specification. For an elevator system, an example code-level requirement is: "Due to the timing constraints, function calls to retrieve elevator call data shall be implemented in the C programming language rather than in the Python programming language."

This work and our requirements language is applicable and can be used at all these requirement and requirement specification levels in order to ensure full reuse and proper development of the new system.

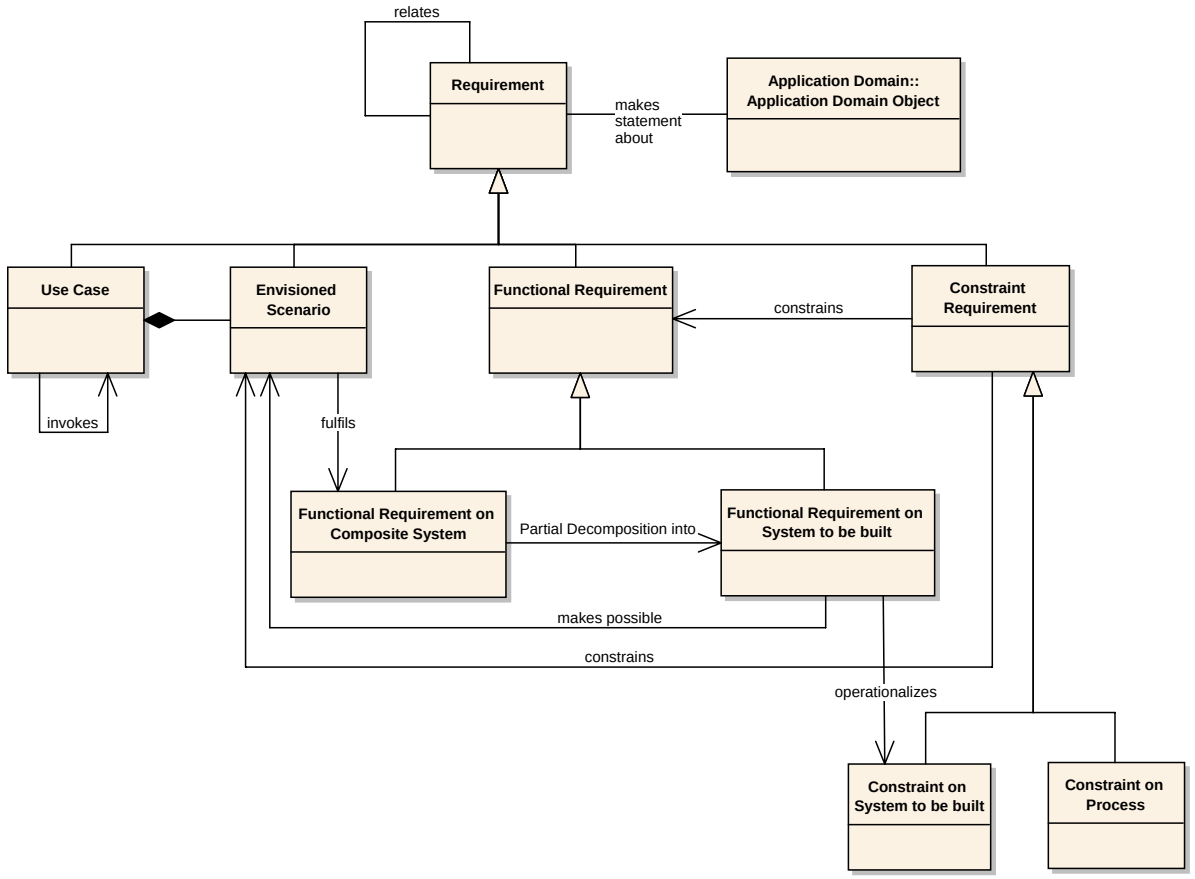


Figure 4.1: Conceptual Requirements Model

4.1 Requirements Model Overview

Figure 4.1 shows our conceptual Requirements Model, i.e., how we conceptualise the domain of requirements. This conceptualisation is influenced by decades of research and practice of requirements engineering and especially [Kai97, Kai00, EK02, Kai05]. This conceptual model shows what models of concrete requirements should look like in our applications. This is already in the spirit of a metamodel, but the formal metamodel of our requirements specification language is given below.

The main entity in the Requirements Language is Requirement. Requirement can be decomposed into a number of other requirements or aggregated to composites, thus the granularity is flexible. There are four specialisations of requirements: Use Case, Envisioned Scenario, Functional Requirement, and Constraint Requirement:

- A Use Case consists of a number of Envisioned Scenarios that belong together in terms of use. E.g., the Use Case for getting cash money has several scenarios of how this is actually envisioned to be achieved.
- Envisioned Scenarios are the means of achieving high-level functions given as Functional Requirements on Composite System (e.g., Cash Withdrawal). The composite system includes the system to be built (e.g., an ATM) and possibly other systems (e.g., a bank system), including human users (e.g., bank customers).
- Functional Requirements are the generalisation of Functional Requirements on Composite System and Functional Requirement on System to be built. The latter are functions needed (e.g., Customer Identification or Cash Provision) that will make possible the enactment of Envisioned Scenarios once available. In effect, Functional Requirements on Composite System are partially decomposed into Functional Requirements on System to be built.
- Functional Requirements are tightly related to Constraint Requirements. Constraint Requirements often constrain Functional Requirements. E.g., only solutions for Cash Provision are acceptable (in the overall solution space) that are secure and reliable. There are two specialisations of Constraint Requirements: Constraint on Process and Constraint on System to be built. The former involve, for instance, development method or tools, the latter, for instance, security or reliability. Some Functional Requirements on System to be built operationalize Constraint Requirements on System to be built. E.g., (required) functions for accepting a password, etc. operationalize a security requirement.

4.2 Requirements Model Details

Requirement

IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology defines requirement as:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a condition or capability as in (1) or (2).

As discussed previously, a requirement can exist at multiple abstraction levels. It is common to decompose higher-level requirements to lower-level requirements forming some kind of requirement decomposition tree. Each requirement is typically related to a number of other requirements. In addition, each requirement can be represented in multiple views. As such, we have a number of dimensions that constrain and make it challenging to capture a requirement properly with all its relationships.

This difficulty was obvious even during our work. It was hard to boil down and to keep in mind the clear definition of what a requirement is. This was particularly difficult when discussing different types of requirements. Any deviation from the common definition resulted in a ripple effect of conflicts with other terms and definitions of other language constructs. This is why we insisted on strictly following and not modifying the standard definition of what a requirement is in the early stages of our work. Nevertheless, at the end we had to adapt and limit this definition as discussed in the Discussion section.

Use Case

The official definition for use cases in our project is:

> "A collection of possible scenarios between the system under discussion and external actors, characterised by the goal the primary actor has toward the system's declared responsibilities, showing how the primary actor's goal might be delivered or might fail." [Coc97]

Use cases are related using *invoke* relationship. This relationship unifies and overrides the standard UML use case relationships *extends* and *includes*.

Envisioned Scenario

The definition for a scenario in our project is:

> "A sequence of interactions happening under certain conditions, to achieve the primary actor's goal, and having a particular result with respect to that goal. The

> interactions start from the triggering action and continue until the goal is delivered or abandoned, and the system completes whatever responsibilities it has with respect to the interaction." [Coc97]

Of course, not every scenario may be considered a requirement. In particular, an as-is scenario is already performed and will not need new functionality from the system to be built. In contrast, an "envisioned" scenario is not yet (fully) supported by software but will have to be executable after deployment of the system to be built. In this sense, an envisioned scenario can be viewed as a requirement.

Note, however, that the system to be built alone will also not be able to execute it alone. It will need the interaction with some actor in its environment. So, it is a requirement on the Composite System, which will have to be able to execute it. In the course of its execution, the currently envisioned scenario will then fulfil a Functional Requirement on Composite System.

Again, the same as with use cases, our language supports writing scenarios using the language of different levels of formality; from informal natural language to constrained versions such as SVO(O). Of course, a more formal representation is to be preferred for any kind of processing by machine, in our case for finding similar cases facilitating reuse.

Functional Requirement

Taking into account the previously mentioned definition of a requirement, one can define a functional requirement as:

1. A capability needed by a user to solve a problem or achieve an objective.
2. A capability that must be met by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a capability as in (1) or (2).

Compared to the previous definition, the definition of a functional requirement limits the requirements definition to the usage of the term capability.

Functional requirements typically represent the majority of the requirements. A major issue to resolve with any functional requirement is who is responsible for fulfilling it, i.e., what is

the system that is performing the activity or activities that will satisfy the respective functional requirement.

This issue often comes up when a functional requirement is discussed in relationship with a use case or scenario. Is a use case or scenario a functional requirement? Since fulfilment of a use case typically involves activities performed by both actors and the system to be built, we cannot say that a use case is a functional requirement on the system to be built. On the other hand, if we take into consideration that actors together with a system to be built from another, composite, system, one can claim that use cases and scenarios are requirements on that composite system. That is, the name of the use case or scenario can be thought of as a functional requirement on a composite system, and the actual steps that the system to be built has to perform in order to satisfy the functional requirement on the composite system can be thought of either as functional requirements on the system to be built or as activities needed to fulfil those requirements, depending on the perspective taken.

Therefore, in our model we have two different kinds of functional requirements: functional requirement on composite system and functional requirement on system to be built.

Functional Requirement on Composite System

Functional Requirement on Composite System is a functional requirement that is supposed to be fulfilled by the system composed of system to be built and its actors. This type of requirement is primarily fulfilled through the envisioned scenarios.

Functional Requirement on System To Be Built

Functional Requirement on System To Be Built is a functional requirement that is supposed to be fulfilled by the system that is being specified. This type of requirement primarily appears as the steps in envisioned scenarios, i.e., makes envisioned scenarios possible.

Constraint Requirement

In the requirements engineering community, it is common practice to distinguish between functional requirements and constraint requirements, which are also called *non-functional* requirements in literature (see, for example [Gli05]). While functional requirements refer to tasks a

system must or may perform, constraint requirements describe mandatory or optional properties of a system. These may relate to e.g. security, reliability and performance conditions or even to political, legal and economical aspects [CdPL04].

Examples for constraint requirements are:

- The system will be proved to have a E4 level in ITSEC standard.
- The system must store and process customer data conforming to current legislation.
- The cost for system development may not be higher than three times the fitness club's monthly net profit.

Certain constraint requirements on system to be built may be operationalized by functional requirements on system to be built. For example, the constraint requirement

> The system must store and process customer data conforming to current legislation.

could be operationalized by

> The encryption of customer data is done using a public key method with 1024 bits of key length.

Similar to functional requirements, taking into account the IEEE definition of a requirement, one can define a constraint requirement as:

1. A condition needed by a user to solve a problem or achieve an objective.
2. A condition that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a condition as in (1) or (2).

Compared to the IEEE definition, the definition of a constraint requirement limits the requirements definition to the usage of the term condition.

Traditionally, constraint requirements are referred to as *non-functional requirement* or as *quality attributes*. In our language they are only a part of *non-functional requirements*. Taking into account that non-functional requirements are a set of all requirements excluding functional requirements, we can see that our set of non-functional requirements consists of *use cases*, *envisioned scenarios*, and *constraint requirements*. Moreover, we completely avoid using the term non-functional requirements since this term may be misinterpreted as representing a type of a requirement that has no relationship to functional requirement, or even a requirement that is not functional, i.e., that does not work.

We recognise two types of constraint requirements: *constraints on process* and *constraints on the system to be built*.

Constraint on Process

Constraint on process is a type of constraint requirement that constrains different development process related decisions. For example, the system shall be implemented in Java programming language. These requirements do not constrain the system's functionality per se.

Constraint on System to be built

Constraint on system to be built is a type of constraint requirement that constraints the functionality of the system to be built. For example, the response time has to be no more than 2 seconds.

4.3 Why No Goals?

Goal-driven requirements engineering is an important area of requirements engineering. In particular, several researchers proposed goal-driven RE [DvLF93, BI96, Kai95, MCY99, Kai00, vLL00] as a promising technique for dealing with domain-level requirements for large systems. Goal-driven RE focuses on ensuring that software actually fulfils business needs and requirements. This focus has been achieved by shifting from considering *what* a system should do to considering *why* the CBS should provide particular functionality. In other words, *requirements rationale* is the main focus.

Although most of the original goal-driven RE techniques concern domain-level requirements, for example, through analysis of *personal* and *system* goals, the main idea of goal-driven RE techniques has been to enhance certain traditional requirements techniques such as use cases [Coc00]. Nevertheless, although goal-driven RE techniques are extensively studied, goal-driven RE remains an immature area. This immaturity is apparent from the many different definitions of the word "goal" [DvLF93, Ant96, MCY99, Kai00]. The common pattern to all these definitions is that goals capture the *intention, i.e., objective,* and the *target state* for the entity under analysis and at the entity's own abstraction level. For example, in the case of an elevator system, a goal for the elevator system is to *deliver passengers to the requested floor*. This goal captures the *intention* of *delivering passengers* and also the *target state* of *arriving at the requested floor*. This particular goal captures the rationale for the elevator's responsibility for carrying passengers from a floor to another.

An interesting point to note is that depending on the abstraction level from which one is observing a system and the goal decomposition, a goal may or may not become a functional requirement. For example, for an elevator system, the next level of the goal decomposition might include goals such as *move elevator cab*, *stop elevator cab*, *pick up a passenger*, and so on. Now, if we start working at this abstraction level, the higher-level goal of *delivering passengers to the requested floor* becomes a functional requirement for the lower-level goals such as *moving elevator cab*. An advantage of this goal hierarchy is that it provides traceability when moving from one abstraction level to another and from one goal decomposition level to another.

Overall, one can see that the goals can be ultimately represented or decomposed as regular requirements. In fact, one can even claim that goals are neither necessary nor sufficient for the specification of a system. As such, in our project we have decided not to include them as part of the requirements model. This is not to be interpreted as if goals are not useful for the specification of a system. Quite contrary, but we had to make this decision in the early stages of specifying our language in order to make it:

- Manageable,
- Useful to those who are not using goals as part of their requirements work,

Nevertheless, it will be possible to extend the language and add goals in the future revisions of the language if deemed necessary or desirable.

Chapter 5

Requirements Representation Model

In this section we discuss different requirements specification techniques. Prior to this discussion, it is important to emphasise different aspects of a system to be built from a RE perspective. The four main aspects of each system from the RE perspective are processes, data, architecture, and interfaces. Requirements specification techniques focus on modelling one of these four main aspects. Nevertheless, in many articles in the requirements literature, this division is represented slightly differently.

5.1 Requirements Representation Model Overview

The Requirements Representation Model is presented in Figure 5.1. The main entity in the Requirements Representation Language is Requirement Representation, which is used to represent a Requirement entity from the Requirements Language. The Requirement Representation Language supports representing requirements in two distinct ways:

1. Through the use of Descriptive Requirement Representation, i.e., through Natural Language Requirement Statements and/or Constrained Language Requirement Statements, and
2. Through the use of Model-Based Requirement Representation, in particular UML-Based Requirement Representation.

The mapping from the Requirements Domain to the Requirements Representation Language is not clear cut. For example, a simple Functional Requirement can be captured through the

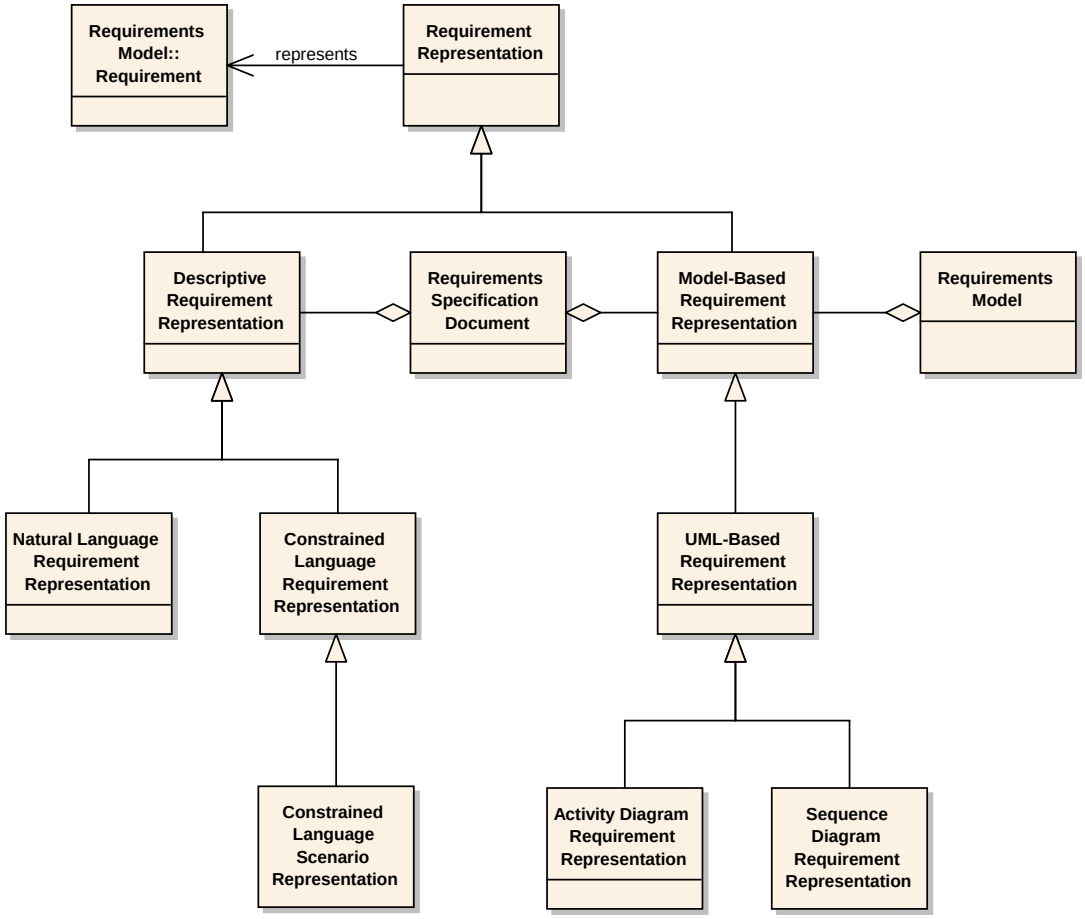


Figure 5.1: Requirements Representation Model

use of a Natural Language Requirement Statement, while a Use Case can be captured through the use of both Constrained Language Requirement Statements and UML-Based Requirement Representations complementing each other.

A self-contained set of Model-Based Requirement Representations makes up a Requirements Model. A Requirements Specification Document contains instances of Descriptive Requirement Representation or Model-Based Requirement Representation.

5.2 Requirements Representation Model Details

Requirement Representation

Requirement Representation is the representation of some requirement using a requirements specification language. We distinguish between Descriptive Requirement Representation and

Model-Based Requirement Representation. A combination of both would normally be part of a Requirements Specification Document.

Descriptive Requirement Representation

Descriptive Requirement Representation is the representation of some requirement as specified using a descriptive specification language, e.g., natural language, SVO(O), etc. The need of the requirement is described in this approach, rather than a model of the system to be built.

Natural Language Requirement Representation

Natural Language Requirement Representation is the representation of some requirement as specified using a natural language, e.g., English, Turkish, Bulgarian, etc. Note, that we technically include also hypertext links into natural-language text, see below. The use of hypertext for representing requirements was already proposed long time ago, see [Kai93, Kai96], but it was not yet defined as precisely as below in a metamodel.

Constrained Language Requirement Representation

Constrained Language Requirement Representation is the representation of some requirement as specified using a controlled/constrained natural language, e.g., SVO(O), Attempto Controlled English (ACE), etc. The point is to take advantage of both the formality introduced by a grammar of a formal language and still the readability of natural language.

Constrained Language Scenario Representation

Constrained Language Scenario Representation is the representation of some envisioned scenario as specified using a constrained language. It gives a precise structure to how such an envisioned scenario is to be written.

Model-Based Requirement Representation

Model-Based Requirement Representation is the representation of some requirement as specified using a modelling language, e.g., UML, etc. In contrast to a Descriptive Requirement Representation, the need of a requirement is specified indirectly. The model specifies what the system to be built should look like, and the requirement is just to build a system like the one modelled.

UML-Based Requirement Representation

UML-Based Requirement Representation is the representation of some requirement as specified in RSL in parts that are based on UML. Using UML for Model-Based Requirement Representation is actually just one of many ways, but it is one appropriate for today´s practice.

Activity Diagram Requirement Representation

Activity Diagram Requirement Representation is the representation of some requirement as specified using UML-Based activity diagrams, as specialised for the requirements specification purposes.

Interaction Diagram Requirement Representation

Interaction Diagram Requirement Representation is the representation of some requirement as specified using UML-Based sequence or communication diagrams, as specialised for the requirements specification purposes.

Chapter 6

Domain entities

The application domain model, referred in the following text as just domain model (DM), consists of three classes: *Application Domain Object*, *System Element*, and *Actor*. The relationships among these three classes of domain elements are depicted in Figure 6.1. Note, that System Element is an element of the composite system and *not* of the system to be built.

These three elements constitute an important support to requirements specifications, i.e., they allow us to create domain specifications. These specifications are clearly linked (through hyperlinks) to the requirements specifications making the overall specification coherent.

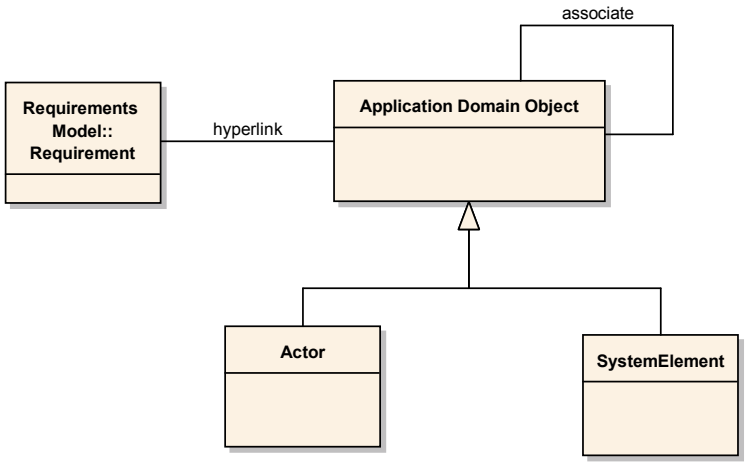


Figure 6.1: Application Domain Model

The main purpose of a DM is to capture the entities that exist in a system's domain. The domain can be seen as consisting of:

- business entities, and

- computer entities, including hardware and software (modelled by SystemElement).

In the sections below we discuss these two main groups. In Chapter 7 we present the domain model with details of representations for individual domain entities.

6.1 Business entities

A software system is part of a larger business system, and serves as a resource to accomplish business goals. To build a useful DM, we need to study and discover different business entities of the domain. The possible sources of business entities are presented below:

1. *Business Resources* — All entities, both physical and abstract, that exist inside the environment of the business are business resources. They include people, information, different systems, and business supplies and products. They participate in the business processes. A subset of these resources is a source of modelling entities for the system to be built. The value of tracking and preserving knowledge about these entities is that these entities are used to perform analysis of the system's architecture, to track changes to the domain and the system from the beginning, and to evaluate how well the system reflects current business needs. For an elevator system, an example business resource is the *cable* used to pull up the elevator cab.

2. *Business Processes* — A system to be built may participate later in several business processes in order to help achieve several business goals. Use cases (UCs) describe subprocesses of larger business processes that are automated by the software system. It is important to understand a business process as it relates its UCs, which in turn relate software requirements that the system has to satisfy. For an elevator system, an example business process is *a passenger's riding of an elevator cab*.

3. *Business Rules* — Business rules are a major source of constraints on a software system. Many constraints directly influence the system's architecture. Therefore, it is important to understand these constraints and to keep track of them, for example, to be able to remove architectural limitations imposed by constraints that do not hold any more. For an elevator system, an example business rule is *the elevator will not change its direction until it services all previously received calls that lie in the current travelling direction*.

The main source of business domain objects are Business Resources, but they are very tightly interlinked with Business Processes and Business Rules. In some cases, they cannot exist sep-

arately. Therefore, we need to seek for domain objects inside business process or business rule descriptions. These are not intended to be described using structural part of RSL, but can be sources to elements expressed in RSL.

6.2 System entities

It is often a case that we are building a new system for which the domain consists of an already existing computer-based system that includes both software and hardware. For such a system, domain entities are not some "natural" objects but rather software and hardware components and other building blocks. Also, taking into account types of systems such as operating systems, compilers, embedded device controllers, shells, GUI libraries, etc. which are all software systems in which all of the domain entities consist purely of what many would call "design components", while in such cases they are all domain entities. For example, a scheduler is a domain entity and Round-Robin algorithm is a scheduling process. For such domains, the main aspects of a system that should be modelled are:

- *System*,
- *subsystems*,
- *modules*,
- *connectors*,
- *processes*, and
- *hardware devices*.

The *System*[1] entity defines the outermost boundary of the system under consideration. The *System* serves as a container for all other entities, and defines the system as a resource in the business system.

A *subsystem* is a part of a *System* or a *subsystem*, being an abstraction of actual physical modules, connectors, and processes. It serves as a container and a building block.

A *module* is a basic architectural building block. For example, in the logical view, it represents a entity that occurs in a domain, and in the implementation view, it represents a code unit.

[1]Note that this "system" is spelled out with initial uppercase letter to distinguish it from the generic "system" used elsewhere.

Modules are abstractions of basic building blocks of the domain, depending on the development technology used.

A *connector* is an abstraction of a communication mechanism or a channel that exists in a system. Its size and complexity vary from a simple procedure call to a connection on the Internet.

A *process* is an executable piece of software. *Processes* are basic building blocks of a run-time architectural view.

A *hardware device* is an entity that occurs in the run-time architectural view, and it represents a physical device that is a part of the system.

The above types of system entities have their place in the domain model created at the requirements level, using the structural part of RSL. The System entity is used throughout the descriptions of functional requirements in general, and in use case descriptions (scenarios) specifically. Other system entities can be used in requirements that specify technical constraints on the prospective system.

Chapter 7

Representation of domains

7.1 Overview

When considering requirements specifications as described in Chapters 4 and 5 we should observe that they do not contain any descriptions of the application (or: problem) domain. In the Requirements Specification Language we clearly want to separate the requirements with their representations from the representations of the problem domain. Thus, none of the requirements representations allows for defining elements of the problem domain. They only allow for defining the system behaviour or quality in their pure form. No interleaving of domain element definitions are allowed.

The main rationale behind this clear distinction is to make requirements specifications unambiguous and consistent. We need to remember that the main purpose of a requirements specification in software engineering is to reflect the real needs of the clients. This specification should be the basis on which developers build a software system of good quality – i.e. a system that meets the clients' expectations to a high extent. Unfortunately, a commonly encountered problem with requirements specifications is that they are imprecise and have many inconsistences. Specifications are often written using a natural written language style or, on the contrary, they are too general. In both cases, the intentions of the writer are hard to understand and interpret causing ambiguity. The majority of requirements specification writers tend to mix descriptions of the system's behaviour, quality or appearance with descriptions of objects (or: notions) contained in the problem domain. Definitions of notions are buried in many different places inside scenarios, stories or simply free text. What is more harmful, the same notions often have conflicting definitions and, on the other hand, a number of different synonyms are used to describe identically (or close to identically) defined notions.

Having the requirements specification of such a poor quality, it is a very hard task to build a system that fully fulfills the *real* clients' needs. It is hard to reflect these requirements in the architecture and in the design of the prospective system as well as to apply changes in the system when requirements change. Finally, imprecise requirements make it close to impossible to apply the concept of software reuse at the level of problem definition.

To overcome all problems mentioned above, we need a separate part of our language that supports creating precise and consistent requirements specifications through the introduction of a separate specification of the domain. The precision of requirements specifications is assured by using hyperlinks that link requirements text with definitions of phrases and terms. These hyperlinks can be embedded in free text requirement representations, structured language representations (like SVO sentences) and textual parts of model-based representations.

Let's consider a simple scenario, forming part of a requirement representation written in the SVO format. This scenario describes a sequence of interactions between a customer of a fitness club and the fitness club system:

- Customer wants to sign up for exercises.
- System shows time schedule.
- Customer chooses time from time schedule.

With these simple sentences we can precisely describe actions performed by the actor (Customer or System). It can be noted, however, that such sentences do not contain definitions of notions used therein. For example, we lack an explanation of what 'exercises' or 'time schedule' is, and how they relate to each other. In order to avoid inconsistencies, as mentioned above, we should not insert definitions of notions into the sentences. In fact, the RSL does not allow for inserting such definitions into structured sentences (like SVO). However, no restriction is set on natural language sentences. With such sentences, the requirements specifiers are strongly advised not to put domain element definitions into this free text. Instead, a hyperlink should be introduced that points to an appropriate definition in the domain specification. This prevents from introducing contradictory definitions of the same notion in different places of the requirements specification.

Considering the above, the Requirements Specification Language provides a means to describe the problem domain of the system. We can crete *domain specifications* that contain *domain element packages* – repositories that keep all necessary notions from the domain along with their definitions and relationships between them. For the above example, the domain specification would contain the following definitions for nouns:

Exercises – Form of physical activity performed in fitness club. Exercises may be [cyclic exercises] or [sporadic exercises].

Time schedule – A program of [exercises] offered to [customers] by the [fitness club] in a given period of time: day, week or month.

Square brackets in notion definitions denote relationships with other notions. Every notion in the domain specification can have different forms (i.e. singular and plural), synonyms and homonyms. In addition to nouns, the domain specification can also contain verbs. However, verbs do not have their own autonomous definitions – they are related to nouns as their meaning depends on the context of a concrete noun. Verbs are treated as behavioural features of related nouns. For example, "*choose* exercise" has a different meaning than "*choose* time from time schedule", although both contain the verb "choose".

The domain specification should be partially created by interviews with the future users of the prospective system as well as with specialists from the problem domain. While writing requirements in the RSL grammar (see the Language Reference part), the writer should have constant access to the domain elements in the domain specification and should be able to insert easily notions directly into sentences. He/she should also be able to extend the domain specification at any time by introducing new notions and their definitions.

Figure 7.1 illustrates the separation of the domain specification from requirements descriptions. In the example, scenario sentences have hyperlinks (depicted as dashed arrow lines) to notions defined in the domain specification (depicted as boxes). The lines connecting the notions in the domain elements denote relationships between them.

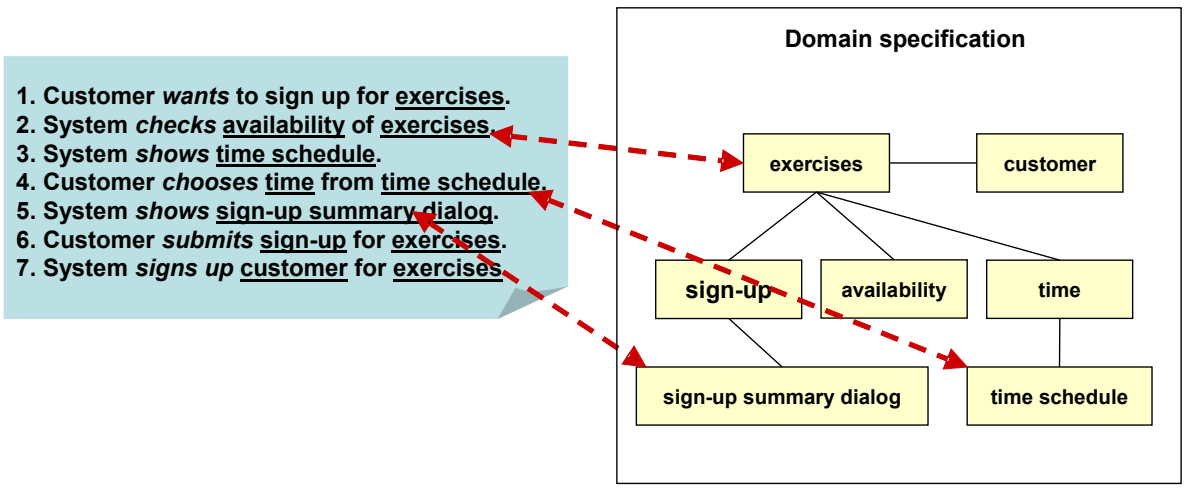


Figure 7.1: Scenario with separated domain specification

7.2 Domain representations using conceptual models

An application domain can be modelled by representing its important concepts and entities in a model. Such conceptual models are usually built from symbols, most popular today is object-oriented representation. An object is usually represented by a class that it is an abstraction of an entity in the application domain.

Unfortunately, representations in object-oriented classes are often confused to be representations of a software design rather than that of an application domain. Such a model typically contains operations in the class definitions, which are already operations of a software object.

That is why we define for our RSL a special kind of structural domain representation that explicitly focuses on *domain elements*. In particular, such a representation does not even allow the inclusion of operations for a conceptual domain model. Apart from that, however, it allows representations according to the following key object-oriented principles:

- Domain elements can be generalised (and inversely specialised).
- Domain elements can be connected through domain element associations.
- Domain elements associations can be aggregations.
- Domain element associations can have multiplicities.

These are actually the same principles as those behind an *ontology*. In contrast to the origin of this notion stemming from the old Greek culture and language, an ontology is today considered a *formal* representation of a domain. In the field of Artificial Intelligence, even special ontology languages have been defined for this purpose, based on formal approaches to so-called knowledge representation.

Still, while not completely formal in a pure sense, a representation in UML or using our domain elements approach derived from it, may still be considered to result in a simple form of a usable ontology.

Even when having a representation in a model according to these principles (being more or less formal), the meaning of a domain entity is often not really obvious for a human and is sometimes just inductively inferred from the name. That is why dictionaries with glossary information are usually suggested, but they are often kept separately and, therefore, not easily accessible.

The RETH tool, for example, embedded for each domain object their glossary entries in a predefined attribute (see, [Kai96, Kai97]). In RSL, we include similar descriptions in notions of the domain elements.

7.3 Domain representation using phrases

Conceptual specification of a particular domain is an important element of any software development project (a so called *software case*). However, conceptual specification is not enough to describe the full specification of the problem domain. This specification should be referenced throughout all the requirements, as pointed out in Section 7.1. The conceptual model allows us to define noun notions, but we also need to define certain phrases containing verbs or adjectives. Such phrases could be used in constrained language requirement representations. At the same time, definitions of these phrases could be kept consistently in a single coherent terminology.

Thus, the RSL introduces a capability to attach to domain elements, certain statements in the form of phrases. The simplest statement contains just the noun with its description (using hypertext). More complex statements contain verb phrases and quantifier or modifier phrases. It is important that every phrase is tightly combined with the noun being part of it. This way, the domain elements are highly structured, where the structure is reflected through the conceptual specification with associations between domain elements (i.e. nouns).

In RSL, every domain element can include many phrases referring to the same noun. These phrases can contain wiki-like descriptions (text with hyperlinks relating to other domain elements or phrases). It can be noted that the name of every domain element is a phrase containing a noun and an optional modifier or quantifier (eg. *user*, *additional user*).

This way, the domain specification becomes coherently embedded into the conceptual model described in the previous section. The RSL shows an overview of all the phrases contained in a domain element. The domain specification can be shown on domain element diagrams (inside domain element icons), and can be presented in a traditional textual form.

Phrases that form the domain specification ensure a clear separation of concerns (behaviour and quality vs. problem domain description) which guarantees consistency of requirements expressed. These phrases form pieces for a constrained language to be used elsewhere throughout the RSL. This approach facilitates the creation of RSL specifications through the use of phrases as atomic “phrase lexemes”. Any phrase can be perceived as a complex domain element. By using phrase lexemes we can easily define grammars based on such complex elements. For

example we can define a Subject-Verb-Objects (SVO[O]) sentence using just two phrases: a subject phrase and a predicate phrase. Subject can include any noun from the conceptual specification optionally grouped with its quantifier and modifier (which together form a noun phrase, e.g. *every registered customer*, *authorised user*). Predicate is a more complex phrase, containing a verb and possibly referring to some other notion (it is a noun phrase). Such a VerbPhrase forms the VO (simple verb phrase) or VOO (complex verb phrase pointing also to another notion) part of the sentence. Let's now consider the following example sentences that use the SVO(O) grammar:

- *User submits form.*
- *System adds user to the user list.*
- *Registered customer cancels reservation.*

The first sentence consists of the *user* phrase (a noun with no quantifier or modifier) in the role of a subject and the *submits form* simple verb phrase in the role of a predicate (VO part of the sentence). The second sentence consists of the *system* phrase in the role of a subject and the *adds user to the user list* complex verb phrase (pointing to the *user list* notion) in the role of predicate (VOO part of the sentence). In the third sentence we have an example of a more complex noun phrase *registered customer* (containing a modifier). For more details and examples please refer to Chapter 6

7.4 Terminology

In order to build phrases as described in the previous section, we need terms. Terms are universal and have general meaning as specified in natural language dictionaries, and they have specific meaning in the context of a software project (or group of projects). Terms have inflections depending on the particular natural language used (English has different inflections than Polish for instance).

The dictionary defines terms (nouns, verbs, adjectives, etc.) with their inflections. Every term is equivalent to a lexeme used in other specifications. It can be noted however, that for a single term, many lexemes can be formed — one for each natural language that we use in our specifications. The dictionary organised in this form can greatly facilitate formulation and comparison of requirements in different languages (especially important on the EU market).

Thus, in RSL we separate the phrases for a given domain from a global dictionary common for all software cases. This separation allows facilitating reuse of specifications in the RSL, as it allows for combining the common dictionary with a thesaurus.

The word *thesaurus* originates in Latin and Greek, meaning "treasury". In the 19th century, a thesaurus was a book of jargon for a specialised field. Today, a thesaurus is an ordered composition of terms from an application domain or area with their relations [Bro03]. Relations defined in a thesaurus are, e.g., *synonym* (similar term), *antonym* (opposite) and hierarchical relations ("broader term" and "narrower term").[1] These relations define the lexical semantics of terms. Based on the lexical semantics of terms it is possible to compare requirements, i.e., the similarity of requirements can be measured based on the lexical semantics defined in the thesaurus.

In fact, a dictionary and a thesaurus complement one another. The dictionary provides a textual description, information about inflections and possibly translations to other languages, while the thesaurus describes the relation of a term to other terms. In RSL this combination of dictionary and thesaurus is placed in the *terminology* package. The RSL allows for specifying a thesaurus through adding relationships between terms. These relationships can be depicted graphically as specified in the language reference.

The combination of dictionary and thesaurus features within one structure is not new. The WordNet lexical database is a well-known example of such a combination that interlinks English nouns, verbs, adjectives and adverbs in a wordnet [MBF$^+$90], [Mil90], [GM90], [BMT90]. Similar wordnets also exist for other European languages (e.g., for German see the Wortschatz-lexikon http://wortschatz.uni-leipzig.de). EuroWordNet, for example was a European resources and development project[2] providing a multilingual database with wordnets for several European languages (Dutch, Italian, Spanish, German, French, Czech and Estonian) [VPG99]. Lexical semantics defined in wordnets can be used to improve the results in information retrieval [Kur04], [CEE$^+$04], e.g., through support for resolving ambiguities.

The use of a wordnet in information retrieval is not new as stated above. Using a wordnet for comparing software requirements is innovative, however.

[1] Thesauri are standardised, e.g. in DIN 1463-1 (German Industry Norm) and ISO 2788 (International Standardisation Organisation).

[2] Project reference number LE-2 4003 & LE-4 8328; http://www.illc.uva.nl/EuroWordNet/index.html

Chapter 8

Representing the user interface and its dynamics

8.1 Elements of the user interface

In order to enable analysts specify user interface requirements in a notation understandable to their stakeholders, prototype-oriented model-based elements are required. This section defines such user interface elements. Since concrete user interface elements mainly differ according to modality and toolkit and there are currently many modalities and toolkits, general user interface elements should be defined independent therefrom. The RSL therefore provides general usage-oriented UIElements that are both modality and toolkit independent. Extending the UIElements and hence tailoring them to concrete elements of a specific domain is an important requirement, which is done by using UML profiles.

In order to obtain general UIElements, the usage of different concrete user interface elements from different sources were analyzed. There are elements for direct interaction with the user and others for structuring the user interface for example by grouping other elements. Consequently the former elements are regarded as simple or atomic while the latter ones are containers. This modality and toolkit independent classification is showed in Figure 8.1.

Chapter 15 of this document provides a detailed description of these elements.

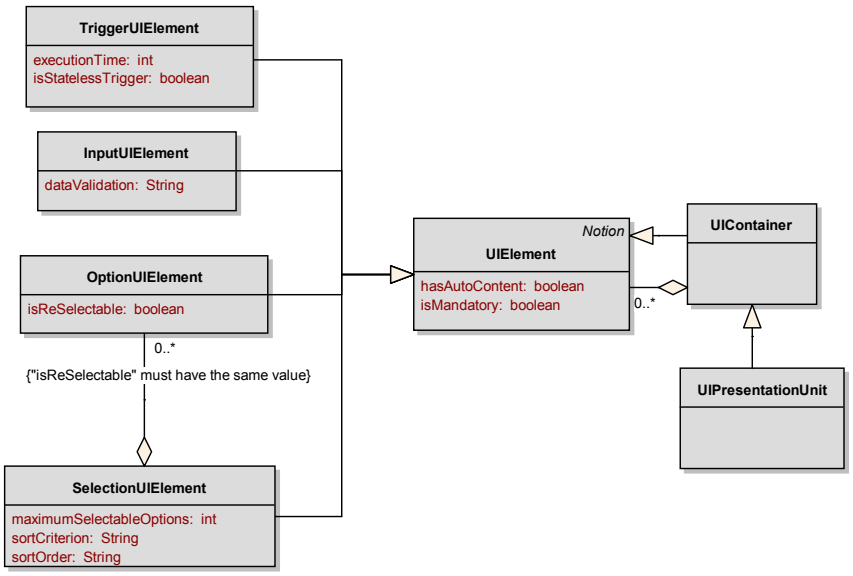


Figure 8.1: Usage-oriented UIElements

8.2 Behaviour of the user interface

The UI representation part of the RSL gives us means to create models of the user interface for the prospective system. With this language we can precisely describe the structure of elements that build the UI. Description of the structure, however, is not enough – we need to have the possibility to express the behaviour of the UI. The UI behaviour representation part of the RSL should allow for describing navigability and interactivity of the system's UI. In particular, it should describe how the system's UI will behave in response to user actions. This description should give a high-level overview of the UI behaviour in order to allow the users to quickly understand and verify it.

The behaviour of the prospective system's UI can be modelled in three ways using the proposed UI behaviour representation part of the RSL. Firstly, it can be used to illustrate interactions between a user and the system within a single use case scenario. A good way of illustrating scenarios are storyboards, mostly known from the Rational Unified Process (RUP) [Kru03]. Storyboards are simply scenarios with screenshots attached to scenario sentences illustrating how the UI changes in response to user or system actions expressed in these sentences. Secondly, with the UI behaviour representation part of the RSL we can describe the overall flow of the application's UI – a combination of UI behaviours for one or more use cases. Such an architectural overview of the UI can be visualised by showing UI elements with labelled associations illustrating user actions that triggers transitions of the interface. Having illustrated all possible ways of navigation through the application's interface it is easy for the users to gain an understanding of how it works and determine if it will be usable. Another way of modeling

UI behaviour is associating elements of user interface (UIElements) with RequirementsSpecifications :: UseCases – every UIElement can trigger a Use Case, in other words, user can begin interaction with system by using element of interface (graphical, voice or other).

Chapter 16 of this document provides a detailed description on how the UI behaviour can be represented using the UI behaviour representation part of the RSL.

Chapter 9

Discussion

Requirements can be compared to novels in literature. Good novels communicate stories treated as sequences of events, and place these stories in a well described environment. Unfortunately, writing "stories" that describe requirements for software systems seems to be equally as hard as (or harder) than as writing good novels. However, unlike writing novels, lack of coherence and ambiguities may cause disaster when developing a system based on such requirements.

Finding inconsistencies in a set of several tens or hundreds of requirements is quite a hard task, especially, when these requirements are written by different people and at different times. It seems that keeping the vocabulary separate from the rest of the requirements specification can significantly facilitate keeping sparse requirements documents consistent by keeping the vocabulary controlled. This is because most inconsistencies in requirements are caused by contradictory definitions of terms. To eliminate the source of such inconsistencies we introduce a single repository of notions (a vocabulary) that can be used in various requirements. This means that for instance, the behavioural requirements could use definitions already found in the repository and just concentrate on the actual sequence of events.

In addition to the above, having a clearly defined vocabulary makes it possible to introduce certain query mechanisms that would allow for easy retrieval and reuse of requirements. For such mechanisms it is very important to be able to compare requirements. This comparison should be based on a terminology where terms with similar meaning are related.

In fact, our thesaurus (as integrated with a dictionary in our vocabulary representation) involves a specialisation relation (and indirectly the inverse generalisation). While this is not unusual for a thesaurus, a complication arises since we link the thesaurus also with a conceptual model, which involves generalisation / specialisation as well. As long as only one of these approaches

will be used for domain representation, this does not pose a problem. Just to the contrary, in the absence of a conceptual model, a thesaurus can define such hierarchies. Whenever both approaches will be used at the same time, however, these relations are somewhat redundant and may, therefore, result in inconsistencies.

One approach to tackle this problem may be to automatically generate a generalisation / specialisation hierarchy for a not yet existing conceptual model from the related hierarchy in the thesaurus (using tool support, of course). Such an approach was taken in [SK94] for a unified hypertext and structured object representation as defined in [KS91]. This representation is similar enough to ours to make this approach promising. Another approach would be to implement automatic consistency checks in a tool. Since one representation is clearly less formal than the other, this approach may be difficult to pursue.

So, it has to be stressed that the language to define vocabularies should be used in conjunction with a tool. This is highly recommended as using notions stored in a vocabulary within requirements and keeping it constantly coherent would be very laborious and error-prone if done manually. Thus such a tool would need to allow for providing consistency between different requirements using the same notions (the notion has the same definition wherever it is used).

Going back to the original IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology definition of requirement that we used for this project:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a condition or capability as in (1) or (2).

we can see in (3) that it includes the documented *representation* of the requirement. During this project we realised that the documented representation of the requirement should not be considered to be the requirement itself, but a representation. There are several reasons:

- A requirement is something that exists even if it is never documented, i.e., represented.
- A requirement can have multiple representations.
- A requirement can be represented at different abstraction levels.

- Each representation of a requirement is usually not complete from each possible perspective.

The first issue tackles the fact that requirements and their representations belong to two different dimensions. Similar to real world entities and the OOA representations, an orange in the real world is different than an orange represented using UML. Besides the fact that it is very hard to elicit all requirements and represent them properly, even when they are elicited, their representations are not necessarily the right ones.

The second issue is that each requirement can be represented in multiple ways and some can be represented in certain ways that others cannot. For example requirements concerning different sound types and ranges are hard to capture using UML sequence diagrams.

The third issue is that requirements can be represented at different abstraction levels leaving some part of the actual requirement out. By doing this any single requirement representation abstraction level represents no full requirement. On the other hand, requirements themselves are harder to decompose into multiple abstraction levels.

Finally, taking into account that each requirement can be represented in a number of different ways, and for each way at different abstraction levels, it is obvious that almost any requirements representation cannot be considered as complete representation of the requirement and as such different than the requirement itself.

This distinction between requirements and their representations is evident throughout our language. As such, it removes the confusion between requirements themselves and their representations that commonly exists in the requirements engineering community. The removal of this problem is one of the prerequisites for successful capture of the requirements, proper traceability, and quality insurance involved in checking that requirement specification is consistent and complete. This is one of the major contributions of our language and this project.

Part II

Language Reference

Chapter 10

Kernel

10.1 Overview

The Kernel part groups the most important concepts of the Requirements Specification Language: Elements, their Relationships, Representations and Attributes. All key meta-classes of the RSL are somehow dependent on Kernel (mostly by generalisations), which was designed as a central part of the RSL, making it more coherent and understandable.

Kernel also acts as a layer between the RSL and the Kernel of UML – the concepts of Package and Relationship from UML Kernel are refined in the RSL Kernel.

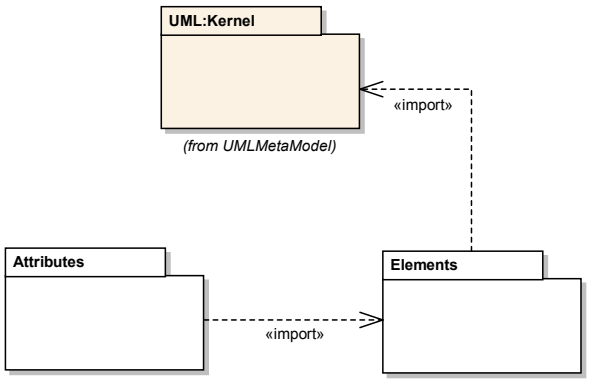


Figure 10.1: Overview of packages inside the Kernel part of RSL

Figure 10.1 shows two packages of the Kernel part of the language. The Elements package contains all the basic entities while the Attributes package adds attribute values to these entities. Both packages depend on certain elements defined in the UML :: Kernel packages. This way,

the RSL can be treated as an extension of UML, but in a very broad sense, as many elements of the language do not necessarily originate in UML.

10.2 Attributes

10.2.1 Overview

RSL Kernel is designed to reflect the concept of Element and its Representation. The Attributes package serves the purpose of attaching Attributes and AttributeSets to Elements :: RepresentableElements (that is Elements :: Elements having a representation).

The main concept behind the Attributes package is to allow grouping of attributes in sets and to separate attributes treated as containers for values from these attributes' definitions.

10.2.2 Abstract syntax and semantics

The diagram in Figure 10.2 shows the abstract syntax of the Attributes package. The following subsections will describe classes in this diagram.

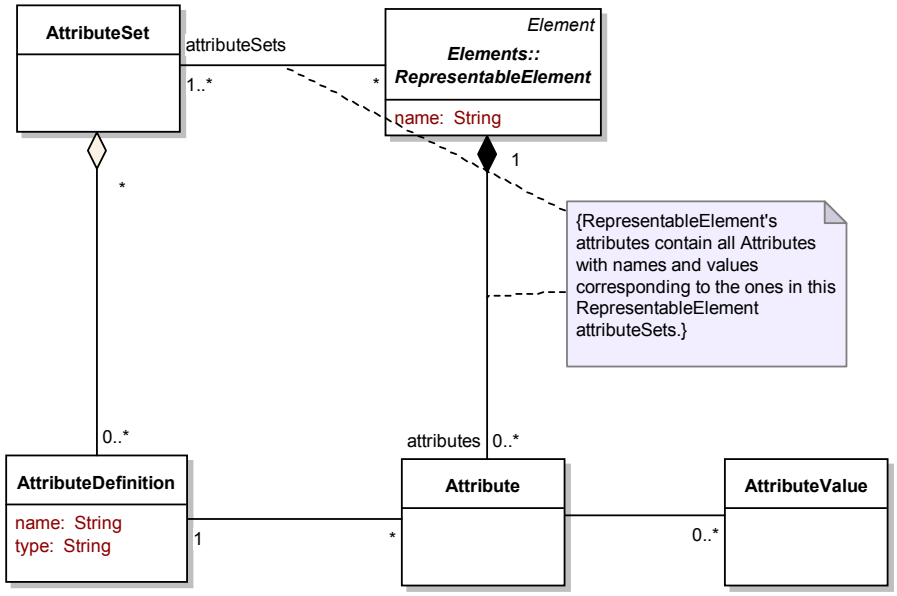


Figure 10.2: Attributes

Attribute

Semantics. An Attribute is an entity containing properties of various elements. Attribute is defined by AttributeDefinition. Attributes can be grouped in AttributeSets. Attribute is a container for value(s), which are attached to Elements :: RepresentableElements (which is constrained by existence of Attribute's definition in an AttributeSet corresponding to this Elements :: RepresentableElement).
Abstract syntax. Attribute is a component of Elements :: RepresentableElement, but only if its definition exists in the AttributeSet associated with this Elements :: RepresentableElement. Attribute is also associated with AttributeDefinition (many Attributes can be associated with one definition). Attribute can have multiple AttributeValues associated with it.

AttributeDefinition

Semantics. AttributeDefinition is a description of associated Attribute and a constraint for its values. Attribute can have a name (AttributeDefinition's name attribute) and type (type attribute, which determines the set of possible values of AttributeValue).
Abstract syntax. AttributeDefinition can be a component of several AttributeSets. AttributeDefinition can be associated with Attributes.

AttributeSet

Semantics. AttributeSet is an entity grouping Attributes via their definitions (AttributeDefinition). AttributeSets can be associated with Elements :: RepresentableElements, which restricts attachment of Attributes to those elements. AttributeSet can be treated as a template for Attributes that should be connected to given Elements :: RepresentableElement.
Abstract syntax. AttributeSet is a container for AttributeDefinitions. AttributeSet is associated with Elements :: RepresentableElements.

AttributeValue

Semantics. AttributeValue represents a value of a property associated with a Elements :: RepresentableElement.
Abstract syntax. AttributeValues can be associated with an Attribute.

10.2.3 Concrete syntax and examples

Attribute. Attribute is expressed through its name and value in the following form:

```
attribute name = value
```

for example

```
version = 345
```

If Attribute has multiple values that are separated by commas:

```
attribute name = value1, value2, ..., valueN
```

for example:

```
Author = John Smith, Alice Brown
```

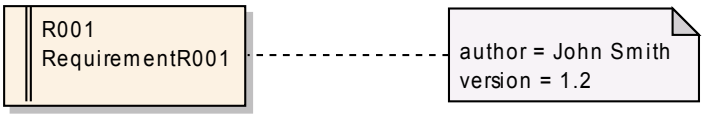


Figure 10.3: Showing Requirements Attributes on a diagram

When showing Attributes on a requirements diagram the above syntax can be placed in a "note" (similar to a UML Comment): in a rectangle with upper right corner bent and connected to the given Elements :: RepresentableElement by a dashed line (see figure 10.3).

When showing Attributes as properties of a Elements :: RepresentableElement it can be represented in tabular form:

Name	Value(s)
attribute1 name	value1, value2, . . . , valueN
attribute2 name	value1, value2, . . . , valueN
. . .	. . .
attributeN name	value1, value2, . . . , valueN

AttributeDefinition. Attribute can be shown in the form of a listing of its attributes and values with a caption showing its definition's name:

```
attribute definition name
name = attribute name
type = attribute type
```

for example

```
Version attribute definition
name = version
type = Integer
```

AttributeSet. AttributeSet can be shown as a listing of attributes' names and types defined in the AttributeDefinitions owned by this AttributeSet (every attribute in separate line), along with this AttributeSet's name:

```
attribute set name
attribute1 name (attribute1 type)
attribute2 name (attribute2 type)
...
attributeN name (attributeN type)
```

for example

```
Requirements attributes
author (String)
version (Number)
```

AttributeValue. AttributeValue can be shown in a way similar to the Attribute (see above).

10.3 Elements

10.3.1 Overview

The Elements package defines basic entities for the RSL, relationships between them and a mechanism for grouping RSL elements. The package meta-classes reflect the RSL's concept of separating Elements from their *representations*. The Elements package also introduces two ways of linking entities – it is done either through relationship meta-classes or through Hypelinks.

10.3.2 Abstract syntax and semantics

The diagrams in Figures 10.4 through to 10.6 show the abstract syntax of the Elements package. The following subsections will describe classes in these diagrams.

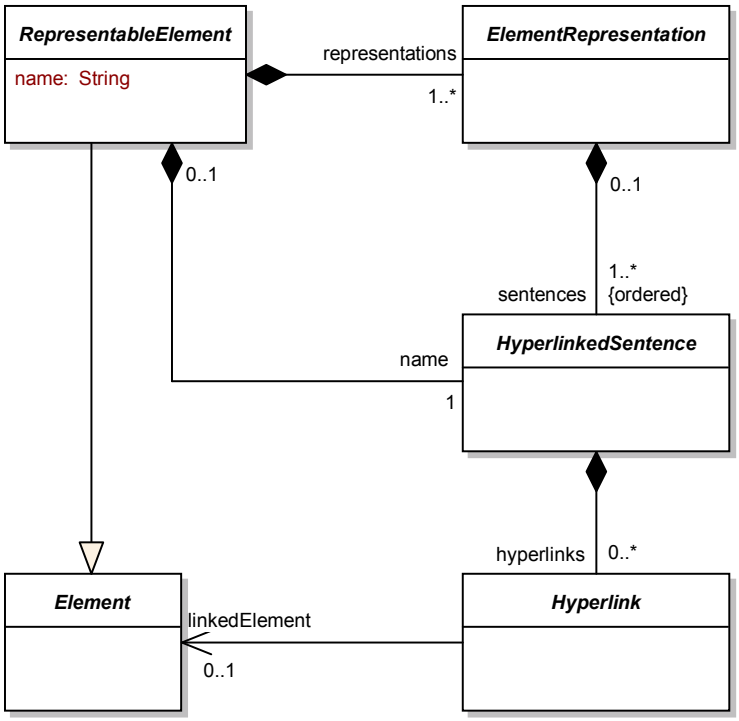


Figure 10.4: Element representations

Element

Semantics. An Element is the most basic entity existing in the RSL. It has a capability of being linked by Hyperlinks.

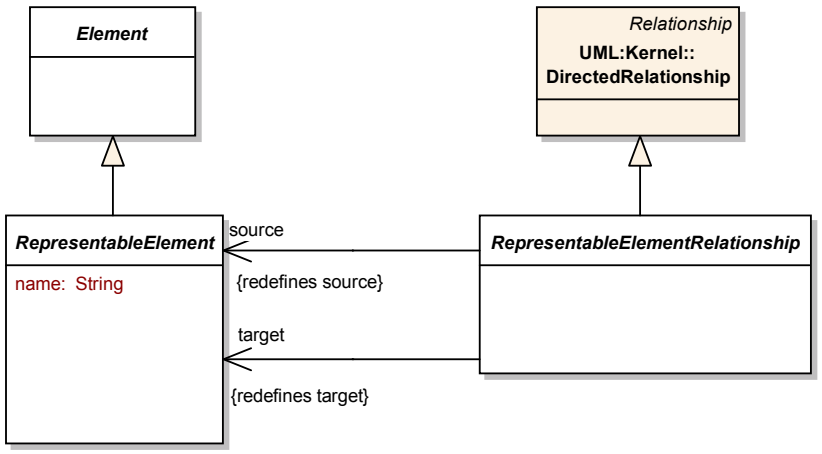


Figure 10.5: Element relationships

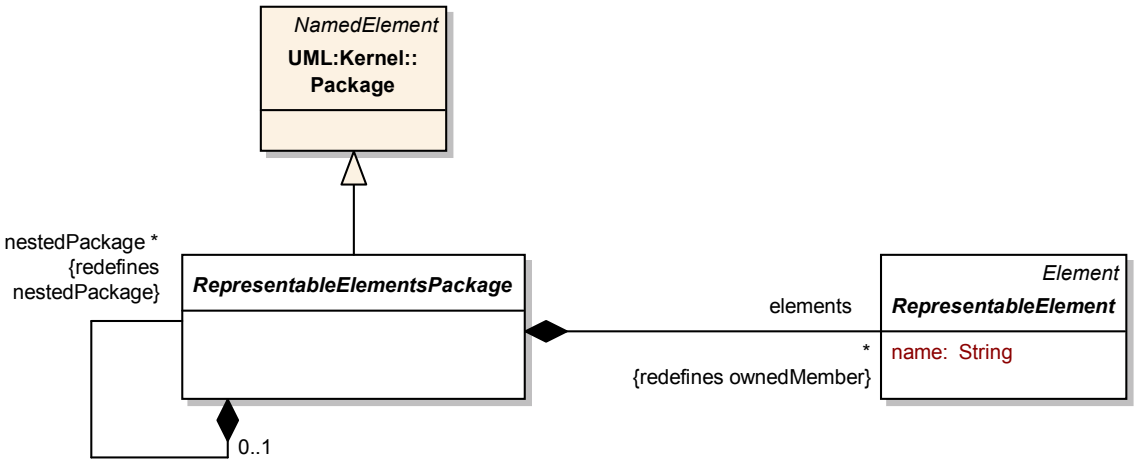


Figure 10.6: Element packages

Abstract syntax. Element can be associated with Hyperlinks (it has the role of linkedElement in this relationship). Element is a superclass for RepresentableElement. Element is abstract.

ElementRepresentation

Semantics. ElementRepresentation is a superclass for all *representation* classes in the RSL. It contains HyperlinkedSentences as sentences forming the representation.

Abstract syntax. ElementRepresentation is a superclass for DomainElementRepresentation, RequirementRepresentation and UIElementRepresentation. ElementRepresentation is a component of RepresentableElement with the role of representation. ElementRepresentation consists of HyperlinkedSentences. ElementRepresentation is abstract.

Hyperlink

Semantics. Hyperlink is used for linking Elements. HyperlinkedSentences are formed of Hyperlinks.
Abstract syntax. Hyperlink may point at exactly one Element. Hyperlink is a component of HyperlinkedSentence. Hyperlink is a base class for Phrases :: PhraseHyperlink and Term :: TermHyperlink. Hyperlink is abstract.

HyperlinkedSentence

Semantics. HyperlinkedSentence is a base class for various types of sentences in RSL. HyperlinkedSentence is a set of Hyperlinks used for representing RepresentableElements.
Abstract syntax. HyperlinkedSentence consists of Hyperlinks. HyperlinkedSentence is a component of ElementRepresentation (with the role of sentences) and RepresentableElement (with the role of name). HyperlinkedSentence is a superclass for NaturalLanguageHypertextSentence, ConstrainedLanguageSentence and DomainElementHyperlinkedSentence. HyperlinkedSentence is abstract.

RepresentableElement

Semantics. RepresentableElement is a base class for Element's specialisations which have a representation. RepresentableElements can be grouped in RepresentableElementsPackages. RepresentableElement can be source and target for RepresentableElementRelationships, allowing all elements that have representation to be linked in way other than by Hyperlinks. Every RepresentableElement has its name.
Abstract syntax. RepresentableElement is a kind of Element. RepresentableElement has the name attribute. RepresentableElement consists of its 'name' (a HyperlinkedSentence) and its 'representations' (ElementRepresentations). RepresentableElement can be the source and/or target for RepresentableElementRelationship. RepresentableElement is a component of RepresentableElementPackage. RepresentableElement consists of Attributes :: Attributes and is associated with an Attributes :: AttributeSet. RepresentableElement is a superclass for Requirement, DomainElement and DomainStatement. RepresentableElement is abstract.

RepresentableElementRelationship

Semantics. RepresentableElementRelationship is used for connecting representable elements in a way other than through Hyperlinks.
Abstract syntax. RepresentableElementRelationship is a kind of UML :: Kernel :: DirectedRelationship. It is associated to RepresentableElements - being its 'source' and 'target'. RepresentableElementRelationship is a superclass for DomainElements :: DomainElementRelationship, Notions :: NotionGeneralisation, UseCaseRelationships :: Participation, RequirementRelationships :: RequirementRelationship, RequirementRelationships :: RequirementVocabularyRelationship and UseCaseRelationships :: Usage. RepresentableElementRelationships is abstract.

RepresentableElementsPackage

Semantics. RepresentableElementsPackage is entity used for grouping of various kinds of RepresentableElements in packages.
Abstract syntax. RepresentableElementsPackage is a kind of UML :: Kernel :: Package. RepresentableElementsPackage consists of elements – RepresentableElements. RepresentableElementsPackage can contain nested packages. RepresentableElementsPackage is a superclass for DomainElements :: DomainElementsPackage, DomainElements :: DomainSpecification, RequirementSpecifications :: RequirementsPackage and RequirementSpecifications :: RequirementsSpecification. RepresentableElementsPackage is abstract.

10.3.3 Concrete syntax and examples

As abstract meta-classes, all classes in the Elements package do not have concrete syntax. Most of them contain several concrete subclasses in which concrete syntax is defined.

Chapter 11

Requirements

11.1 Overview

The Requirements part defines all the RSL constructs that pertain to Requirements as such and relationships between them. This part of the language defines only the top level elements which do not show details of individual representations of Requirements. Language users will typically use elements from this part to present requirements as such (in diagrams and project trees) contained in the requirements specifications they create. These diagrams or trees will generally consist of icons denoting individual Requirements (including UseCases) and lines denoting appropriate RequirementRelationships (including relationships for UseCases).

The specification in this part contains three packages, as shown in Figure 11.1 (marked in blue on colour print-outs).

- The RequirementsSpecifications package contains all the general constructs. These constructs allow for expressing whole requirements specifications, groups of logically related requirements (their packages) and individual requirements. It «import»s from the Kernel :: Elements package to allow for specialising the syntax and semantics of general meta-classes from this package. It also «import»s from the UML :: Kernel package to allow for using UML elements. It «merge»s the UML :: UseCases package as it redefines the UML's UseCase class. Different relationships between requirements are defined in the RequirementsRelationships package.
- The RequirementsRelationships package generally introduces the possibility to relate individual requirements. Different types of dependencies between individual requirements

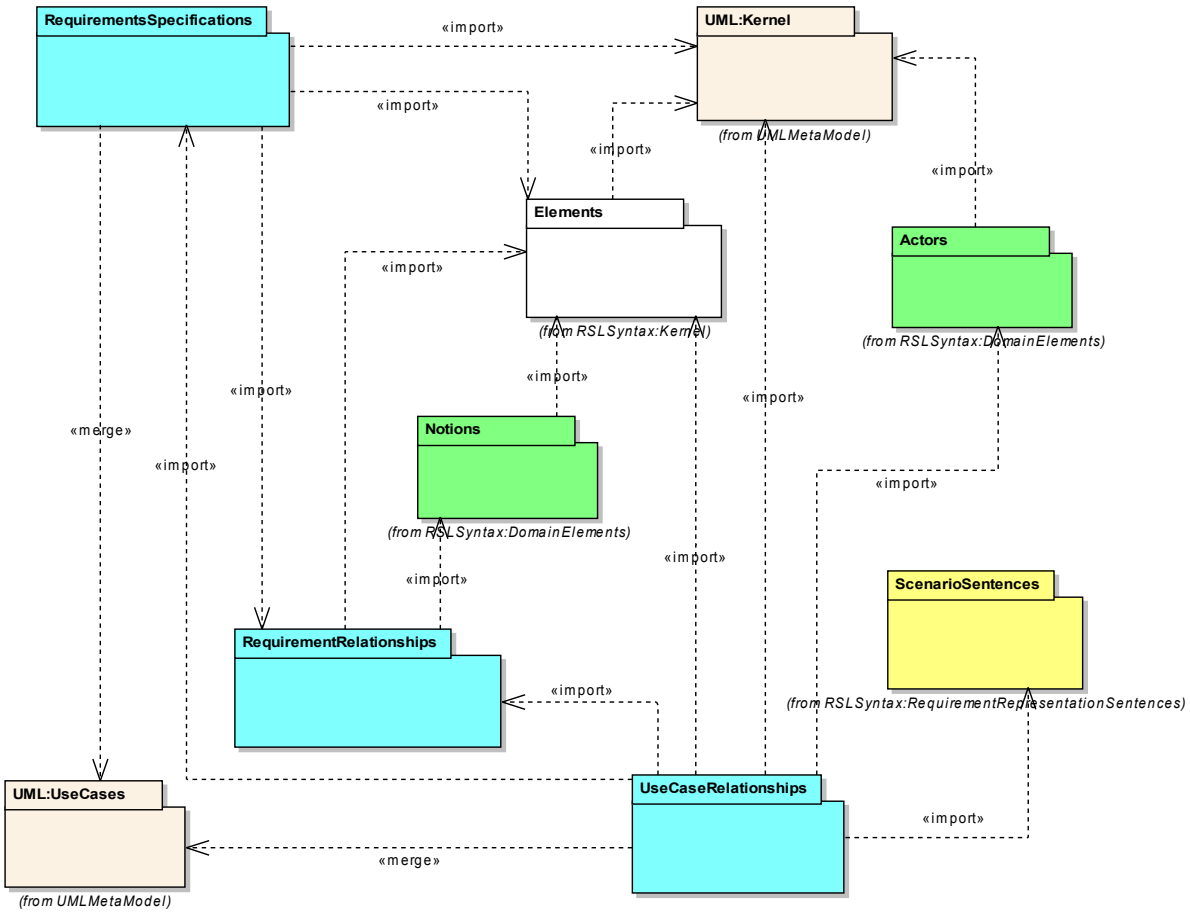


Figure 11.1: Overview of packages inside the Requirements part of RSL

can be expressed by appropriate relationships defined in this package. In this way, conceptual relationships between Requirements as specified in Chapter 4 can be expressed. This package imports from Kernel :: Elements in order to reuse the syntax and semantics of more general elements. Since the relationships defined in this package include those connecting requirements to notions, also the DomainElements :: Notions package is imported.

- The UseCaseRelationships package is a modification of the UML :: UseCases package with which it «merges» to redefine use case relationship classes. This package also «import»s from DomainElements :: Actors – it relates UseCases with DomainElements :: Actors :: Actors, which makes such relationships explicit as opposed to what is defined in UML. Moreover, the package imports the RequirementRepresentationSentences :: Scenariosentences package. This is because the «invoke» relationship points to a sentence in a scenario which needs to be associated with it.

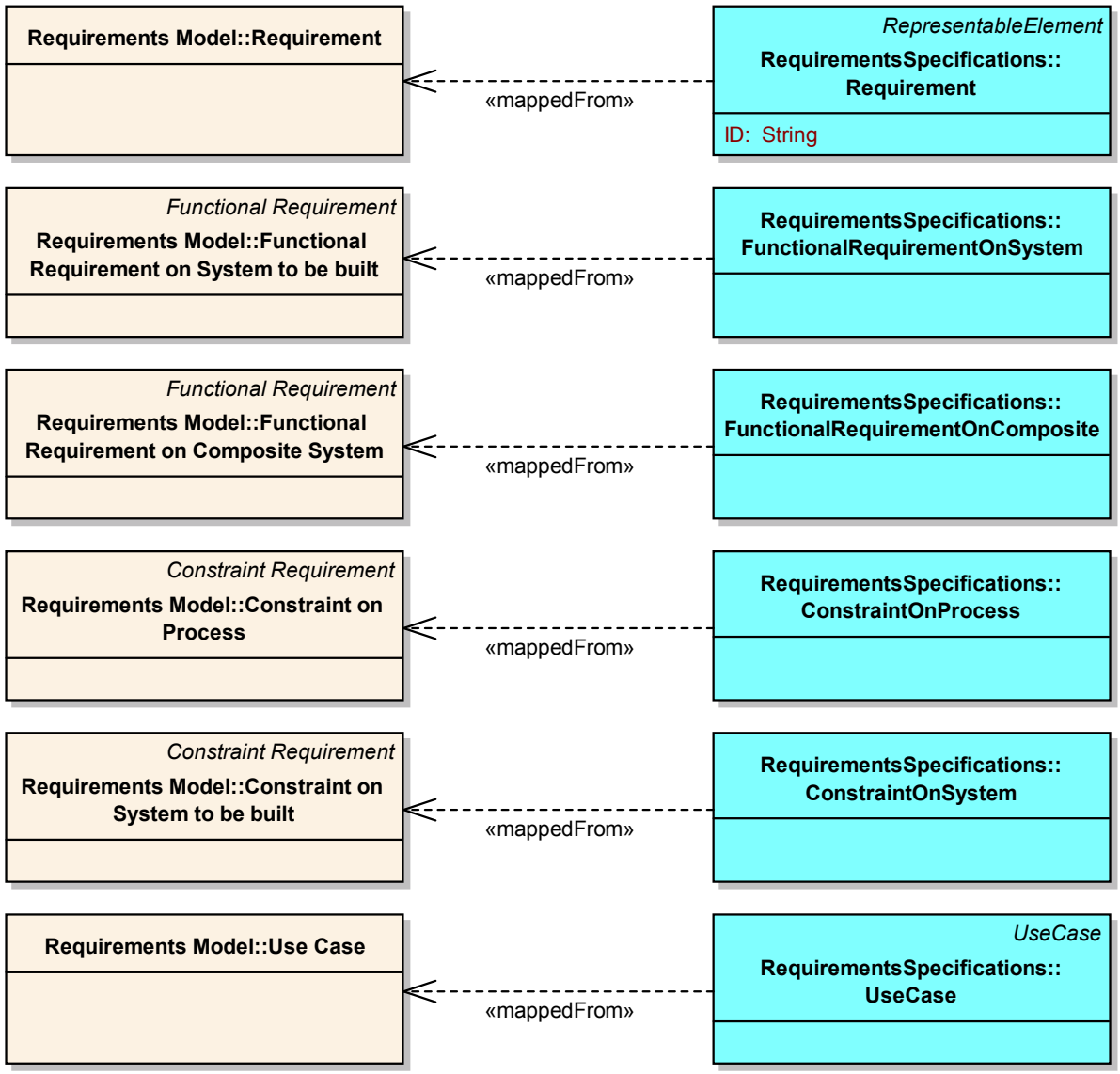


Figure 11.2: Mappings between meta-classes representing requirements from the Requirements package and meta-classes from the conceptual model

Individual classes in the above packages can be mapped from [1] the conceptual model described in Chapter 4. Mappings between conceptual requirements and requirements meta-classes in the RequirementsSpecifications package are shown in Figure 11.2. Figure 11.3 shows from which conceptual meta-associations, the requirement relationship meta-classes are mapped.

The main class in this part is the Requirement class and all its specialisations (see Fig. 11.2). These classes allow for expressing requirements as such without going into details of the requirement's representation. The source of these classes are the Requirements Model :: Requirement class from the conceptual model and its subclasses. It can be noted that among possible

[1] «mappedFrom» specifies a relationship between RSL model elements that represent corresponding ideas in the conceptual model.

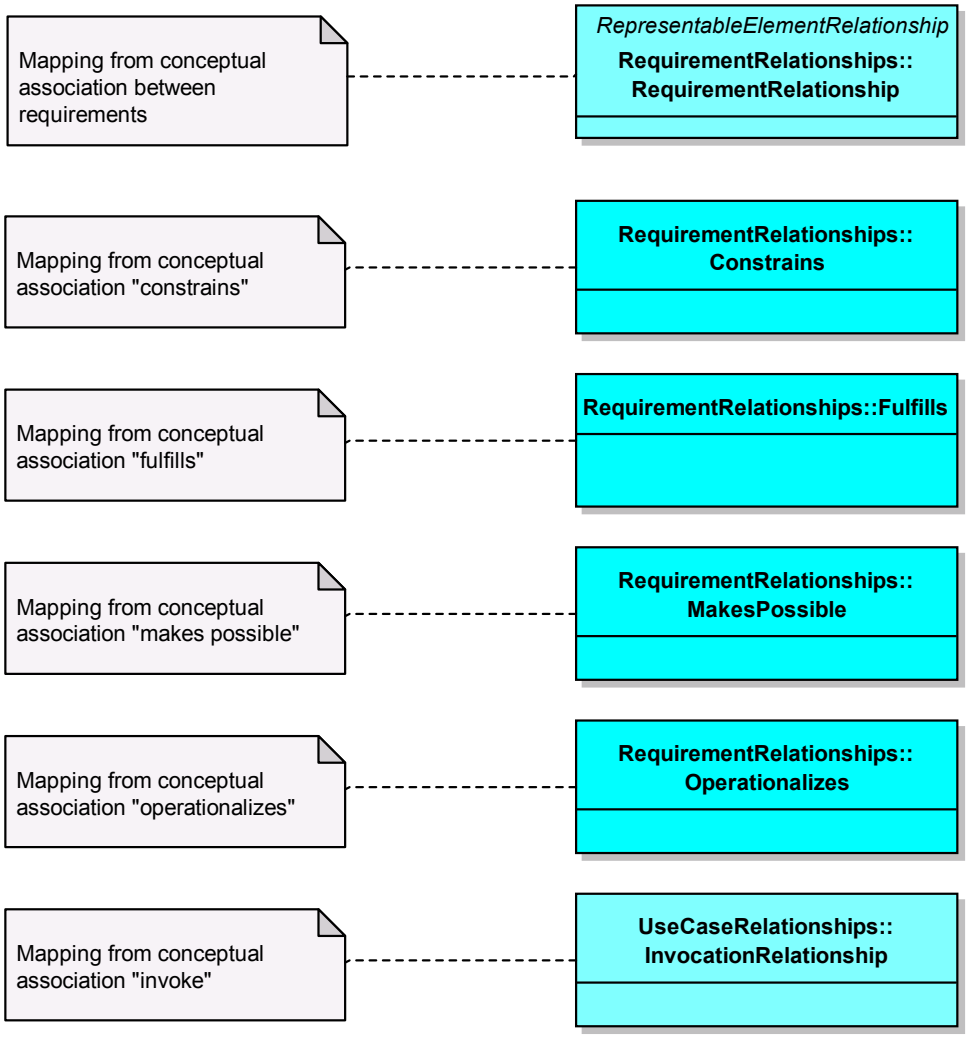


Figure 11.3: Mappings between meta-classes representing requirements relationships from the Requirements package and meta-classes from the conceptual model

specialised requirements types, use cases are special. This is because of specific syntax for relationships between UseCases. As for other types, this class directly traces from the Requirements Model :: UseCase class found in the conceptual model. However, a separate package in the RSL syntax meta-model is devoted only to use case relationships.

In addition, specialisations of the RequirementRelationship class (see Fig. 11.3) allow for expressing in RSL connections of Requirements through different relation types defined in the conceptual model: ‘invokes’, ‘constrains’, ‘operationalizes’, ‘fulfils’ and ‘makes possible’. Other relationships in this part include relationships that pertain to the UseCase class and are not directly described in the conceptual model. However, these relationships directly specialise and modify appropriate relationships from the UML model.

All the elements contained in the Requirements part can be shown on Requirements Diagrams or Use Case Diagrams. Requirements Diagrams show Requirement icons with relevant Require-

mentRelationships between them. Use Case diagrams show UseCases with DomainElements :: Actors :: Actors, also with appropriate relationships between them. These diagrams do not show the details of individual elements. These details can be shown using diagrams or text as defined in the RequirementRepresentations part.

Requirements can be also logically grouped into larger containers - RequirementPackages. These containers can be shown on Package Diagrams as derived from UML. Containment of Requirements in RequirementPackages can be shown in Project Trees. These trees show containment hierarchies of RequirementsSpecifications with RequirementsPackages and Requirements.

11.2 Requirements specifications

11.2.1 Overview

This package describes the general structure of requirements specifications. This structure is similar to the structure of Models in UML. So, by analogy, we have the RequirementsSpecification class that defines the top level element holding a complete specification of requirements for a specific system. Every such specification has to have a DomainElements :: DomainSpecification and can be divided into many RequirementsPackages. RequirementsPackages can be nested and contain all types of Requirements (including UseCases).

Requirements are presented on Requirements Diagrams as simple rectangle icons with their 'name', 'ID' and type appropriately expressed. This notation is modified for UseCases by substituting rectangles with ovals (for consistency with UML) on Use Case Diagrams. RequirementsSpecifications and RequirementsPackages can be presented on Package Diagrams that have their syntax derived from UML Package Diagrams. Requirements and UseCases can be placed in Project Trees under appropriate Packages and a RequirementSpecification. These trees are presented as any browser tree with appropriate small icons expressing all the above elements.

11.2.2 Abstract syntax and semantics

Abstract syntax for the RequirementsSpecifications package is described in Figures 11.4 and 11.5.

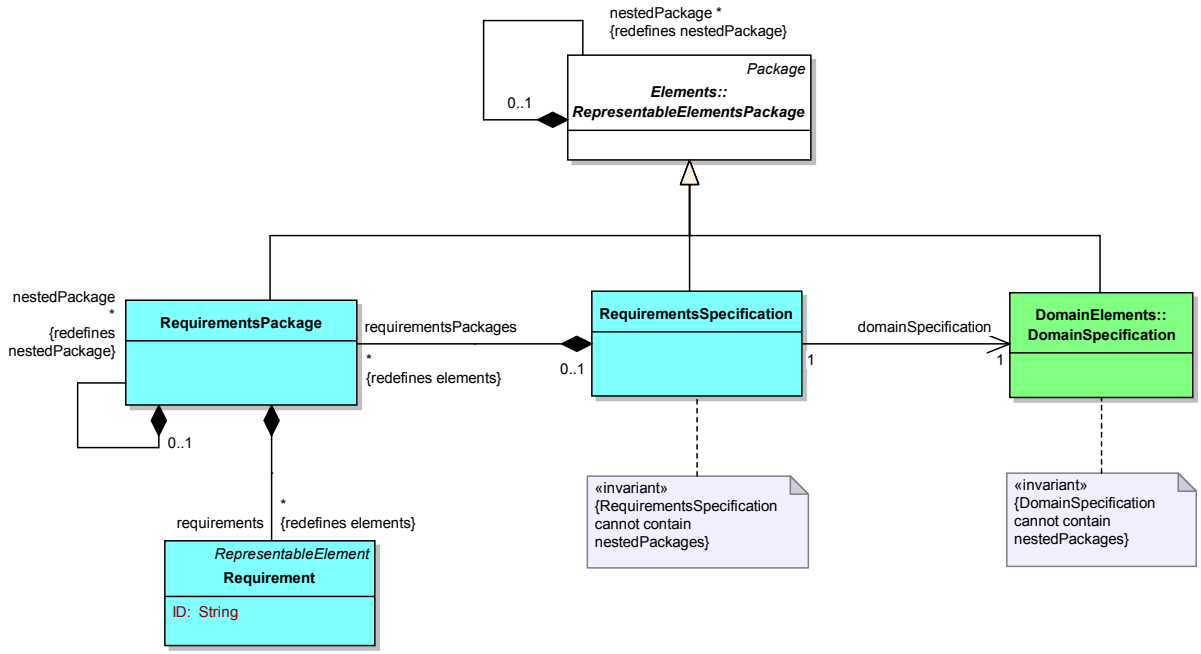


Figure 11.4: Requirements specifications

Requirement

Semantics. Requirement is understood as a placeholder for one or more RequirementRepresentations :: RequirementRepresentations. It is treated as a concise way to symbolise this representation. Requirement is very general and it can express every kind of requirement (functional requirements, constraint requirements, etc.). To express a requirement of a concrete type (functional requirement on system or composite, constraint requirement on system or process, use case) the RSL defines appropriate specialisations of Requirement described below.

Abstract syntax. Requirement is a kind of Elements :: RepresentableElement. Requirement has derived the 'name' property. As a 'name', there can be used any concrete subtype of Elements :: HyperlinkedSentence. Requirement is detailed with one or more 'representations' in the form of RequirementRepresentations :: RequirementRepresentation. Requirements can be related with other Requirements through RequirementRelationships :: RequirementRelationships. Requirements can be grouped into RequirementsSpecifications :: RequirementsPackages. Requirement is a superclass for meta-classes representing requirements of a specific type.

ConstraintOnProcess

Semantics. ConstraintOnProcess is a type of Requirement. It is used to express constraint requirements on process – i.e. requirements that constrain different development process related

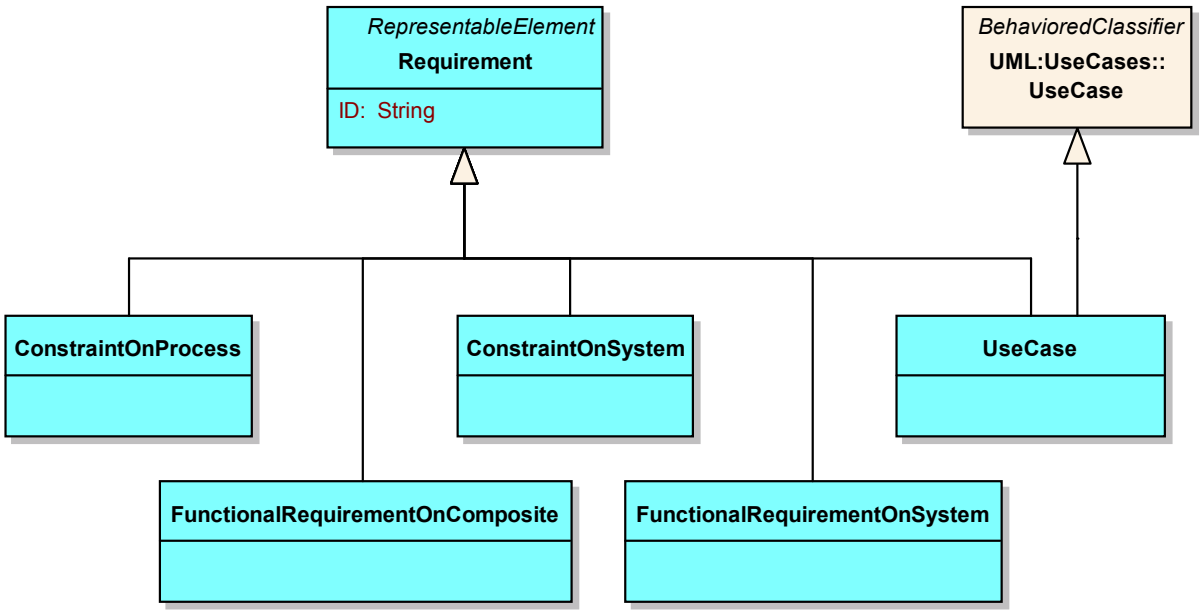


Figure 11.5: Requirement types

decisions.

Abstract syntax. ConstraintOnProcess is a specialisation of Requirement. It derives whole abstract syntax from its superclass.

ConstraintOnSystem

Semantics. ConstraintOnSystem is a type of Requirement. It is used to express constraint requirements on system to be built – i.e. requirements that constrain the functionality of the system to be built.

Abstract syntax. ConstraintOnSystem is a specialisation of Requirement. It derives whole abstract syntax from its superclass.

FunctionalRequirementOnComposite

Semantics. FunctionalRequirementOnComposite is a type of Requirement. It is used to express functional requirements that are supposed to be fulfilled by the system composed of system to be built and its actors.

Abstract syntax. FunctionalRequirementOnComposite is a specialisation of Requirement. It derives whole abstract syntax from its superclass.

FunctionalRequirementOnSystem

Semantics. FunctionalRequirementOnSystem is a type of Requirement. It is used to express functional requirements that are supposed to be fulfilled by the system that is being specified.
Abstract syntax. FunctionalRequirementOnSystem is a specialisation of Requirement. It derives whole abstract syntax from its superclass.

UseCase

Semantics. UseCase has the same meaning as described in the UML Superstructure: "A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system." ([Obj05b]) This definition is analogous to one specified in section 4.2. In accordance with the classification in Chapter 4, this semantics is extended by stating that UseCase is a special kind of Requirement. It is also a placeholder for its Representations which might be of various kinds, as specified in the ConstrainedLanguageRepresentations, ActivityRepresentations and InteractionRepresentations packages. These representations describe the behaviour (set of actions) for the UseCase.
Abstract syntax. UseCase is a specialisation of UML :: UseCases :: UseCase and Requirement. Instances of UseCase can be related with each other by UseCaseRelationships :: InvocationRelationships. This relationship redefines UML's extend and include. UseCase can contain several UseCaseRelationship :: Participation relationships and can be pointed to by UseCaseRelationship :: Usage relationships. These relationships relate it with Actors :: Actors. It can contain InteractionRepresentations :: InteractionScenarios, ConstrainedLanguageRepresentations :: ConstrainedLanguageScenarios and an ActivityRepresentations :: ActivityScenario as its 'representation's. UseCase can be associated with one or more UIElements :: UIElements as their 'triggeredUseCase'.

RequirementsSpecification

Semantics. RequirementsSpecification is a type of Elements :: RepresentableElementsPackage. It can contain all elements of a requirements specification for a given project – Requirements grouped in appropriate packages and elements which form specification of the system domain. RequirementsSpecification is a root package for the whole requirements specification. It can be treated as equivalent of Model from UML.
Abstract syntax. RequirementsSpecification is a specialisation of Elements :: RepresentableEle-

mentsPackage. It redefines 'elements' from the superclass with RequirementsPackages. It is also associated with one DomainElements :: DomainSpecification.

RequirementsPackage

Semantics. RequirementsPackage is a type of Elements :: RepresentableElementsPackage. It can contain Requirements and their specialisations as well as nested RequirementsPackages.
Abstract syntax. RequirementsPackage is a specialisation of Elements :: RepresentableElementsPackage. It redefines 'elements' from the superclass. Owned members for the RequirementsPackage must be Requirements. It also redefines 'nestedPackage', which can only be another RequirementsPackage. Every RequirementsPackage can be part of a RequirementsSpecification.

11.2.3 Concrete syntax and examples

Requirement. It is depicted as a rectangle with two additional vertical lines on its left. Requirement's 'ID' is written in the top left corner of the box. Requirement's 'name' is written inside the rectangle centred horizontally and vertically. See Figure 11.6 for illustration of this, and Figure 11.14 for an example of usage of these icons in a Requirements Diagram.

Optionally, Requirement can have its type shown in the form of text indicating this type surrounded by double angle brackets ("« »", an "angle quote").

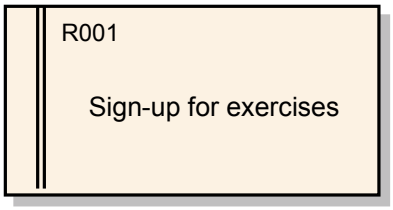


Figure 11.6: Requirement example

ConstraintOnProcess. Concrete syntax is the same as for Requirement except that it has mandatory requirement type indicator – shown as a text surrounded by double angle brackets: "«constraint on process»".

ConstraintOnSystem. Concrete syntax is the same as for Requirement except that it has mandatory requirement type indicator – shown as a text surrounded by double angle brackets: "«constraint on system»".

FunctionalRequirementOnComposite. Concrete syntax is the same as for Requirement except that it has mandatory requirement type indicator – shown as a text surrounded by double angle brackets: "«functional on composite»".

FunctionalRequirementOnSystem. Concrete syntax is the same as for Requirement except that it has mandatory requirement type indicator – shown as a text surrounded by double angle brackets: "«functional on system»".

UseCase. Concrete syntax is an extension of concrete syntax for UML :: UseCases :: UseCase, as described in UML Superstructure ([Obj05b], paragraph 16.3.6, page 579). "A use case is shown as an ellipse, either containing the name of the use case or with the name of the use case placed below the ellipse." As for any Requirement, every UseCase icon can present the 'ID' (see concrete syntax for Requirement). Additionally, UseCase can be presented with a minimised icon on a tree structure. See Figures 11.7, 11.8 for illustration of the above on a Use Case Diagram and Project Tree structure, respectively. See also 11.16 for example of usage of UseCases on Use Case Diagrams.

Figure 11.7: UseCase example

RequirementsPackage. Concrete syntax is almost the same as for UML :: Kernel :: Package, described in UML Superstructure (in [Obj05b], paragraph 7.3.37, page 104): "A package is shown as a large rectangle with a small rectangle (a 'tab') attached to the left side of the top of the large rectangle. The members of the package may be shown within the large rectangle. Members may also be shown by branching lines to member elements, drawn outside the package. A plus sign (+) within a circle is drawn at the end attached to the namespace (package). If the members of the package are not shown within the large rectangle, then the name of the package should be placed within the large rectangle. If the members of the package are shown within the large rectangle, then the name of the package should be placed within the tab. In addition to the above Kernel :: Package description, RequirementsPackage has two vertical lines to the left of the "rectangle with a tab" icon. It can also be presented in a tree structure. See Figures 11.9, 11.10 for examples of concrete syntax in a Package Diagram and in a Project Tree structure, respectively.

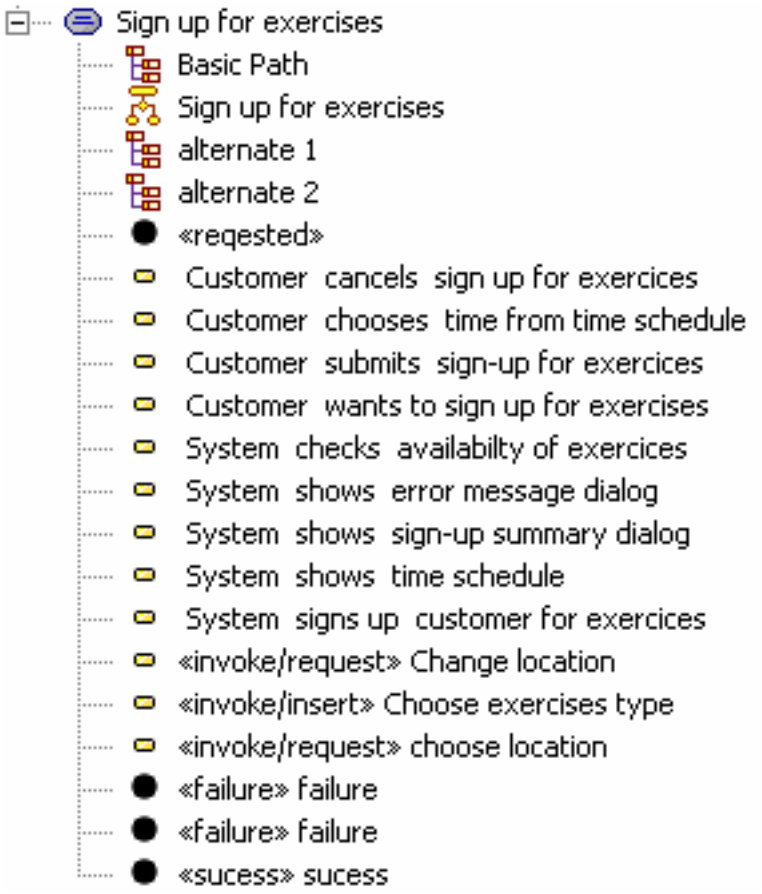


Figure 11.8: UseCase tree example

RequirementsSpecification. Concrete syntax is almost the same as for UML :: Kernel :: Package, described in UML Superstructure (in [Obj05b]); for this description see concrete syntax for RequirementsPackage. In addition to concrete syntax for plain Packages, RequirementsSpecification has one thick vertical line on its left. It can also be presented in a Project Tree structure with a minimised icon. See Figures 11.11, 11.12 for illustration of RequirementSpecification icon on a Package Diagram and in a Project Tree.

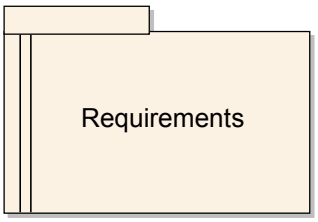


Figure 11.9: RequirementsPackage example

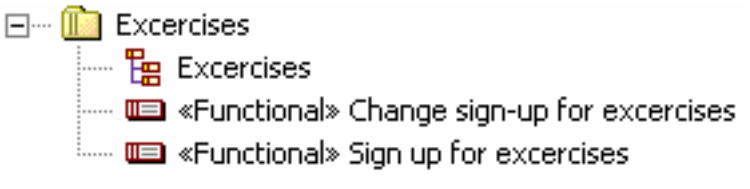


Figure 11.10: RequirementsPackage tree example

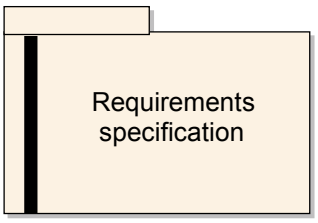


Figure 11.11: RequirementsSpecification example

11.3 Requirement relationships

11.3.1 Overview

This package describes relationships of different types between Requirements. Standard relationships, as specified in the conceptual model are defined here. Moreover, a relationship between a Requirement and Notions found in the domain specification are made available.

In general, relationships between Requirements are presented on Requirements Diagrams as dashed arrows with appropriate relationship type expressed through its name in angle brackets.

11.3.2 Abstract syntax and semantics

Abstract syntax for the RequirementRelationships package is described in Figure 11.13.

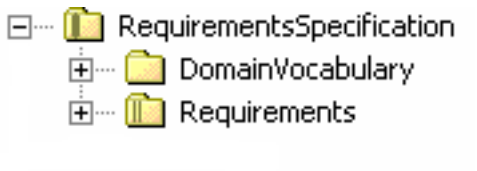


Figure 11.12: RequirementsSpecification tree example

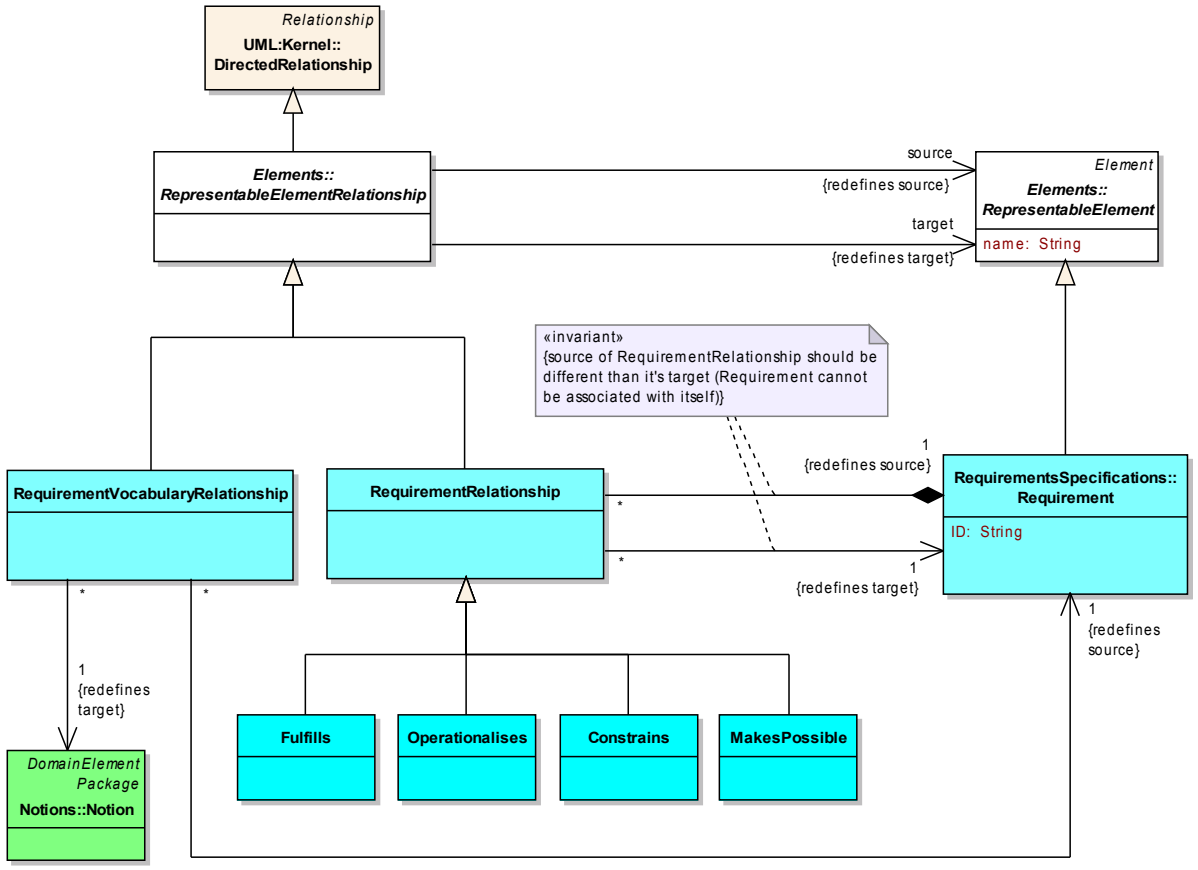


Figure 11.13: Requirement relationships

RequirementRelationship

Semantics. RequirementRelationship denotes a relationship between two requirements. The type of a relationship (e.g. similarity, conflict) is specified by a stereotype defined in an appropriate Profile.

Abstract syntax. RequirementRelationship is a kind of Elements :: RepresentableElementRelationship. RequirementRelationship is a component of RequirementsSpecifications :: Requirement (source of the relationship) and it points to another RequirementsSpecifications :: Requirement (target of the relationship). Source of the relationship should be different than its target – RequirementsSpecifications :: Requirement cannot be associated with itself. RequirementRelationship is the base meta-class for Constrains, Fulfills, MakesPossible and Operationalizes which precisely define types of relationships between Requirements.

Constrains

Semantics. Constrains is a type of RequirementRelationship. The relationship of this type denotes that one requirement (Constraint Requirement) impose a constraint on another requirement (Functional Requirement).
Abstract syntax. Constrains is a specialisation of RequirementRelationship. It derives whole abstract syntax from its superclass.

Fulfills

Semantics. Fulfills is a type of RequirementRelationship. The relationship of this type denotes that one requirement (Envisioned Scenario) is responsible for fulfillment of the responsibility captured in another requirement (Functional Requirement on Composite System).
Abstract syntax. Fulfills is a specialisation of RequirementRelationship. It derives whole abstract syntax from its superclass.

MakesPossible

Semantics. MakesPossible is a type of RequirementRelationship. The relationship of this type denotes that one requirement (Functional Requirement on System to be built) is made feasible, i.e., partially or fully fulfilled, by another requirement (Envisioned Scenario).
Abstract syntax. MakesPossible is a specialisation of RequirementRelationship. It derives whole abstract syntax from its superclass.

Operationalizes

Semantics. Operationalizes is a type of RequirementRelationship. The relationship of this type denotes that one requirement (Functional Requirement) represents functionality required to make the responsibility captured in another requirement (Constraint Requirement) possible.
Abstract syntax. Operationalizes is a specialisation of RequirementRelationship. It derives whole abstract syntax from its superclass.

RequirementVocabularyRelationship

Semantics. RequirementVocabularyRelationship denotes a relationship between a requirement and a notion from the domain specification. This means that related notion is applicable to the realisation of the requirement.
Abstract syntax. RequirementVocabularyRelationship is a kind of Elements :: RepresentableElementRelationship. It redefines source with RequirementsSpecifications :: Requirement and target with Notions :: Notion. RequirementVocabularyRelationship can have exactly one source RequirementsSpecifications :: Requirement and one target Notions :: Notion.

11.3.3 Concrete syntax and examples

RequirementRelationship is drawn as a dashed line connecting two RequirementsSpecifications :: Requirements. An open arrowhead may be drawn on the end of the line indicating the target of the relationship. The line is labeled with an appropriate stereotype determining the type of a relationship. The line may consist of many orthogonal or oblique segments.

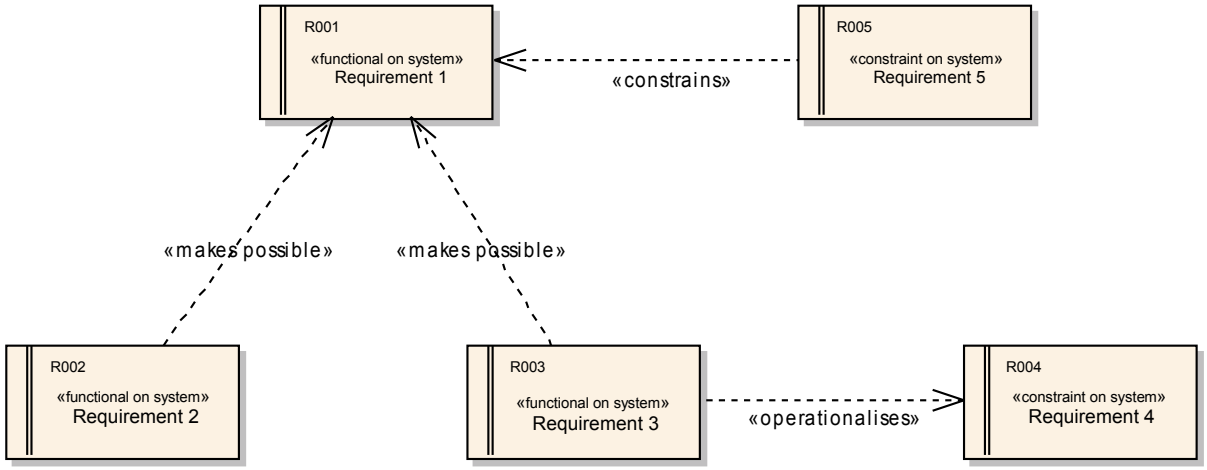


Figure 11.14: Requirements and requirement relationships concrete syntax example

Constrains. Concrete syntax is the same as for RequirementRelationship. It is labeled with appropriate text indicating the type of the relationship surrounded by double angle brackets: “«constrains»”.

Fulfills. Concrete syntax is the same as for RequirementRelationship. It is labeled with appropriate text indicating the type of the relationship surrounded by double angle brackets: “«fulfills»”.

MakesPossible. Concrete syntax is the same as for RequirementRelationship. It is labeled with appropriate text indicating the type of the relationship surrounded by double angle brackets: "«makes possible»".

Operationalizes. Concrete syntax is the same as for RequirementRelationship. It is labeled with appropriate text indicating the type of the relationship surrounded by double angle brackets: "«operationalizes»".

RequirementVocabularyRelationship is drawn as a dashed line connecting a RequirementsSpecifications :: Requirement and a Notions :: Notion. An open arrowhead is drawn on the end of the line indicating Notions :: Notion – the target of the relationship. The line may consist of many orthogonal or oblique segments.

11.4 Use case relationships

11.4.1 Overview

This package describes relations between UseCases and Classifiers (mainly Actors) or between two UseCases. The UseCaseRelationships package redefines parts of the UseCases package from the current UML specification [Obj05b].

Relationships between UseCases are generally reduced to «invoke» relationships denoted as dashed arrows. Generalisations between UseCases are also possible, as derived from UML. Actors can be related to UseCases with appropriate solid arrows that denote usage and participation of an actor in a use case.

11.4.2 Abstract syntax and semantics

Abstract syntax for the UseCaseRelationships package is described in Figure 11.15.

InvocationRelationship

Semantics. InvocationRelationship substitutes «include» and «extend» relationships from UML ([Obj05b], p.570) and unifies their disadvantageous semantics ([Sim99], [MOW01]). Invoca-

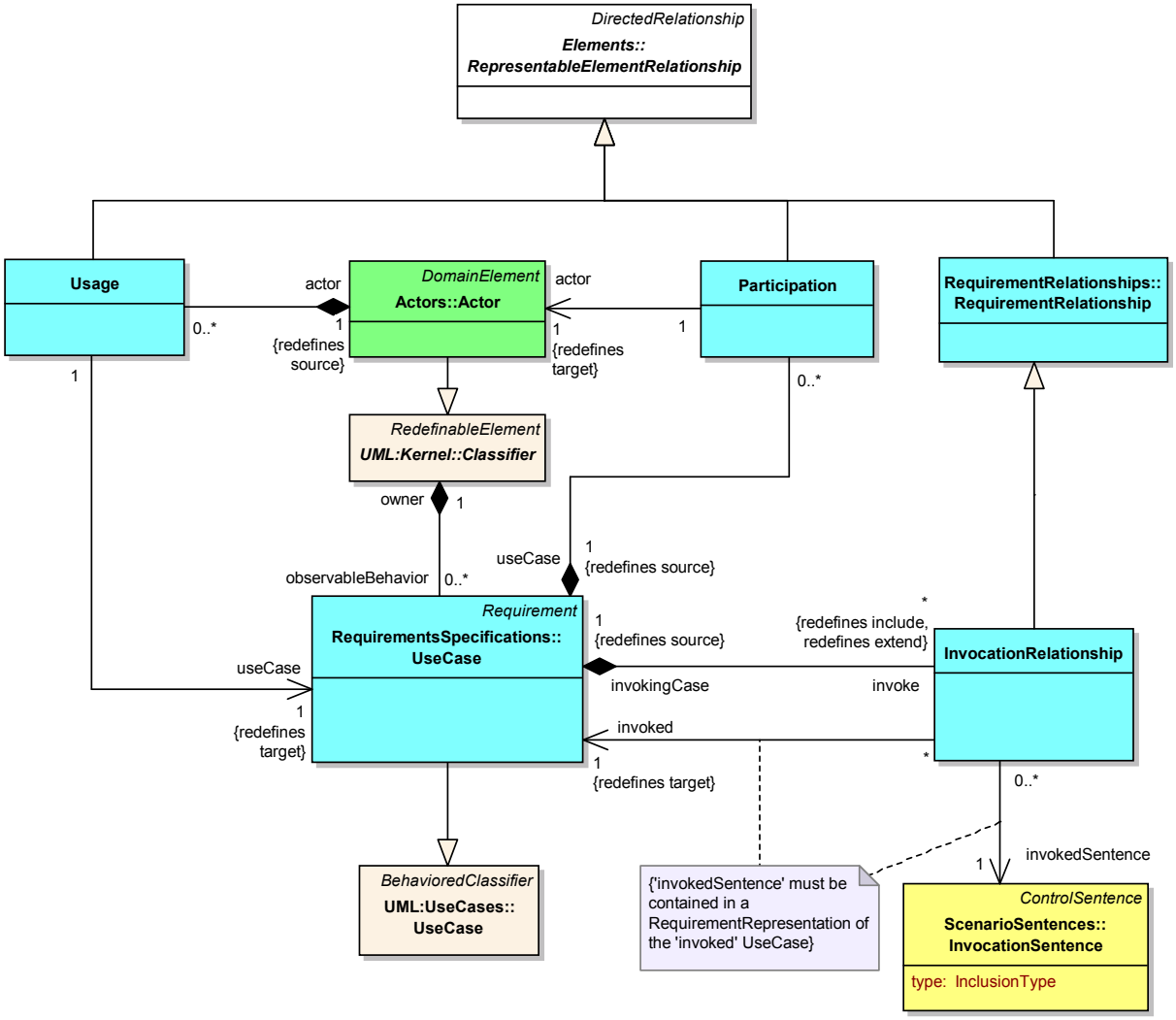


Figure 11.15: Use case relationships

tionRelationship denotes that another use case (actually, one of its scenarios) can be invoked from within currently performed use case. After performing one of the final actions in the invoked use case, the flow of control returns to the invoking use case right after the point of invocation to perform the remaining part of the base use case. There are two types of invocation: a use case can be invoked conditionally – only when requested by an actor, or unconditionally – every time the appropriate scenario of the base use case is performed. The type of the invocation, the name of a use case to be invoked and the exact point of invocation in the invoking use case scenario is defined by a special kind of scenario sentence (see ScenarioSentences :: InvocationSentences in Chapter 13.4).

Abstract syntax. InvocationRelationship is a kind of RequirementRelationships :: RequirementRelationship. It redefines source and target elements of its superclass with RequirementsSpecifications :: UseCase. It is a part of an 'invoking' RequirementSpecifications :: UseCase pointing to an 'invoked' UseCase. It also points to an ScenarioSentences :: InvocationSentence, which

must be contained in a RequirementRepresentations :: RequirementRepresentation of the 'invoked' UseCase.

Usage

Semantics. Usage indicates possibility for an Actors :: Actor to initiate a particular UseCase performance directly as a primary actor (the one that expects to reach the use case's goal).
Abstract syntax. Usage is a kind of RequirementRelationships :: RequirementRelationship. It is a component of an Actor and points to a RequirementSpecifications :: UseCase.

Participation

Semantics. Participation indicates possibility for an Actor to participate as a secondary actor in the execution of a particular UseCase.
Abstract syntax. Participation is a kind of RequirementRelationships :: RequirementRelationship. It is a component of a RequirementSpecifications :: UseCase and points to an Actor.

11.4.3 Concrete syntax and examples

InvocationRelationship. It can be shown similarly to a UML Dependency relationship between RequirementSpecifications :: UseCases with an «invoke» stereotype and an open arrowhead denoting navigability on the end of the 'invoked' RequirementSpecifications :: UseCase (see Figure 11.16).

Usage. It's concrete syntax is a solid line between an Actor and a RequirementSpecifications :: UseCase and an arrowhead on the side of the UseCase (see Figure 11.16). This arrow can be appended with a «use» UML-like stereotype.

Participation's concrete syntax is a solid line between a RequirementSpecifications :: UseCase and an Actors :: Actor and an arrowhead on the side of the Actor (see Figure 11.16). This arrow can be appended with a «participate» UML-like stereotype.

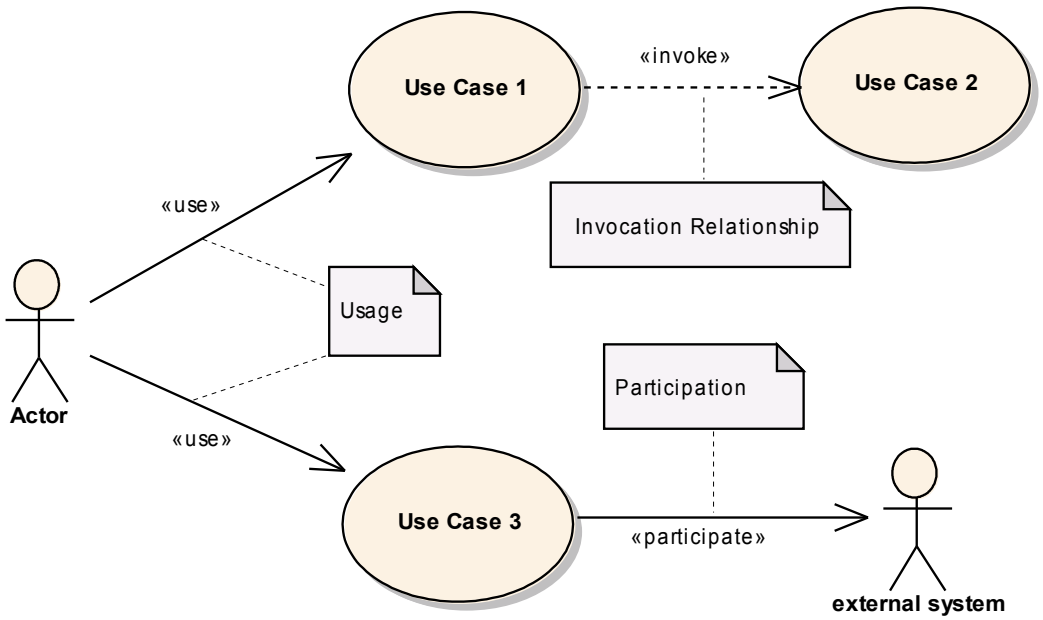


Figure 11.16: Use case relationships concrete syntax example

Chapter 12

Requirement representations

12.1 Overview

In the RequirementRepresentations part we describe how our language defines requirement representations and present differences between various requirement representations. As stated in Part I, requirements can have descriptive representations (natural or constrained language) or schematic representations (model-based). Language users will typically use elements from this part to describe the contents of individual requirements using the notations chosen from those available in the language. Requirement representations are tightly bound to the appropriate requirements they represent. Particular representation depends highly on the requirement type, as specified in the Requirements part.

The specification in this part contains five packages, as shown in Figures 12.1 and 12.2 (marked in blue on colour print-outs, excluding RequirementSpecifications belonging to the previous part).

- The RequirementRepresentations package contains all the general constructs needed to polymorphically represent differing requirement representations. It «import»s from the RequirementsSpecifications package to relate representations with appropriate requirements defined there. It also «import»s the Elements package, for RequirementRepresentation is a subtype of ElementRepresentation. As such the RequirementRepresentations package gives access to the RequirementsSpecifications package. The RequirementRepresentations package describes a general way of representing Requirements: every RequirementRepresentation is a component for its Requirement.

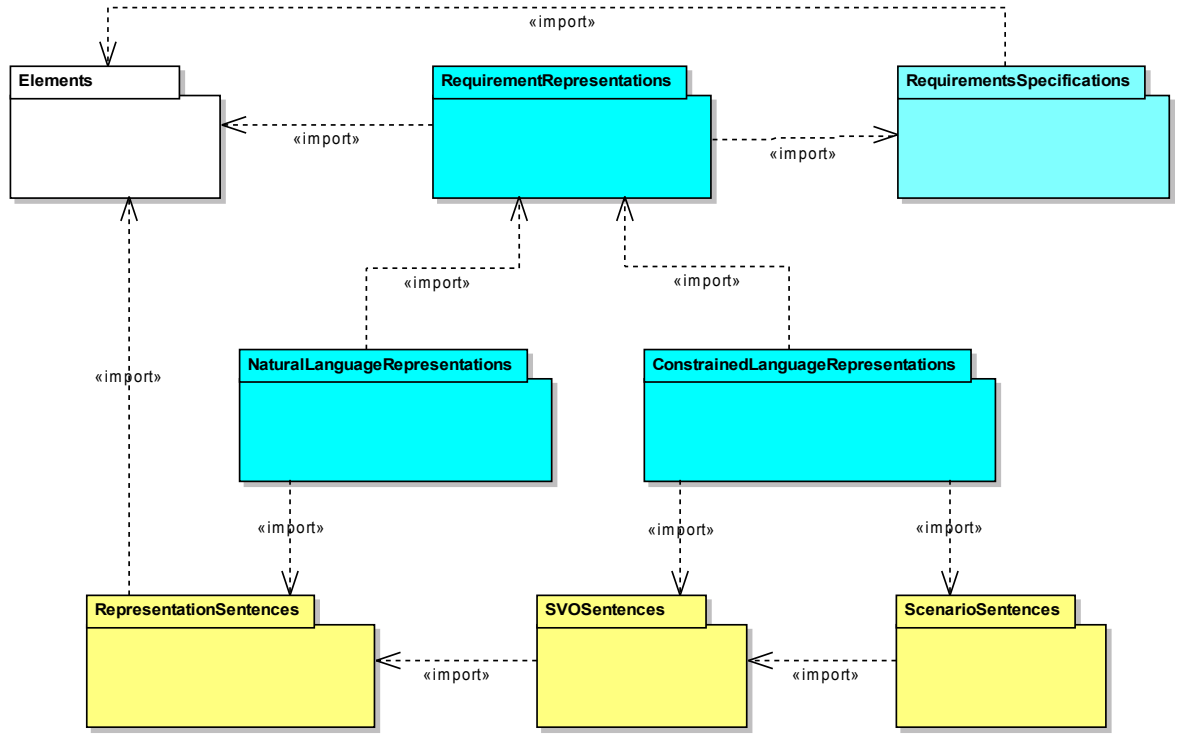


Figure 12.1: Overview of packages inside the RequirementRepresentations part of RSL (generic representations, natural language and constrained language)

- The NaturalLanguageRepresentations package contains all the constructs needed to express requirements in natural language. It «import»s from the RequirementRepresentations package. In this manner, natural language representations are specialisations of classes in RequirementRepresentations. As such the NaturalLanguageRepresentations package gives access to the RequirementRepresentations package and the RequirementsSpecifications package for any variant of natural language requirement representation being utilised within a requirements specification. This package also «import»s from RepresentationSentences where natural language hypertext sentences (which constitute natural language representations) are defined.

- The ConstrainedLanguageRepresentations package contains all the constructs needed to express requirements in constrained language. It «import»s from the RequirementRepresentations package. In this manner constrained language representations can be specified as being specialisations of elements in RequirementRepresentations. As such the ConstrainedLanguageRepresentations package gives access to the RequirementRepresentations package and the RequirementsSpecifications package for any variant of constrained language requirement representation being utilised within a requirements specification. The ConstrainedLanguageRepresentations package also «import»s from ScenarioSen-

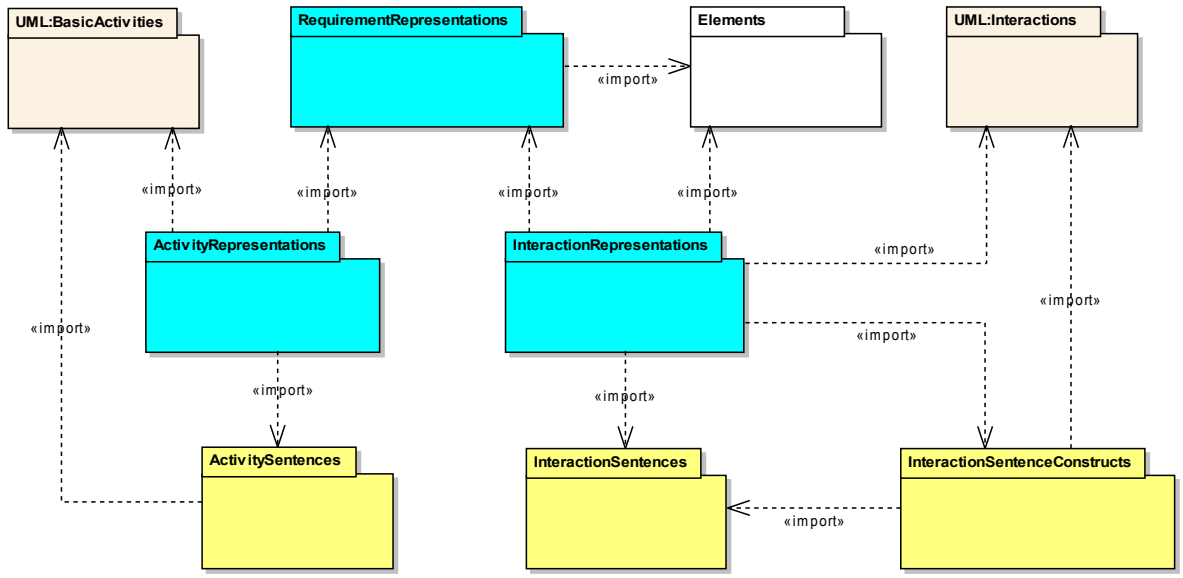


Figure 12.2: Overview of packages inside the RequirementRepresentations part of RSL (activities and interactions)

tences and SVOSentences packages where the sentences used in this representation are defined.

- The ActivityRepresentations package contains all the constructs needed to express requirements with diagrams that specialise from UML activity diagrams. It «import»s from the RequirementRepresentations package and the UML :: BasicActivities package. In this manner activity requirement representations are specialisations of UML Activity and elements from RequirementRepresentations. As such, the ActivityRepresentations package gives access to the RequirementRepresentations package, the RequirementsSpecifications package, and the BasicActivities package for any variant of activity diagram based requirement representation being utilised within a requirements specification. This package also «import»s from the ActivitySentences package which contains special kind of sentences that can be used in activity representations.

- The InteractionRepresentations package contains all the constructs needed to represent UML 2.0 interaction diagram based requirement representations. It «import»s from the RequirementRepresentations package and the UML :: Interactions package. In this manner interaction requirement representations are specialisations of UML Interaction and elements from the RequirementRepresentations package. As such the InteractionRepresentations package gives access to the RequirementRepresentations package, the RequirementsSpecifications package, and the Interactions package for any variant of interaction diagram based requirement representation being utilised within a requirements specification. This package also «import»s from the InteractionSentences package and Interac-

tionSentenceConstructs which contain special kind of sentences and other constructs that can be used in interaction representations.

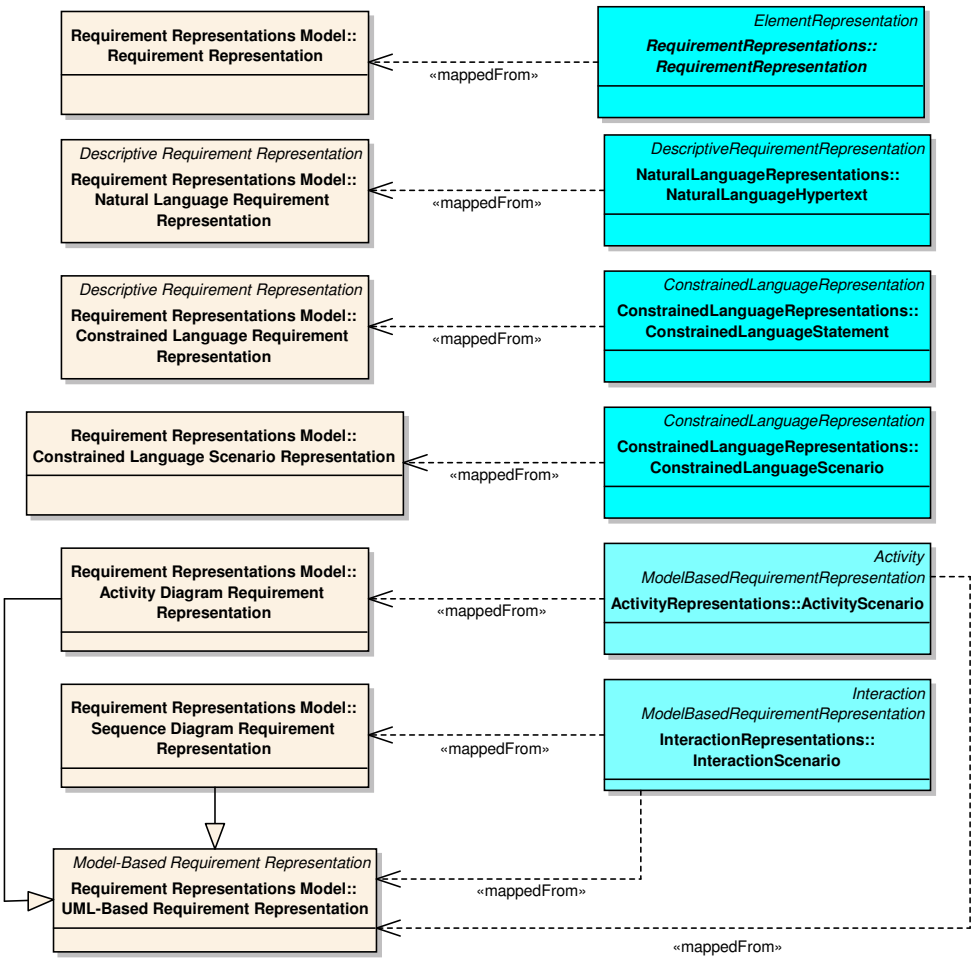


Figure 12.3: Main classes in the RequirementRepresentations part with mappings to the conceptual model

Individual classes in the above packages can be mapped from the conceptual model described in Chapter 5. These mappings are shown in Figure 12.3. The most general class in this part is the RequirementRepresentation class. This class allows for expressing details of requirement representations. Its source is the RequirementsModel :: RequirementRepresentation class from the conceptual model. Different specialisations of RequirementRepresentation also trace from relevant classes of the conceptual model, and particularly, the representation hierarchy as shown in Figure 5.1.

Representations include diagrams as well as text. Diagrams include Activity Diagrams where ActivityRepresentations can be shown and Sequence Diagrams where InteractionRepresentations can be shown. Concrete syntax of these diagrams derives from the syntax of appropriate UML diagrams. Concrete syntax for textual representations is composed of “source” and

"view" syntax. The first variant allows to represent various elements of representation in purely textual way. The second variant uses also font variations (underlining, bolding, etc.).

12.2 Requirement representations

12.2.1 Overview

The RequirementRepresentations package contains the most general and abstract constructs of the representation language. On this structure, all the concrete representations are built. Generally, every RequirementRepresentation is part of an appropriate RequirementsSpecifications :: Requirement (see figure 12.4).

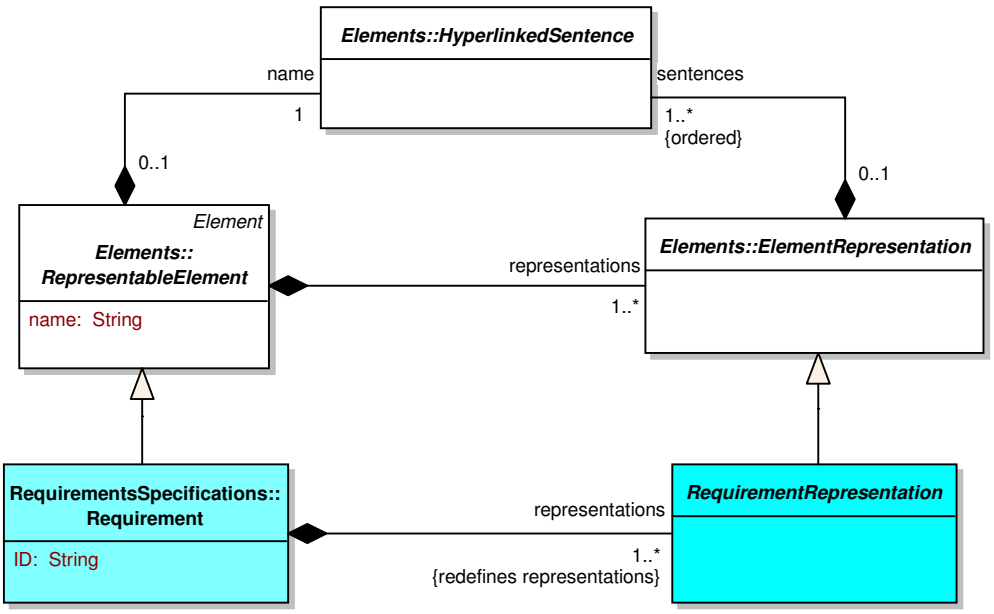


Figure 12.4: Requirement representation

Figure 12.5 shows a hierarchy of requirements representations that are allowed by the current language.

As one possible variant, requirements can be presented in textual form (DescriptiveRequirementRepresentation). NaturalLanguageRepresentations allow requirements to be represented as NaturalLanguageRepresentations :: NaturalLanguageHypertext. ConstrainedLanguageRepresentations allow requirements to be represented as either a ConstrainedLanguageRepresentations :: ConstrainedLanguageStatement or a ConstrainedLanguageRepresentations :: ConstrainedLanguageScenario.

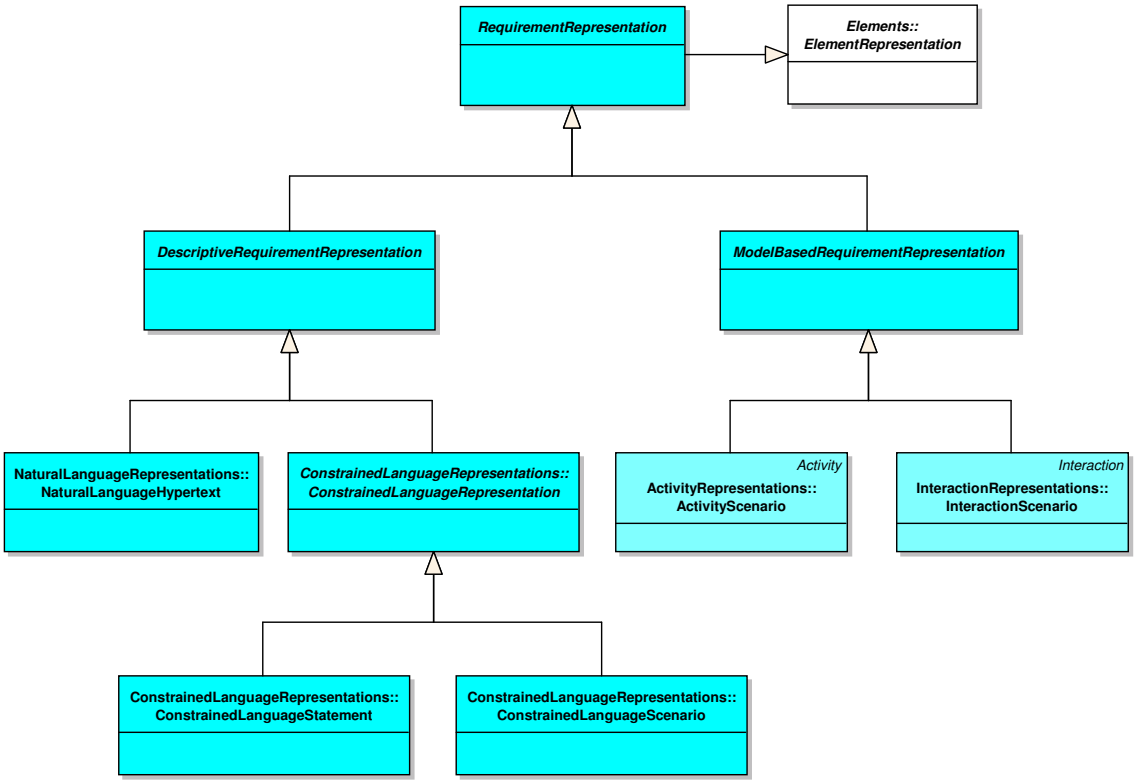


Figure 12.5: Requirement representations hierarchy

As a cecond possibility, requirements can also be presented in model-based form (ModelBasedRequirementRepresentation). ActivityRepresentations allow a requirement to be represented as an ActivityRepresentations :: ActivityScenario. InteractionRepresentations allow a requirement to be represented as an InteractionRepresentations :: InteractionScenario.

Furthermore, RequirementRepresentations introduces several meta-associations defining the possible representations for RequirementsSpecifications :: UseCases (see figure 12.6).

RequirementsSpecifications :: UseCase meta-class is a kind of RequirementsSpecifications :: Requirement and also inherits from UML's UseCase :: UseCase. It's content can be expressed through three different perspectives:

- ConstrainedLanguageRepresentations :: ConstrainedLanguageScenario – textual representation of UseCase's scenarios
- ActivityRepresentations :: ActivityScenario – adds graphical representation of control flow between different scenarios of a single UseCase
- InteractionRepresentations :: InteractionScenario – emphasises the aspect of interaction between a system and its users by showing a sequence of messages sent between them

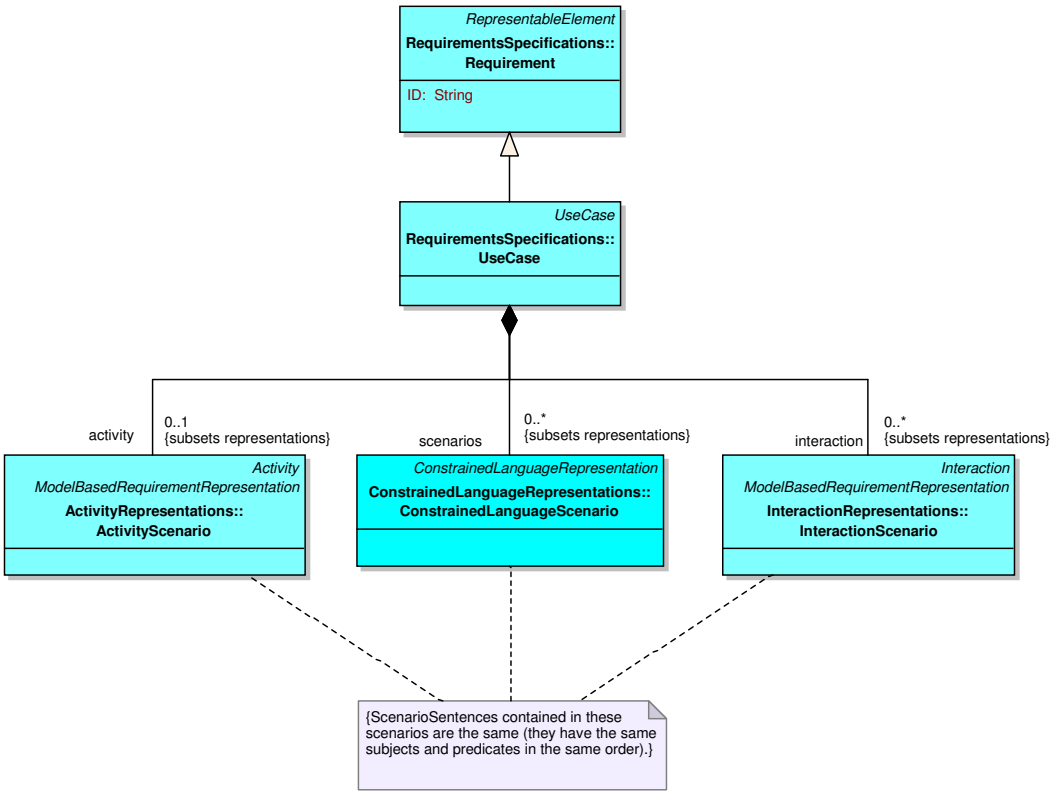


Figure 12.6: UseCase representations

It has to be stressed that the above three representations for the same UseCase should contain the same information. These representations show this information in three different aspects.

12.2.2 Abstract syntax and semantics

RequirementRepresentation

Semantics. Defines the content of a RequirementsSpecifications :: Requirement which should, according to IEEE definition, generally constitute a condition or capability needed by a user to solve a problem or achieve an objective. It also contains a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. This content depends on the concrete representation type that specialises RequirementRepresentation.

Abstract syntax. It is part of every RequirementsSpecifications :: Requirement and forms one of its 'representations'. It consists of one or more 'sentences' in the form of HyperlinkedSentences derived from ElementRepresentations :: ElementRepresentation. RequirementRepresentation is abstract and has several concrete specialisations.

DescriptiveRequirementRepresentation

Semantics. This meta-class allows for textual representation of requirements in the form of free or constrained text.
Abstract syntax. It is a kind of RequirementRepresentations :: RequirementRepresentation. DescriptiveRequirementRepresentation is an abstract class.

ModelBasedRequirementRepresentation

Semantics. This meta-class allows for representing requirements in schematic form.
Abstract syntax. It is a kind of RequirementRepresentations :: RequirementRepresentation. ModelBasedRequirementRepresentation is an abstract class.

Meta-associations between UseCase and its representations

Apart from the above meta-classes, this package defines several meta-associations that define relationship between UseCases and their representations.

Appropriate abstract syntax is presented in Figure 12.6. UseCase is a special kind of Requirement that can have its content represented by three RequirementRepresentations. Two of them are ModelBasedRequirementRepresentations and one of them is a DescriptiveRequirementRepresentation. All the three representations of the UseCase content (ConstrainedLanguageRepresentations :: ConstrainedLanguageScenario, ActivityRepresentations :: ActivityScenario and InteractionRepresentations :: InteractionScenario) are described in detail in sections 12.4, 12.5 and 12.6, respectively.

12.2.3 Concrete syntax and examples

RequirementRepresentation. As an abstract meta-class, this meta-model element has no concrete syntax. It can be formulated in any of representations of meta-classes that derive from it.

DescriptiveRequirementRepresentation. As an abstract meta-class, this meta-model element has no concrete syntax. However any of meta-classes that specialise from it may have capital letter “D” in their concrete syntaxes, indicating their descriptive character.

ModelBasedRequirementRepresentation. As an abstract meta-class, this meta-model element has no concrete syntax. However any of meta-classes that specialise from it may have capital letter “M” in their concrete syntaxes, indicating their model-based character.

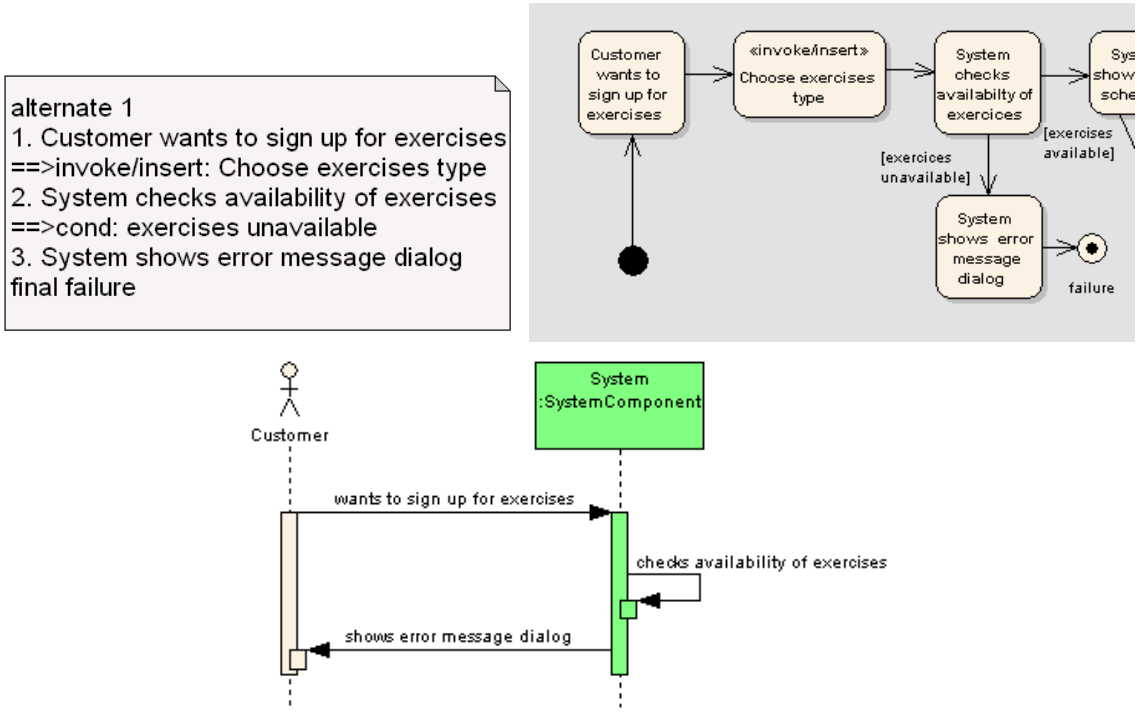


Figure 12.7: The same scenario in three different representations: ConstrainedLanguageScenario, ActivityScenario and InteractionScenario

UseCase representations. Figure 12.7 shows three different representations of content of a UseCase. This diagram compares alternative notations for the contents of UseCases. Details of concrete syntax for the three alternative notations is given in sections 12.4, 12.5 and 12.6.

12.3 Natural language representations

12.3.1 Overview

The NaturalLanguageRepresentations package describes ways to represent requirements in plain natural language without any formal structure. Sentences in natural language may contain

Hyperlinks to build a coherent connection to the domain knowledge contained in the domain specification.

12.3.2 Abstract syntax and semantics

The diagram in Figure 12.8 shows the abstract syntax of the NaturalLanguageRepresentations package. The following subsections will describe relationships between the classes in this diagram.

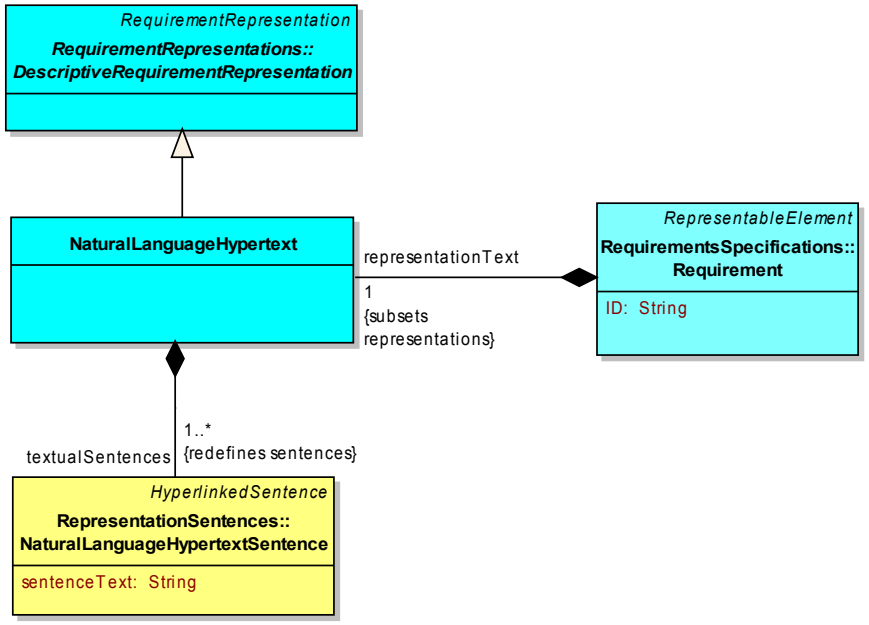


Figure 12.8: Natural language representations

NaturalLanguageHypertext

Semantics. A NaturalLanguageHypertext is the simplest possible representation of a single requirement. The text is written in natural language.
Abstract syntax. A NaturalLanguageHypertext is derived from RequirementRepresentations :: DescriptiveRequirementRepresentation. When it represents a Requirement, its role is 'representationText'. The text consists of one or more NaturalLanguageHypertextSentences.

Source:

```
Every [[customer]] may [[sign up for exercises]] at the [[terminals]] or
online over [[the Internet]]. After the registration, the [[customer]]
must [[recieve sign-up confirmation]].
```

View:

Every customer may sign up for exercises at the terminals or online over the Internet. After the registration, the customer must recieve sign-up confirmation.

Figure 12.9: NaturalLanguageHypertext example

12.3.3 Concrete syntax and examples

NaturalLanguageHypertext. Figure 12.9 shows an example for the concrete syntax of NaturalLanguageHypertext. The text is composed of several NaturalLanguageHypertextSentences which contain zero or more hyperlinks. The upper sentence shown in the example is the syntax as the requirements engineer will write it down, the lower sentence shows the presentation in the requirements document.

12.4 Constrained language representations

12.4.1 Overview

The ConstrainedLanguageRepresentations package allows for representing requirements by using constrained language, i.e. a subset of natural language whose sentences are limited to a certain structure. Refer to Chapter 13 and specifically to section 14.7 for details on this structure.

12.4.2 Abstract syntax and semantics

Figure 12.10 shows the three classes contained in the package ConstrainedLanguageRepresentations: ConstrainedLanguageRepresentation, ConstrainedLanguageStatement and ConstrainedLanguageScenario.

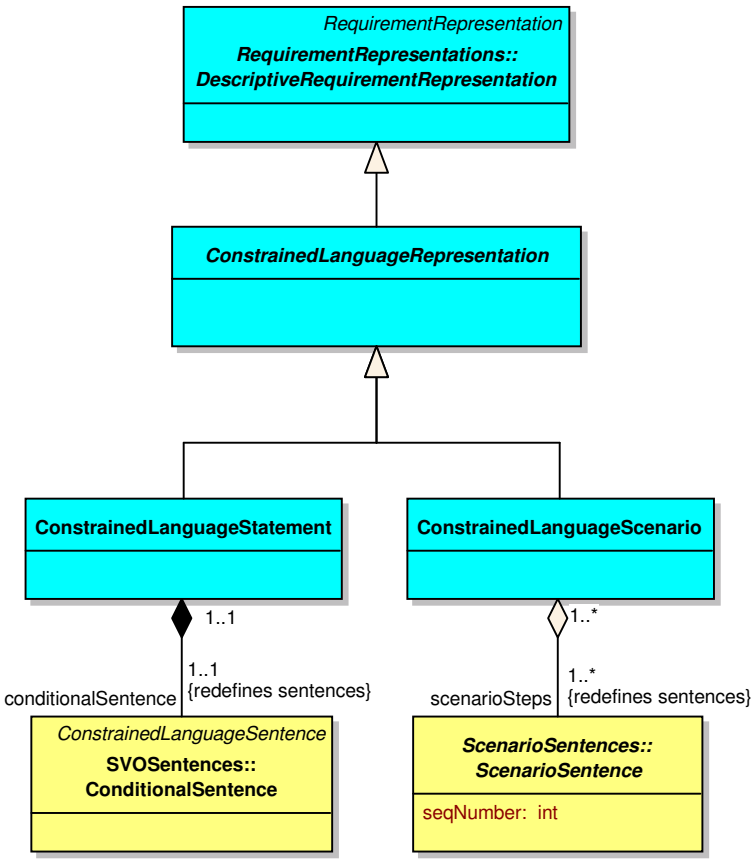


Figure 12.10: Constrained language representations

ConstrainedLanguageRepresentation

Semantics. ConstrainedLanguageRepresentation constitutes the description of a requirement by one or more sentences in a constrained language.
Abstract syntax. ConstrainedLanguageRepresentation is a kind of RequirementRepresentations :: DescriptiveRequirementRepresentation.

ConstrainedLanguageStatement

Semantics. This class represents a requirement by a single sentence in a constrained language. This sentence may be a conditional sentence consisting of a conditional clause and a main clause with a modal verb expressing the liability of the requirement. It is possible to omit the condition part of the sentence, thus forming a simple sentence which only comprises a main clause. The sentence may contain hyperlinks to phrases or terms in the vocabulary.
Abstract syntax. ConstrainedLanguageStatement is a kind of ConstrainedLanguageRepresen-

Source:

```
[[d:The n:Fitness Club]] a:should [[v:provide n:bracelets]].

c:If [[d:a n:customer]] [[v:signs up p:for d:a n:course]], [[d:the
n:system]] a:must [[v:bill d:this n:customer]].
```

View:

The : Fitness Club : should : provide : bracelets.

If : a : customer : signs up : for : a : course : , the : system : must : bill : this : customer.

Figure 12.11: Examples of ConstrainedLanguageStatements

tation. It is composed of exactly one single SVOSentences :: ConditionalSentence in the role of 'conditionalSentence'.

ConstrainedLanguageScenario

Semantics. ConstrainedLanguageScenario represents a requirement as a scenario and is part of a use case. This scenario consists of a sequence of sentences in constrained language constituting its individual steps.
Abstract syntax. ConstrainedLanguageScenario is a kind of ConstrainedLanguageRepresentation. It is composed of one or more ScenarioSentences :: ScenarioSentences taking the role of 'scenarioSteps'. ConstrainedLanguageScenario is part of a RequirementsSpecifications :: UseCase.

12.4.3 Concrete syntax and examples

ConstrainedLanguageRepresentation. As an abstract meta-class, ConstrainedLanguageRepresentation does not have a concrete syntax. It can be formulated in any of the representations of its subclasses.

ConstrainedLanguageStatement. Figure 12.11 shows two example of the concrete syntax of a ConstrainedLanguageStatement. The *Source* part shows the statement as it is entered by the requirements engineer. The words enclosed in double square brackets denote a hyperlink. The *View* part below depicts the statement's presentation in the requirements document. Hyperlinks appear coloured and underlined. The preceding letter with the colon denotes the part of speech (e.g. *n*oun, *v*erb). See section 14.7 for more details.

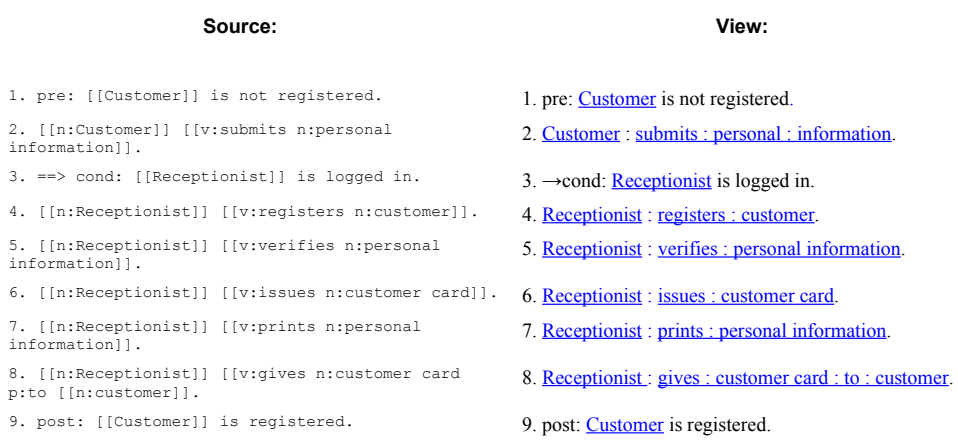


Figure 12.12: Example of a ConstrainedLanguageScenario

ConstrainedLanguageScenario. An example for a ConstrainedLanguageScenario can be taken from Figure 12.12. The left hand side, the *Source* side, displays the sequence of ScenarioSentences as it is entered by the requirements engineer. The words enclosed in double square brackets denote a hyperlink. On the right hand side, the result in the requirements document is shown. Hyperlinks appear coloured and underlined. The preceding letter with the colon denotes the part of speech (e.g. *n*oun, *v*erb). See section 14.7 for more details.

The above example also includes ScenarioSentences :: ControlSentences (see lines one and nine) and ScenarioSentence :: ConditionSentence (line three). They are described in more detail in section 13.4.

12.5 Activity representations

12.5.1 Overview

Activity representations package describes ActivityScenario as an alternative way of representing UseCase scenarios. Such representation emphasises flow of control between scenarios within a UseCase in the form of a UML :: Activity.

12.5.2 Abstract syntax and semantics

Abstract syntax for this package is shown in 12.13.

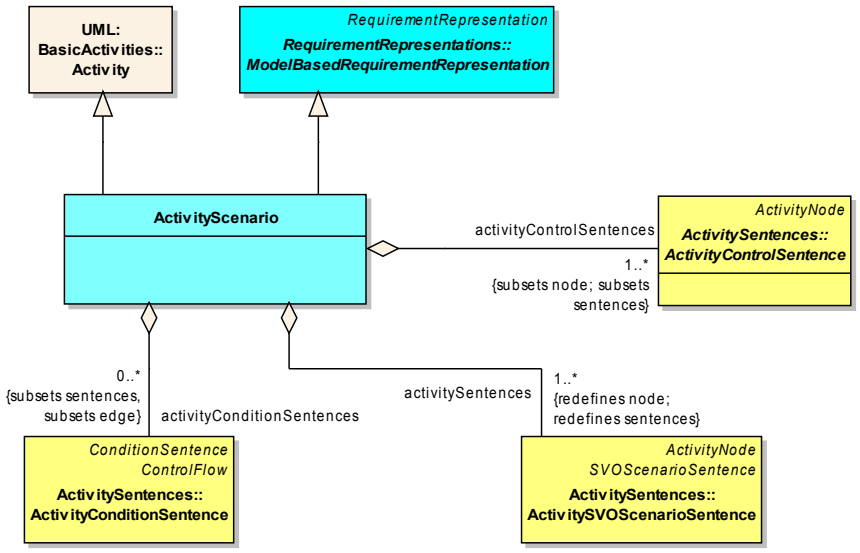


Figure 12.13: Activity representations

ActivityRepresentation

Semantics. An ActivityScenario is an alternative to ConstrainedLanguageScenario as a way of representing a UseCase's content. In this representation, UseCase scenarios are represented in the form activities. Beside showing the sequence of ScenarioSentence (a scenario), it also represents in a graphical way the flow of control between different scenarios within one UseCase.
Abstract syntax. An ActivityScenario is a kind of RequirementsRepresentations :: ModelBasedRequirementRepresentation. It also specialises BasicActivities :: Activity out of the UML2.0 superstructure. ActivityScenario contains zero or more AcivityConditionSentences's which subsets 'edge' from Activity superclass ,one or more ActivityControlSentences which subset 'node' and one or more ActivitySVOScenarioSentence. These tree classes redefine and subset sentences from the BasicRepresentations :: RequirementRepresentation superclass.

12.5.3 Concrete syntax and examples

ActivityScenario. Figure 12.14 shows an example of the concrete syntax of an ActivityScenario. The notation for an activity is a combination of the notations of the nodes and edges it contains (just like the notation of UML's BasicActivities :: Activity). For more details please refer to sections 13.5 and 13.6.

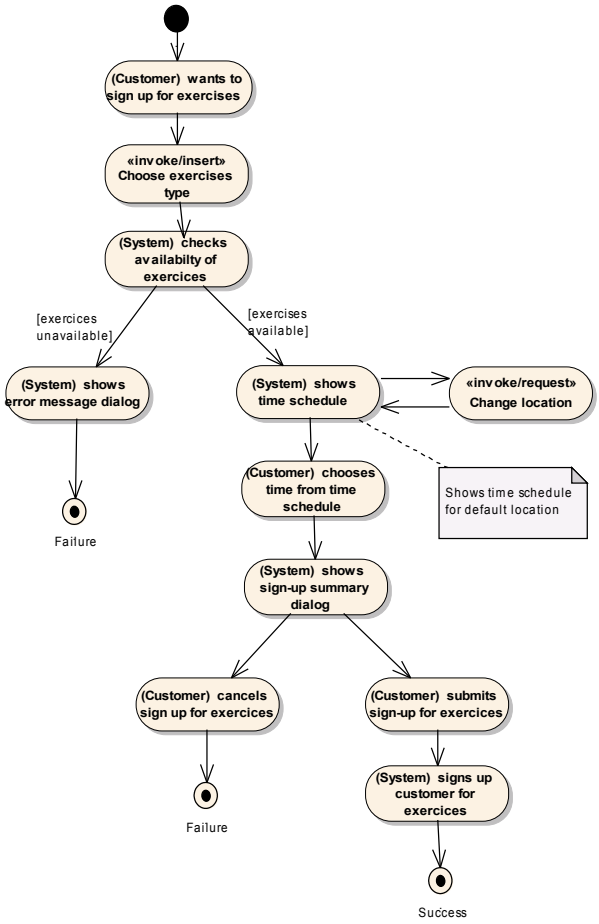


Figure 12.14: ActivityScenario example

12.6 Interaction representations

12.6.1 Overview

In addition to natural and constrained language descriptions and ActivityRepresentations, the package InteractionRepresentations provides another way to model scenarios. The main meta-class of this package is InteractionScenario, and allows for expressing scenarios in a notation based on UML interaction diagrams.

12.6.2 Abstract syntax and semantics

The diagram in figure 12.15 illustrates the abstract syntax of the interaction diagrams that can be used to describe requirements in the RSL. The following subsections explain the consecutive classes displayed in this diagram.

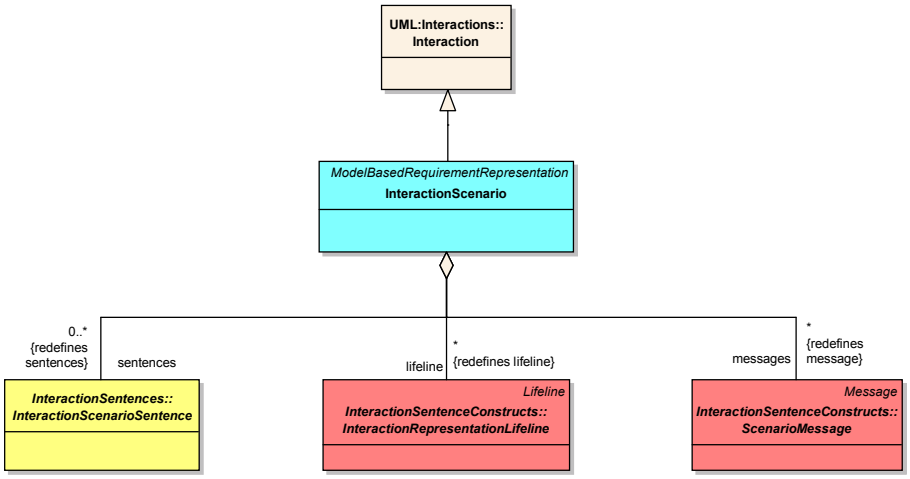


Figure 12.15: Interaction representation

InteractionScenario

Semantics. An InteractionScenario is one possible way to describe a scenario in a UseCase that constitutes a requirement. It contains lifelines and messages between these lifelines. The lifelines and messages build up InteractionScenarioSentences, where the lifelines constitute subjects and objects of these sentences and the messages are predicates.
Abstract syntax. The base classes of InteractionScenario are the classes Interaction from the UML :: Interactions package and ModelBasedRequirementRepresentation from RequirementRepresentations. While a general Interaction may contain Lifelines, an InteractionScenario may contain only InteractionRepresentationLifelines. For detailed information about the different messages and lifelines refer to sections 13.7 and 13.8.

12.6.3 Concrete syntax and examples

The Figures 12.16 and 12.17 describe the concrete syntax of the interaction diagram that can be used to describe requirements in the RSL. The first Figure shows a sequence diagram as one possible form of interaction diagram, the second Figure shows the other possible form – the communication diagram.

InteractionScenario. Both diagrams shown in the Figures 12.16 and 12.17 show the same InteractionScenario. Concrete syntax of specific elements of InteractionScenario are described in sections 13.7 and 13.8.

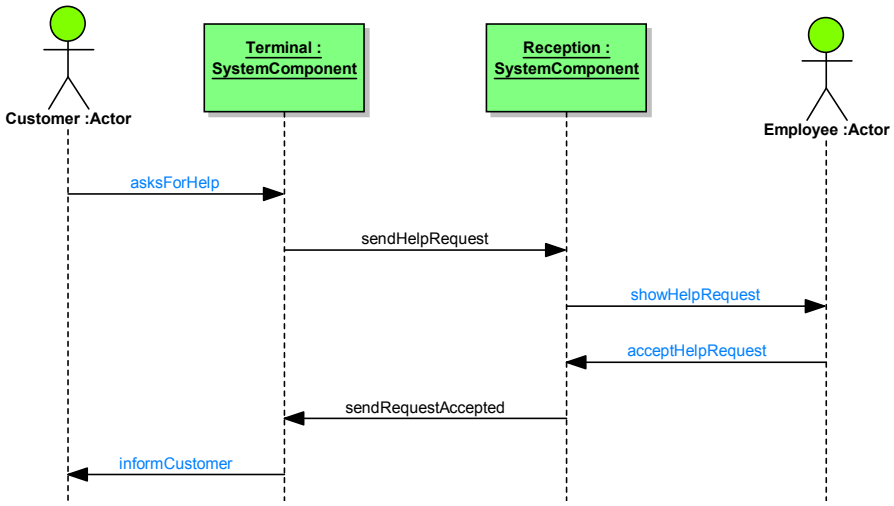


Figure 12.16: Interaction representation with sequence diagram

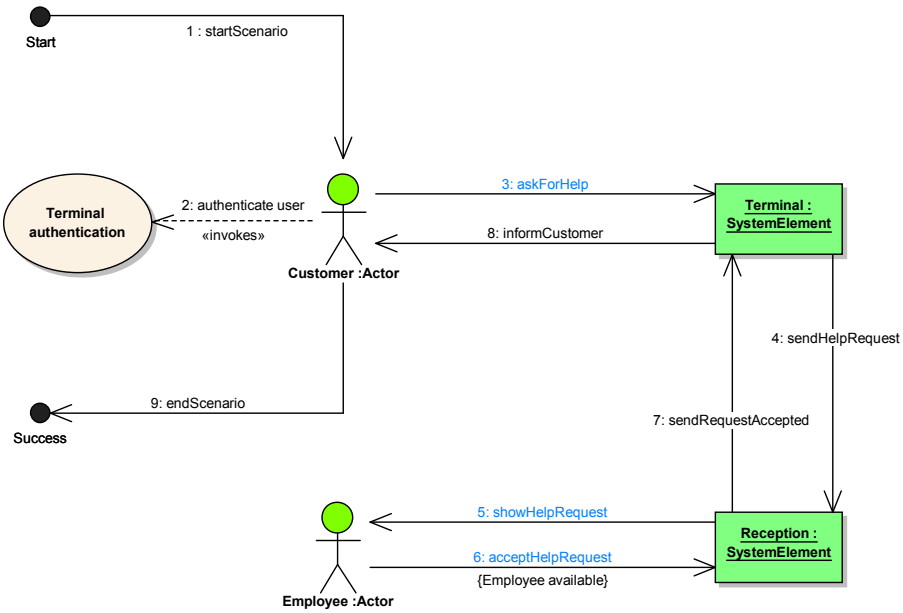


Figure 12.17: Interaction representation with communication diagram

Chapter 13

Requirement representation sentences

13.1 Overview

This chapter covers the different types of sentences in constrained language which can be used to represent requirements. Figures 13.1 and 13.2 give an overview of the seven packages (marked in yellow on colour print-outs) inside this part of the Requirements Specification Language.

- The RepresentationSentences package contains classes representing single sentences in natural and constrained language. The class representing a single sentence in constrained language is abstract and it has its concrete specialisations in the other two packages described below. RepresentationSentences packages «import»s from Kernel :: Elements in order to reuse the syntax and semantics of more general elements.
- Inside the SVOSentences[1] package, there exist constructs representing concrete types of constrained language sentences and their breakdown into more fine-granular elements, such as Subject or Predicate. Therefore SVOSentences package «import»s from RepresentationSentences package, Phrases package and Terms package.
- The contents of the ScenarioSentences package which imports SVOSentences are used for describing sentences of scenarios. They differ from "ordinary" SVOSentences by containing a sequence number denoting their position inside the scenario. The subtypes of a ScenarioSentence allow for describing a single scenario step as well as for expressing

[1] *SVO* stands for *subject – verb – object*. The identifier refers to the SVO(O) (subject – verb – object – (object)) grammar used for the constrained language.

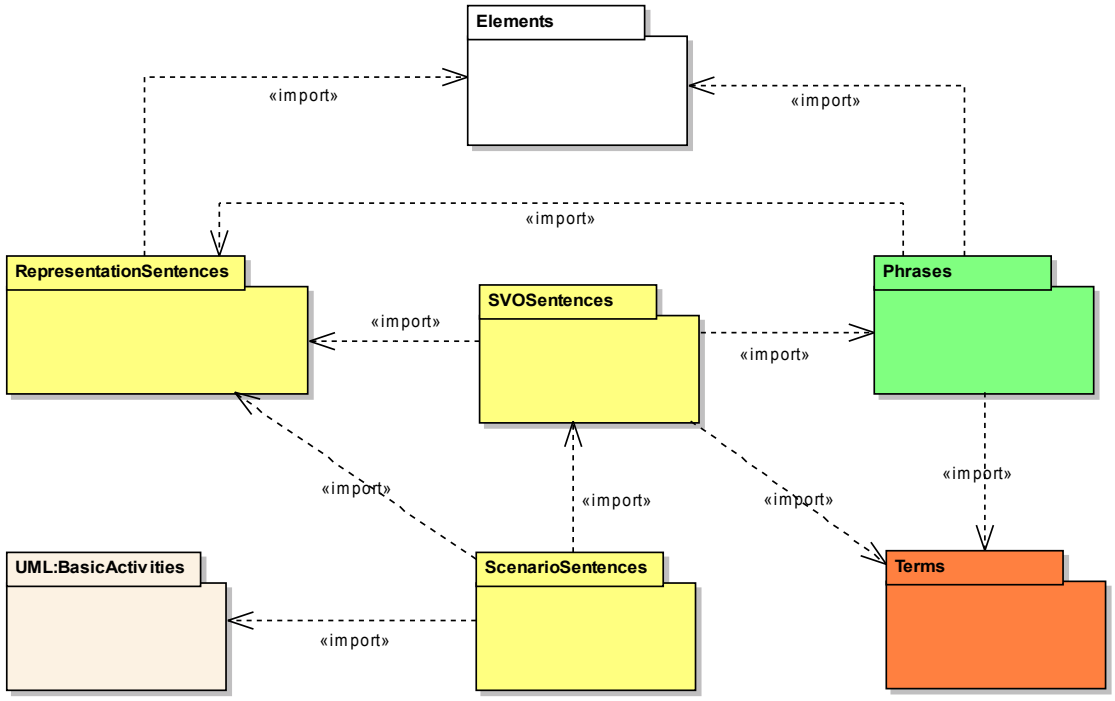


Figure 13.1: Overview of packages inside the RequirementRepresentationSentences part of RSL (representation, SVO and scenario sentences)

the control flow of a scenario's execution. This package «import»s from packages UML :: BasicActivities, RepresentationSentences and SVOSentences in order to specialise more general elements form these packages and reuse their syntax and semantics.

- The ActivitySentences package contains meta-classes that defines scenario sentences for requirement representations defined in ActivityRepresentations package. It «import»s from ScenarioSentences package and UML :: BasicActivities package in order to specialise more general elements contained there.

- The ActivitySentenceConstructs package contain definitions of additional constructs. These constructs are introduced to allow for representing scenario sentences in the form of activity representations.

- The InteractionSentences package contains meta-classes that specialise scenario sentences in order to use them in the representations defined in InteractionRepresentations. To extend more general constructs, this package «import»s from ScenarioSentences package.

- The InteractionSentenceConstructs package contains definitions of all constructs needed to represent scenarios in the form of interaction diagrams. Thus, this package «import»s from UML :: Interactions in order to reuse syntax and semantics of UML constructs. It

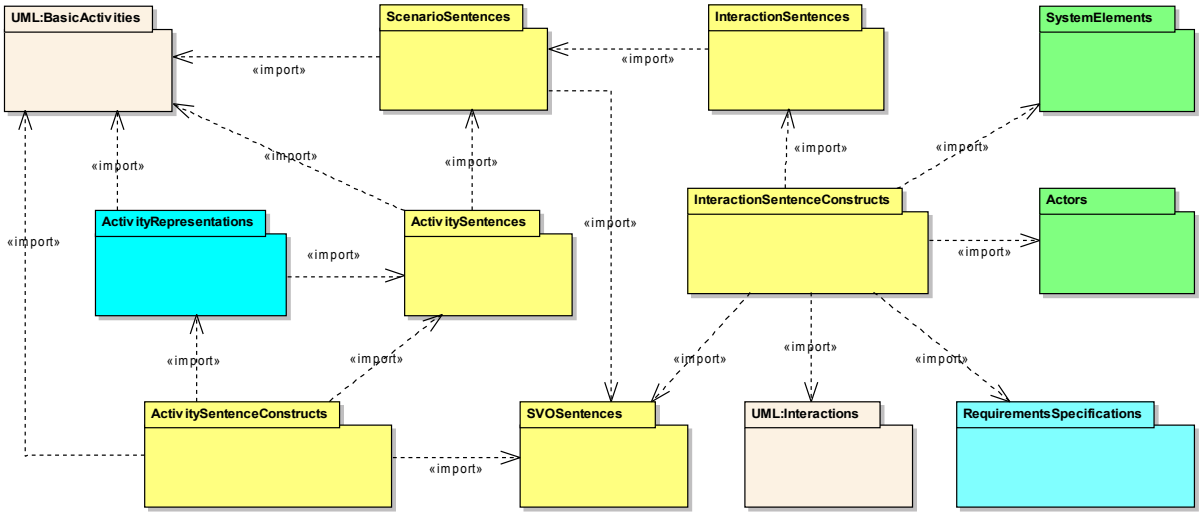


Figure 13.2: Overview of packages inside the RequirementRepresentationSentences part of RSL (activity and interaction sentences)

also «import»s from SystemElements and Actors packages to allow representing elements defined there in the interaction diagrams.

13.2 Representation sentences

13.2.1 Overview

This section introduces sentences written in constrained language which may be used to describe requirements.

13.2.2 Abstract syntax and semantics

Figure 13.3 shows the abstract syntax of the RepresentationSentences package. Specific metaclasses are described in the sections below.

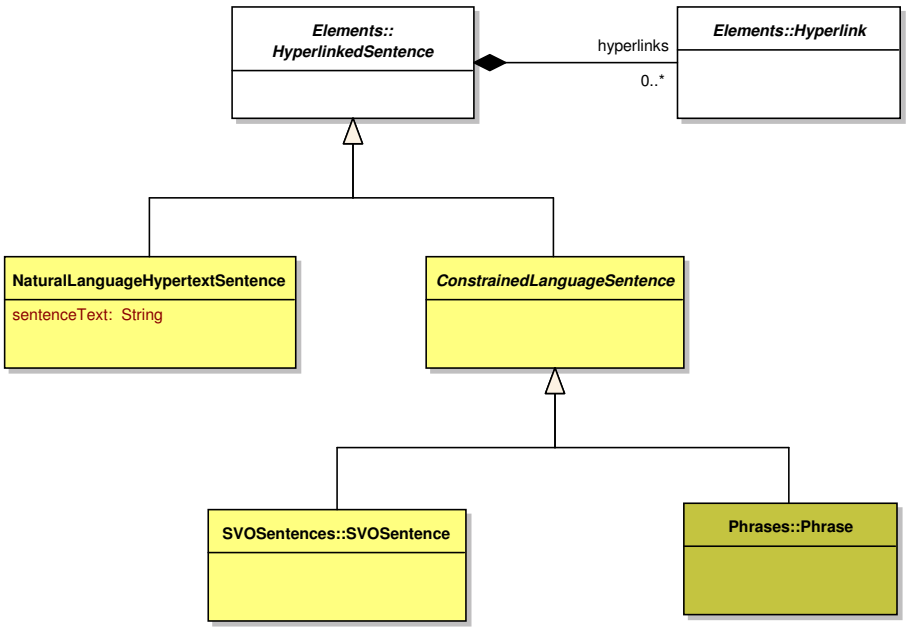


Figure 13.3: RepresentationSentences

NaturalLanguageHypertextSentence

Semantics. A NaturalLanguageHypertextSentence is used in a natural language description of a requirement. Using wiki-like hyperlinks in the sentence, a connection to the domain knowledge in the vocabulary is possible. If the sentence does not contain any Hyperlink, it is simply free text.

Abstract syntax. A NaturalLanguageHypertextSentence is part of a NaturalLanguageHypertext, its role is textualSentence. Since NaturalLanguageHypertextSentence is derived from HyperlinkedSentence, it may contain zero or more Hyperlinks. Each of those wiki-like hyperlinks links to a Term or Phrase in the vocabulary.

ConstrainedLanguageSentence

Semantics. Constrained language is a subset of natural language which is structured by some restrictions. Every type of constrained language sentence which is used in the RSL meta-model is a specialisation of this class. A more detailed explanation of the different kinds of constrained language sentences used in the RSL can be found in the sections below. In addition to its specific structure, the ConstrainedLanguageSentence contains zero or more Elements :: Hyperlinks.

Abstract syntax. The ConstrainedLanguageSentence is an abstract base class for all other sentences that use structured language. These are ConditionalSentence, SVOSentence (see section 13.3 for both) and Phrase, which are stored in the vocabulary. ConstrainedLanguageSentence itself is derived from Elements :: HypertextSentence, so it may contain Elements :: Hyperlinks.

13.2.3 Concrete syntax and examples

ConstrainedLanguageSentence. Since ConstrainedLanguageSentence is an abstract meta-class, there is no concrete syntax.

Source:

```
Every [[customer]] may [[sign up for exercises]] at the [[terminals]] or
online over [[the Internet]]. After the registration, the [[customer]]
must [[receive sign-up confirmation]].
```

View:

Every customer may sign up for exercises at the terminals or online over the Internet. After the registration, the customer must receive sign-up confirmation.

Figure 13.4: Example for NaturalLanguageHypertextSentence

NaturalLanguageHypertextSentence. Figure 13.4 shows an example for the concrete syntax of NaturalLanguageHypertextSentence as a part of NaturalLanguageHypertext. NaturalLanguage-HypertextSentence is natural language sentence, which contain zero or more hyperlinks. It can be shown in source and view form. In source form, hyperlinks are marked with double square brackets. In view form hyperlinks are shown as underlined, coloured text.

13.3 SVO sentences

13.3.1 Overview

This package describes the meta-model for three kinds of simple grammar sentences used for expressing individual sentences inside requirement representations.

13.3.2 Abstract syntax and semantics

Figure 13.5 shows the part of the RSL meta-model which deals with the content of the SVOSentences package.

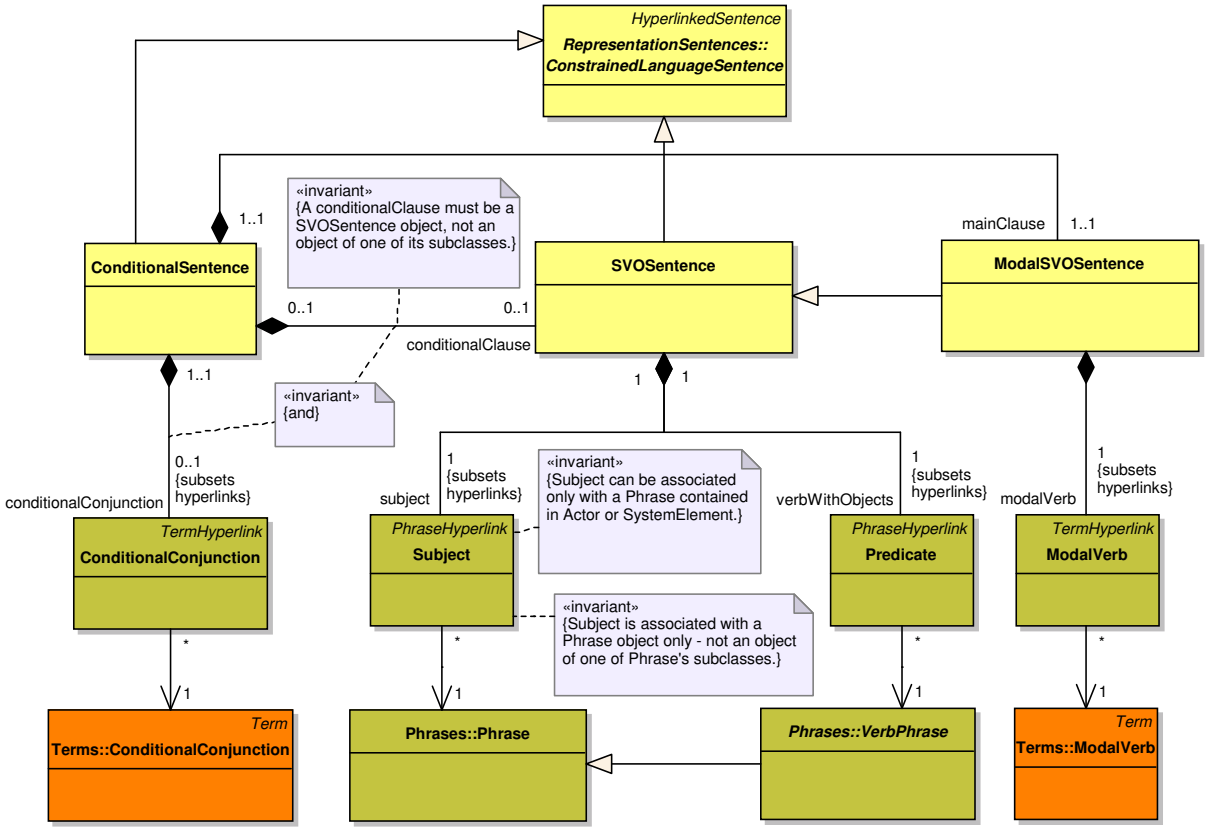


Figure 13.5: SVOSentences

SVOSentence

Semantics. Represents a sentence in a simple SVO(O) [2] grammar, where the VO(O) part is represented by a Predicate pointing to a Phrases :: VerbPhrase.

Abstract syntax. SVOSentence is a kind of RepresentationSentences :: ConstrainedLanguage-Sentence. It has one Subject and one Predicate (in the role of a 'verbWithObjects').

ConditionalSentence

Semantics. A ConditionalSentence is a sentence consisting of a condition part ('conditional-Clause') beginning with ConditionalConjunction, e.g. "if", and a 'mainClause' representing the consequence if the condition proves to be true.

Abstract syntax. ConditionalSentence is a kind of RepresentationSentences :: Constrained-LanguageSentence. It is composed of one mandatory ModalSVOSentence in the role of 'main-Clause' and, optionally, of a ConditionalConjunction together with SVOSentence in the role of 'conditionalClause'.

[2] Subject – Verb – Object – (Object)

ModalSVOSentence

Semantics. ModalSVOSentence is a SVOSentence extended by a ModalVerb allowing to express: 1) the priority of the described activity, 2) the modality of the described activity, 3) the obligation or possibility of the subject to perform an action (described by a Predicate)
Abstract syntax. ModalSVOSentence is a kind of SVOSentence with an additional ModalVerb (kind of Terms :: TermHyperlink) pointing to a Terms :: ModalVerb. It constitutes the 'mainClause' of a ConditionalSentence.

ConditionalConjunction

Semantics. ConditionalConjunction is part of a ConditionalSentence and commences the 'conditionalClause'.
Abstract syntax. ConditionalConjunction is a kind of Terms :: TermHyperlink that, as part of ConditionalSentence, subsets 'hyperlink', being part of Elements :: HyperlinkedSentence. It points to a Terms :: ConditionalConjunction (see section 14.8).

Subject

Semantics. Subject denotes the part of an SVOSentence being its subject from the point of view of natural language grammar. Subject points to a Phrases :: Phrase that is associated with an Actors :: Actor or SystemRepresentations :: SystemUnderDevelopment (see sections 14.4 and 14.5). This element can perform an action described by the predicate of the SVOSentence.
Abstract syntax. Subject is a kind of Phrases :: PhraseHyperlink that in a context of an SVOSentence subsets the 'hyperlink' being part of a Elements :: HyperlinkedSentence. It is thus part of an SVOSentence and points to a Phrases :: Phrase. The Phrase that is associated with the Subject cannot be one of Phrase's subclasses. The Phrases :: Phrase has to belong to an Actors :: Actor or a SystemElements :: SystemElement.

Predicate

Semantics. Predicate hyperlinks an action performed by a Subject and all the words governed by this action's Phrases :: VerbPhrase or modifying it in a given SVOSentence.
Abstract syntax. Predicate is kind of Phrases :: PhraseHyperlink that in a context of SVOSen-

tence subsets 'hyperlink' being part of Elements :: HyperlinkedSentence. It is thus part of an SVOSentence and points to a Phrases :: VerbPhrase. The VerbPhrase that is associated with the Predicate must be either Phrases :: SimpleVerbPhrase or Phrases :: ComplexVerbPhrase.

ModalVerb

Semantics. ModalVerb is an additional element of ModalSVOSentence. It allows for expressing modality, priority, obligation and/or possibility of action performed by the Subject of the sentence.
Abstract syntax. ModalVerb is kind of Elements :: Hyperlink that in a context of ModalSVOSentence subsets hyperlink being part of Elements :: HyperlinkedSentence. It points to a Terms :: ModalVerb (see section 14.8).

13.3.3 Concrete syntax and examples

Source:

```
[[n:Customer]] [[v:signs up p:for n:exercises]].
```

View:

Customer : signs up : for : exercises.

Figure 13.6: SVOSentence concrete syntax example

SVOSentence. Its concrete syntax depends on the context in which the particular SVOSentence is presented to the user. It can be represented in a source form or view form, where hyperlinks are presented as in a Wiki. In the source form, SVOSentence consists of a hyperlink to a Phrases :: Phrase (the Subject) and a hyperlink to a Phrases :: VerbPhrase (the Predicate). In the view form, the SVOSentence is represented as a set of coloured hyperlinks separated with colons (see Figure 13.6). The preceding letter with the colon denotes the part of speech (e.g. *n*oun, *v*erb). See section 14.7 for more details.

Source:

```
[[n:Customer]] a:must [[v:receive n:sign-up confirmation]].
```

View:

Customer : must : receive : sign-up confirmation.

Figure 13.7: ModalSVOSentence concrete syntax example

ModalSVOSentence. Its concrete syntax is analogous to the SVOSentence's concrete syntax, with addition of a ModalVerb between a Subject and a Predicate (see Figure 13.7). The preced-

ing letter with the colon denotes the part of speech (e.g. *n*oun, *v*erb, *a*uxiliary (modal verb)). See below section 14.7 for more details.

Source:

```
c:If [[d:a n:customer]] [[v:signs up p:for d:a n:course]], [[d:the
n:system]] a:must [[v:bill d:this n:customer]].
```

View:

If : a : customer : signs up : for : a : course : , the : system : must : bill : this : customer.

Figure 13.8: ConditionalSentence concrete syntax example

ConditionalSentence. A ConditionalSentence consists of one ModalSVOSentence and, optionally, of one SVOSentence. Their concrete syntax is described above. Additional syntax elements are the ConditionalConjunction preceding the ConditionalSentence and the comma separating the 'conditionalClause' and the 'mainClause'. The first is only included if the sentence contains a 'conditionalClause' (see figure 13.8). The preceding letter with the colon denotes the part of speech (e.g. *n*oun, *v*erb, *c*onditional conjunction). See below and section 14.7 for more details.

Subject. Predicate. Their concrete syntax is not changed in respect to that of the Phrases :: PhraseHyperlink meta-class (see section 14.7).

ConditionalConjunction. ModalVerb. Their concrete syntax depends on the context in which they are presented to the user. They can be represented in source or view form. In the source form, they consist of the linked terms' names preceded by a letter with a colon (":") indicating the term type ("c:" for conditional conjunction, "a:" for modal verb (auxiliary)). In view form, they are represented as the linked terms' names separated by colons (see figures 13.8).

13.4 Scenario sentences

13.4.1 Overview

This package describes scenario sentences. It contains SVOScenarioSentence as a sentence in the SVO grammar, ConditionSentence as a condition referring to a sentence being next in a scenario and ControlSentence determining the flow of control in a scenario. ControlSentence has three specialised concrete classes: InvocationSentence, PreconditionSentence, PostconditionSentence.

13.4.2 Abstract syntax and semantics

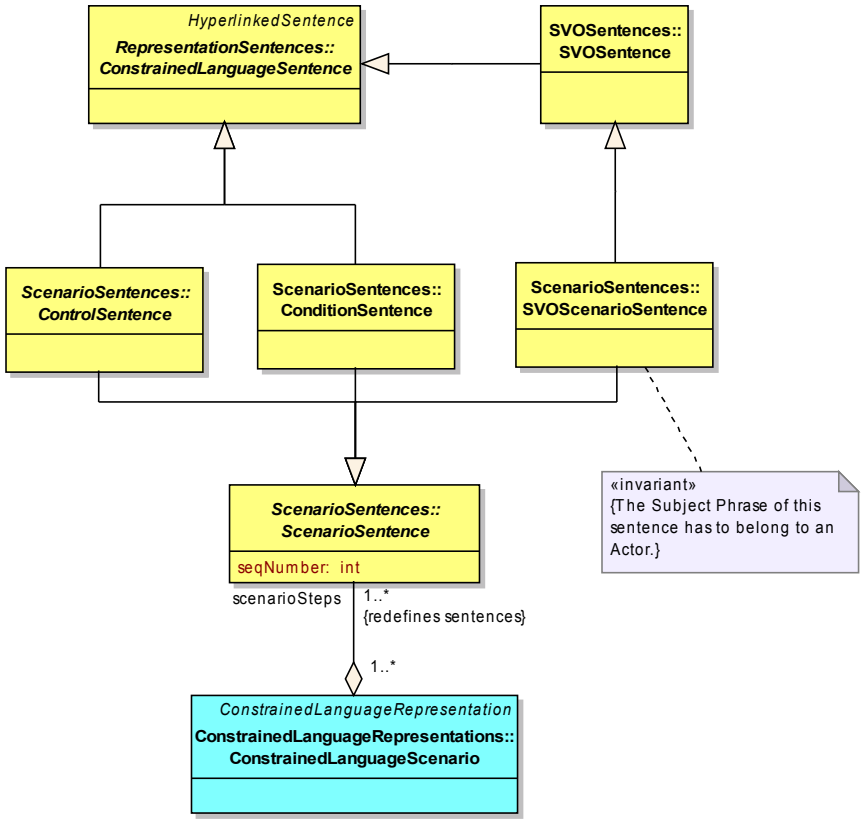


Figure 13.9: Scenario Sentences

Figure 13.9 shows part of the RSL meta-model which deals with the content of the ScenarioSentences package. The classes in this package are described in detail below.

ScenarioSentence

Semantics. A ScenarioSentence is a sentence which can be used in a scenario. To use sentence types which do not specialise from the ScenarioSentence in a scenario description is not possible since the sentences in a scenario description must have an order. Since the sentences in a scenario description may have different purposes, the ScenarioSentence is just the base for several more specialised sentence types.

Abstract syntax. ScenarioSentence is an abstract class and is a base for all the scenario sentences. It includes an attribute called seqNumber and type int. This attribute defines the sentence's position in the scenario description. ScenarioSentences form scenarioSteps of ConstrainedLanguageRepresentations :: ConstrainedLanguageScenarios. ScenarioSentence's

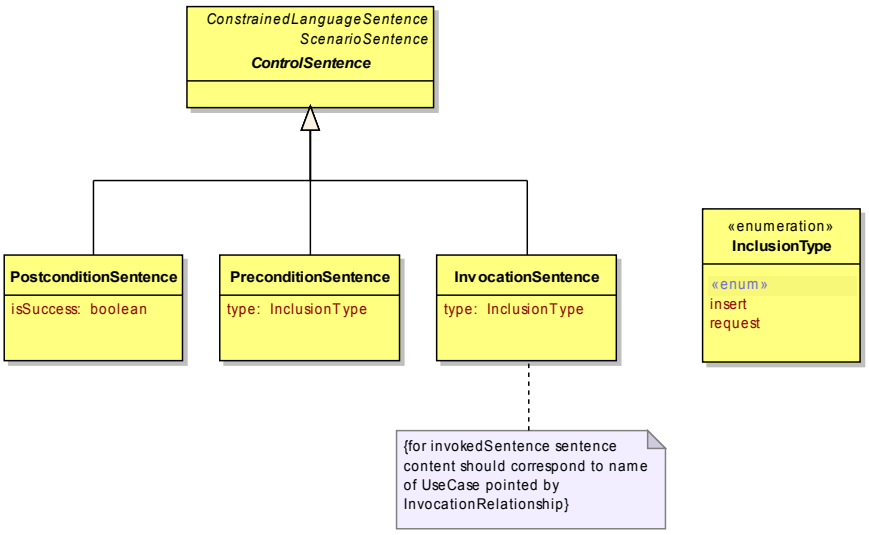


Figure 13.10: Control Sentences

subclasses are SVOScenarioSentence, ControlSentence and ConditionSentence, which are described in detail in the sections below.

SVOScenarioSentence

Semantics. SVOScenarioSentence describes a single scenario step (an action) in the form of a sentence in the SVO(O) grammar. This action can be performed by an actor or by the system.
Abstract syntax. SVOScenarioSentence is a kind of RepresentationSentences :: ScenarioSentence and has the whole syntax of SVOSentence :: SVOSentence. Because the action described in SVOScenarioSentence can be performed only by an actor or by the system, there is a constraint that the Phrases :: Phrase associated with this sentence as a subject (see SVOSentences :: SVOSentence) has to belong to an Actors :: Actor or SystemRepresentations :: SystemUnderDevelopmet.

ConditionSentence

Semantics. ConditionSentence is a special kind of scenario sentence that controls the flow of scenario execution. It is a point of conditional control flow: the following scenario step can be executed only when the condition expressed by the ConditionSentence is true.
Abstract syntax. ConditionSentence is a kind of RepresentationSentences :: ScenarioSentence It also derives from RepresentationSentences :: ConstrainedLanguageSentence.

ControlSentence

Semantics. ControlSentence is a general type of scenario sentences that control the flow of scenario execution. Depending on the concrete kind of ControlSentence, the flow of execution can be initiated, stopped or moved to another use case.
Abstract syntax. ControlSentence is a kind of RepresentationSentences :: ScenarioSentence. It also derives from RepresentationSentences :: ConstrainedLanguageSentence. This abstract class is a generalisation of concrete classes: InvocationSentence, PreconditionSentence and PostconditionSentence.

InvocationSentence

Semantics. InvocationSentence denotes the invocation of another use case scenario from within the currently performed use case scenario. There are two types of InvocationSentence: insert and request. Insert means that the system invokes another use case by inserting its scenario sentences. Request means that the Actor requests invoking another UseCaseRelationships :: UseCase – it depends on the actor decision whether scenario sentences of invoked use case will be inserted or not. After performing all scenario steps of the invoked use case, the flow of execution returns to the invoking use case scenario to execute the remaining sentences. InvocationSentence is semantically related to PreconditionSentence (see below).
Abstract syntax. InvocationSentence is a kind of ControlSentence. It has the 'type' attribute determining the type of InvocationSentence, which can have one of the values enumerated in InclusionType.

PreconditionSentence

Semantics. PreconditionSentence is an initial sentence of every use case scenario. It indicates where the flow of control of every use case scenario starts. There are two types of PreconditionSentence: insert and request. PreconditionSentence of type request is always performed when the actor triggers a use case directly or requests invoking a use case (see InvocationSentence above) from another use case scenario through initial actor action (first SVO(O) sentence in the scenario). When use case is invoked by inserting its scenario into the flow of invoking use case, the initial action is omitted. In this case PreconditionSentence of type insert is performed. PreconditionSentence may contain an associated condition which must be fulfilled before executing the sentence.
Abstract syntax. PreconditionSentence is a kind of ControlSentence. It has the 'type' attribute

determining the type of PreconditionSentence, which can have one of the values enumerated in InclusionType.

PostconditionSentence

Semantics. PostconditionSentence is a final sentence of every use case scenario. It indicates if the goal of a use case has been reached or not.
Abstract syntax. PostconditionSentence is a kind of ControlSentence. Its isSuccess attribute can have value 'true' or 'false'.

InclusionType

Semantics. InclusionType specifies the type of InvocationSentence and PreconditionSentence scenario sentences.
Abstract syntax. InclusionType is an enumerator which defines values: insert and request.

13.4.3 Concrete syntax and examples

ScenarioSentence. As an abstract meta-class, this meta-model element has no concrete syntax. It can be formulated in any of the representations of meta-classes that specialise from it.

ControlSentence. As an abstract meta-class, this meta-model element has no concrete syntax. It can be formulated in any of the representations of meta-classes that specialise from it.

Source:

```
1. [[n:Customer]] [[v:wants to sign up p:for n:exercises]]
```

View:

1. Customer : wants to sign up : for : exercises

Figure 13.11: SVOScenarioSentence example

SVOScenarioSentence. SVOScenarioSentence's concrete syntax is an ordered list of words. It has structure similar to SVOSenteces :: SVOSentence. In addition to SVOsentence, SVOScenarioSentence has its sequence number in a scenario placed at its front. See Figure 13.11 for an example.

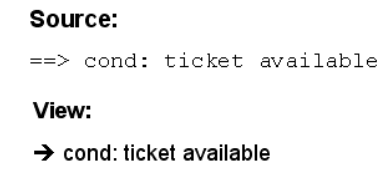


Figure 13.12: ControlSentence example

ConditionSentence. Concrete syntax of ConditionSentence is a special sign '==>', key word 'cond:' and a set of words forming a ConstrainedLanguageSentence. See Figure 13.12 for an example.

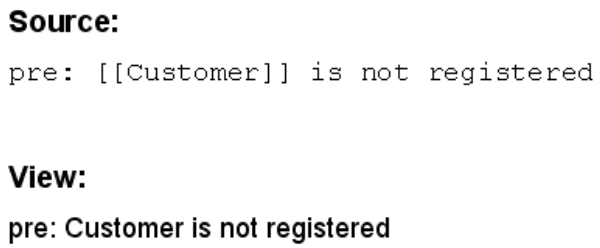


Figure 13.13: PreconditionSentence example

PreconditionSentence. PreconditionSentence's is notated by a keyword 'pre:' and a sentence in a constrained language (wiki-like description). PreconditionSentence can occur only before the first sentence in the scenario. See Figure 13.13 for an example.

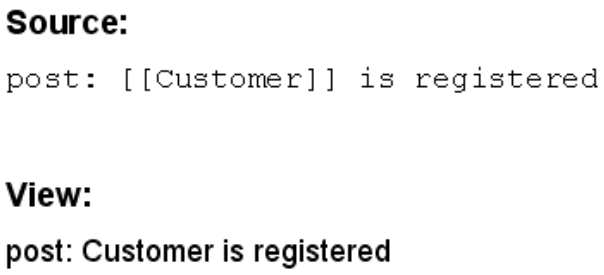


Figure 13.14: PostconditionSentence example

PostconditionSentence. PostconditionSentence is notated by a keyword 'post:' and a sentence in a constrained language (wiki-like description). PostconditionSentence can occur only after the last sentence in the scenario. See Figure 13.14 for an example.

InvocationSentence. InvocationSentence is notated by a special sign '==>', one of two keywords: 'invoke/request:' or'invoke/insert:' and a sentence in a constrained language (this constitutes the name of invoked use case). See Figure 13.15 for an example.

InclusionType. Concrete syntax of this element is one of two expressions: 'invoke/request:', 'invoke/insert:'. See Figure 13.15.

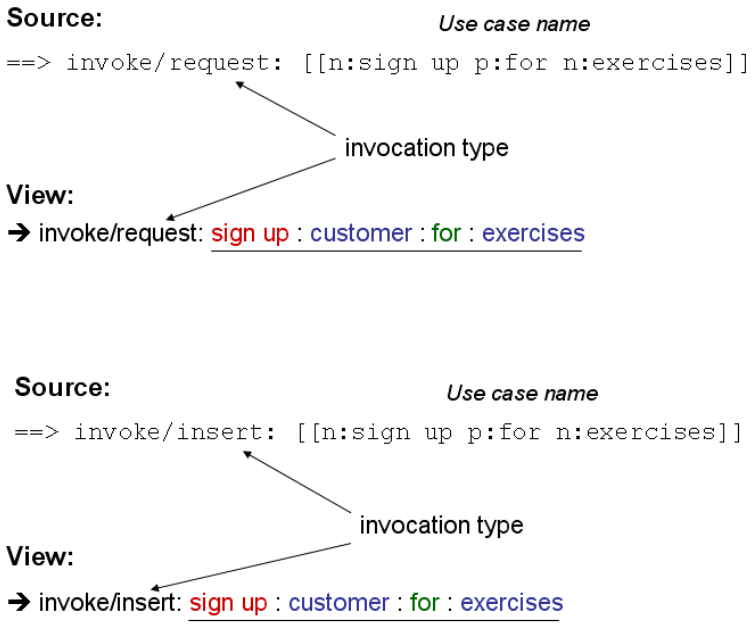


Figure 13.15: InvocationSentence example

13.5 Activity sentences

13.5.1 Overview

This package describes scenario sentences for ActivityRepresentations. It contains ActivitySVOScenarioSentence as a sentence in the SVO grammar, ActivityConditionSentence and ActivityControlSentence. ActivityControlSentence has three specialised concrete classes: ActivityInvocationSentence, ActivityPreconditionSentence, ActivityPostconditionSentence.

13.5.2 Abstract syntax and semantics

Abstract syntax for this package is shown in Figures 13.16 and 13.17.

ActivitySVOScenarioSentence

Semantics. An ActivitySVOScenarioSentence represents ScenarioSentences :: SVOScenarioSentence in ActivityRepresentations :: ActivityScenario. It has similar semantics to its base class. Additionally, ActivitySVOScenarioSentence as an UML :: BasicActivities :: ActivityNode represents scenario step on activity diagram in context of flow control within an Require-

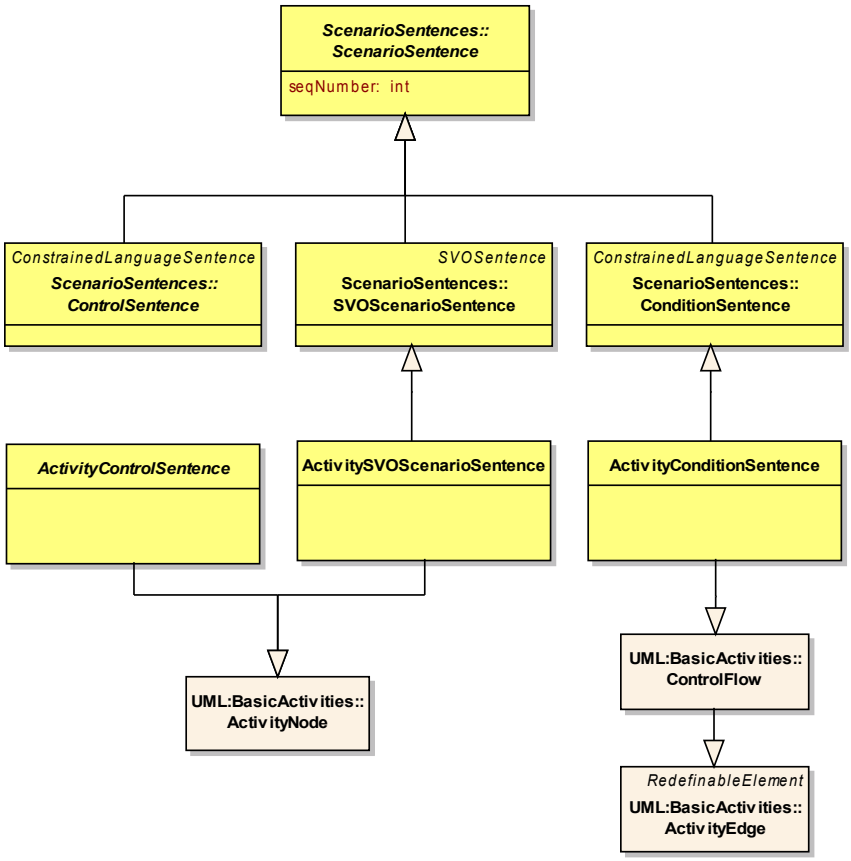


Figure 13.16: ActivityScenarioSentences

mentsSpecifications :: Usecase.

Abstract syntax. An ActivitySVOScenarioSentence is a kind of ScenarioSentences :: SVOScenarioSentence. It also derives from UML :: BasicActivities :: ActivityNode. It redefines its 'subject' and verbWithObjects with ActivitySentenceConstructs :: ActivitySubject and ActivitySentenceConstructs :: ActivitySubject.

ActivityConditionSentence

Semantics. An ActivityConditionSentence represents ScenarioSentences :: ConditionSentence in ActivityRepresentations :: ActivityScenario.

Abstract syntax. An ActivityConditionSentence is a kind of ScenarioSentences :: ConditionSentence. It also derives from UML :: BasicActivities :: ControlFlow.

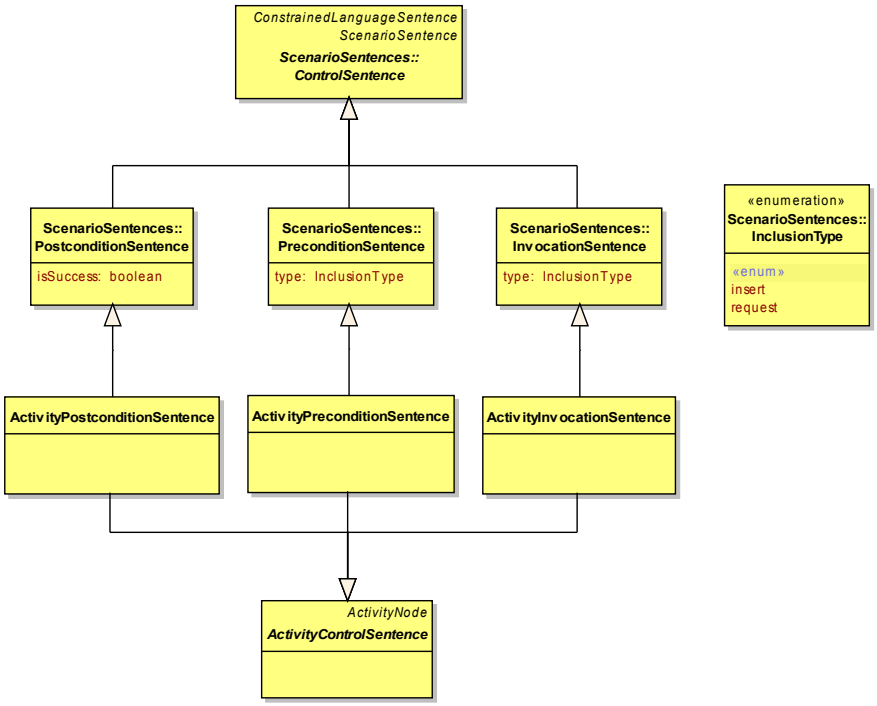


Figure 13.17: ActivityControlSentences

ActivityControlSentence

Semantics. An ActivityControlSentence represents ScenarioSentences :: ControlSentence in ActivityRepresentations :: ActivityScenario. It has three concrete subclasses: ActivityInvocationSentence, ActivityPreconditionSentence, ActivityPostconditionSentence. Each of these three subclasses corresponds to an appropriate ScenarioSentences :: ControlSentence's subclass.
Abstract syntax. An ActivityConditionSentence is a kind of UML :: BasicActivities :: ActivityNode. It is aggregated by ActivityRepresentations :: ActivityScenario in role of activityControlSentences and subsets its 'nodes' and 'sentences'

ActivityInvocationSentence

Semantics. An ActivityInvocationSentence represents ScenarioSentences :: InvocationSentence in ActivityRepresentations :: ActivityScenario. It shows point of another RequirementsSpecifications :: Usecase invocation in an activity diagram.
Abstract syntax. An ActivityInvocationSentence is a kind of ActivityControlSentence. It also derives from ScenarioSentences :: InvocationSentence. An ActivityInvocationSentence indirectly inherits from UML :: BasicActivities :: ActivityNode

ActivityPreconditionSentence

Semantics. An ActivityPreconditionSentence represents ScenarioSentences :: PreconditionSentence in ActivityRepresentations :: ActivityScenario. It shows starting point of a scenario on an activity diagram. ActivityPreconditionSentence can have precondition of the scenario attached as a constraint. It has semantics similar to ScenarioSentences :: PreconditionSentence. Additionally it has semantics of UML's UML :: BasicActivities :: InitialNode.
Abstract syntax. An ActivityPreconditionSentence is a kind of ActivityControlSentence. It also derives from ScenarioSentences :: PreconditionSentence.

ActivityPostconditionSentence

Semantics. An ActivityPostconditionSentence represents ScenarioSentences :: PostconditionSentence in ActivityRepresentations :: ActivityScenario. It shows end point of scenario on activity diagram. ActivityPreconditionSentence can have precondition of the scenario attached as a constrained . It also shows, if scenario ends with success or failure. It has semantics similar to ScenarioSentences :: PostconditionSentence. Additionally it has semantics of UML's UML :: BasicActivities :: FinalNode.
Abstract syntax. An ActivityPostconditionSentence is a kind of ActivityControlSentence. It also derives from ScenarioSentences :: PostconditionSentence.

13.5.3 Concrete syntax and examples

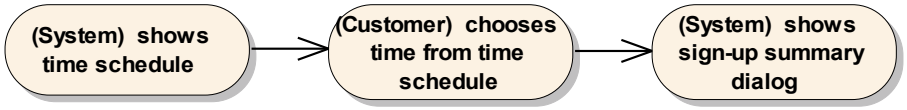


Figure 13.18: ActivitySVOScenarioSentence example

ActivitySVOScenarioSentence. ActivitySVOScenarioSentence has concrete syntax as an UML :: Activities :: Action (([Obj05b], paragraph 12.3.2, page 303): "Actions are notated as round-cornered rectangles. The name of the action or other description of it may appear in the symbol." In our case, the 'name' is a ActivitySentenceConstructs :: ActivityPredicate. Additionally, before name stands sentence's ActivitySentenceConstructs :: ActivitySubject in round brackets as a swimlane (UML :: Basic Activities ::ActivityPartition). See Figure 13.18 for an example.

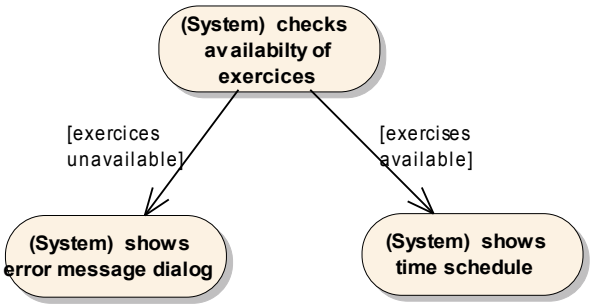


Figure 13.19: ActivityConditionSentence example

ActivityConditionSentence. Concrete syntax of ActivityConditionSentence is an arrowed line connecting two ordered ActivitySVOScenarioSentences. In addition, it contains a NaturalLanguageHypertextSentence in square brackets put near the arrow. See Figure 13.19 for an example.

ActivityControlSentence. As an abstract meta-class, ActivityControlSentence does not have a concrete syntax.

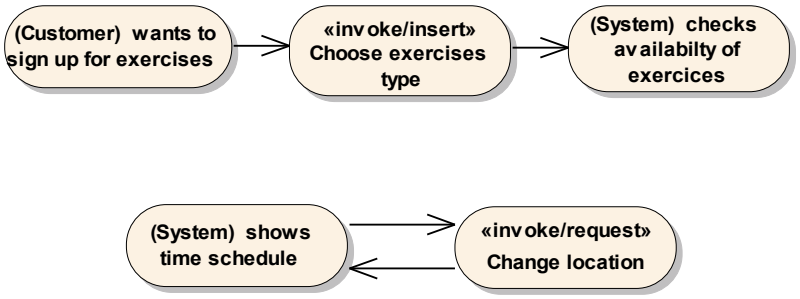


Figure 13.20: ActivityInvocationSentence example

ActivityInvocationSentence. Concrete syntax of ActivityInvocationSentence is a round-cornered rectangle with stereotype 'invoke/request' or 'invoke/insert'. The name of the ActivityInvocationSentence appears in the symbol. ActivityInvocationSentence is connected with ActivitySVOScenarioSentences with two arrowed lines. See Figure 13.20 for an example.

Figure 13.21: ActivityPreconditionSentence example

ActivityPreconditionSentence. Concrete syntax of ActivityPreconditionSentence is UML :: Basic Activities :: InitialNode with attached note. See Figure 13.21 for an example.

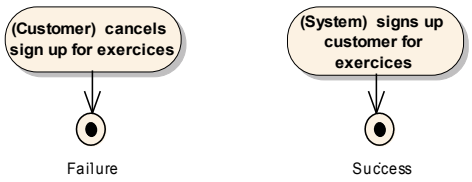


Figure 13.22: ActivityPostconditionSentence example

ActivityPostconditionSentence. Concrete syntax of ActivityPostconditionSentence is UML :: Basic Activities :: FinalNode with attached note. Additionally, the type of the sentence (success or failure) is placed near the node. See Figure 13.22 for an example.

13.6 Activity sentence constructs

13.6.1 Overview

This package describes additional constructs, introduced to fit ScenarioSentences :: SVOScenarioSentence into activity diagrams. Such Constructs are ActivityPredicate derived from BasicActivities :: ActivityNode and ActivitySubject derived from BasicActivities :: ActivityPartition.

13.6.2 Abstract syntax and semantics

Abstract syntax for this package is shown in Figures 13.23 and 13.24.

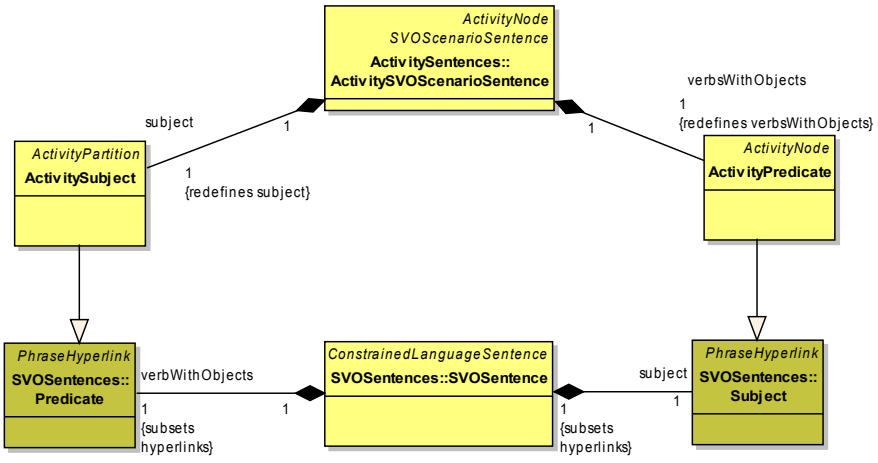


Figure 13.23: ActivitySVOSentence

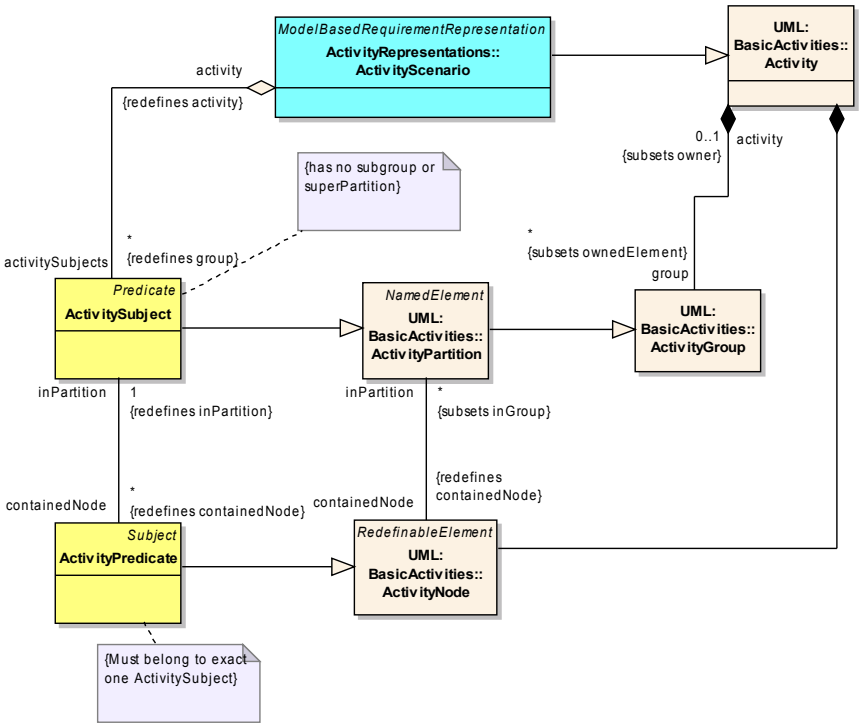


Figure 13.24: ActivityScenarioPartition

ActivitySubject

Semantics. An ActivitySubject represents SVO sentence's subject on Activity diagram as swimlane.

Abstract syntax. An ActivitySubject is kind of SVOSentences :: Subject. It also derives from BasicActivities :: ActivityPartition. It redefines its containedNode with ActivityPredicate. An ActivitySubject is associated with ActivityRepresentations :: ActivityScenario in role of activitySubjects (redefines its activityGroup). It is also aggregated by ActivitySentences :: ActivitySVOScenarioSentence. It redefines its subject.

ActivityPredicate

Semantics. An ActivityPredicate represents SVO sentence's predicate on Activity diagram as an ActivityNode .

Abstract syntax. An ActivityPredicate is kind of SVOSentences :: Predicate. It also derives from BasicActivities :: ActivityNode. It redefines its 'inPartition' with ActivitySubject. An ActivitySubject is aggregated by ActivitySentences :: ActivitySVOScenarioSentence. It redefines its verbWithObjects.

13.6.3 Concrete syntax and examples

ActivitySubject. ActivitySubject's concrete syntax is similar to BasicActivities :: ActivityPartition's concrete syntax. It is represented as a subject's text in round brackets placed in Activity symbol before name (ActivityPredicate). See Figure 13.25 for an example.

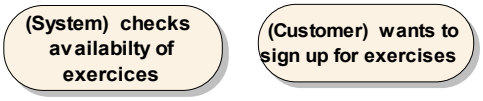


Figure 13.25: ActivitySubject and Preditace example

ActivityPredicate. ActivityPredicate's concrete syntax is a predicate's text placed in Activity symbol after partition name (ActivitySubject). See Figure 13.25 for an example.

13.7 Interaction sentences

13.7.1 Overview

This package describes scenario sentences in InteractionRepresentations. It contains InteractionSVOScenarioSentence as a sentence in the SVO grammar, InteractionConditionSentence and InteractionControlSentence. InteractionControlSentence has three specialised concrete classes: InteractionInvocationSentence, InteractionPreconditionSentence, InteractionPostconditionSentence. Syntax and semantics of all these classes is described in the sections below.

13.7.2 Abstract syntax and semantics

Abstract syntax for this package is shown in Figures 13.26, 13.27 and 13.28.

InteractionScenarioSentence

Semantics. An InteractionScenarioSentence represents ScenarioSentences :: ScenarioSentence in an InteractionScenario.
Abstract syntax. An InteractionScenarioSentence is the abstract base class for all sentences

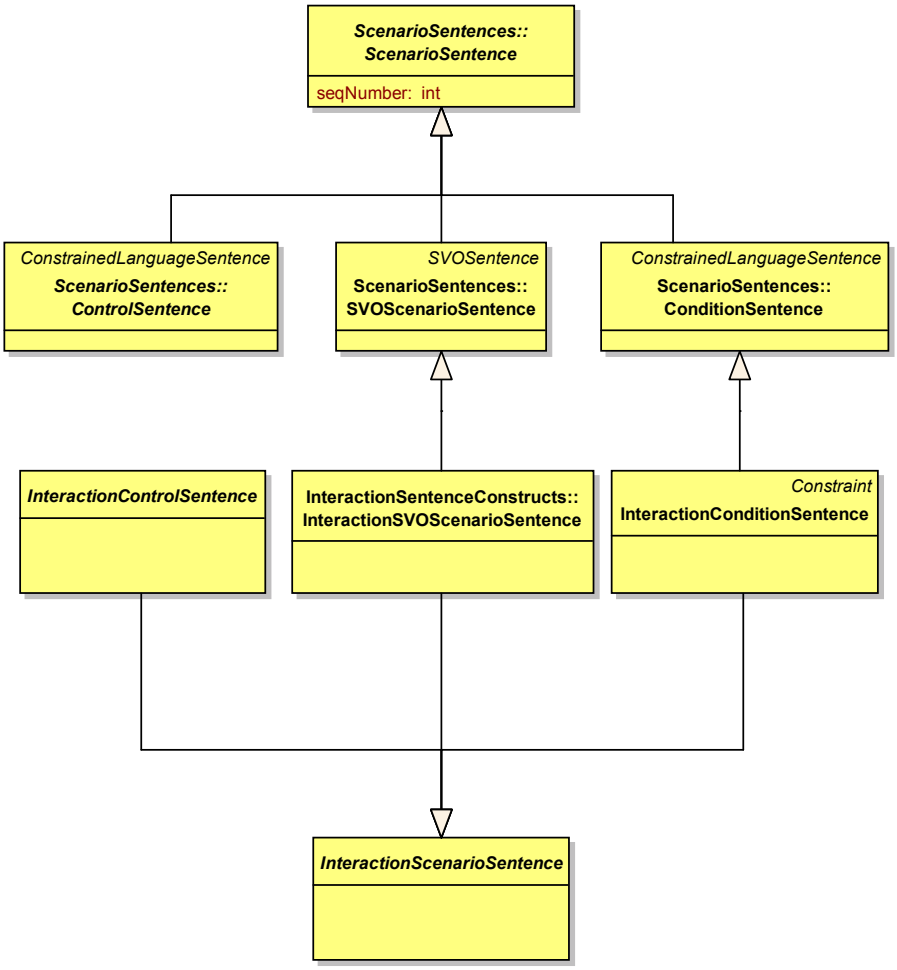


Figure 13.26: InteractionScenarioSentences

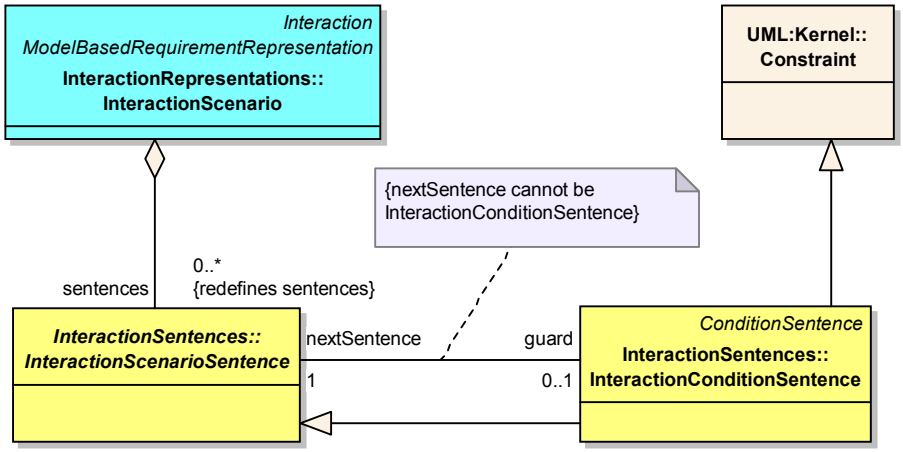


Figure 13.27: InteractionConditionSentences

that may occur in InteractionScenarios. It is associated with InvocationConditionSentence and is a component of InteractionRepresentations :: InteractionScenario.

InteractionConditionSentence

Semantics. An InteractionConditionSentence represents a conditional sentence in an InteractionScenario. It acts as guard for another InteractionScenarioSentence, which gets executed only if the condition evaluates to true. InteractionConditionSentence's condition is represented as it is common for constraints in UML 2.0.

Abstract syntax. An InteractionConditionSentence is a kind of ScenarioSentences::ConditionSentence, so it inherits all associations from this class. It is also derived from InteractionScenarioSentence as it occurs only in InteractioScenarios. Additionally, it is a subclass of UML :: Kernel :: Constraint. There is a InteractionScenarioSentence associated to InteractionConditionSentence with the role nextSentence, the InteractionConditionSentence is the guard of that InteractionScenarioSentence.

InteractionControlSentence

Semantics. An InteractionControlSentence represents ScenarioSentences :: ControlSentence in InteractionRepresentations :: InteractionScenario. It has three concrete subclasses: InteractionyInvocationSentence, InteractionPreconditionSentence and InteractionPostconditionSentence. Each of these three subclasses corresponds to appropriate ScenarioSentences :: ControlSentence's subclass.

Abstract syntax. An InteractionControlSentence is a kind of InteractionScenarioSentence, so it

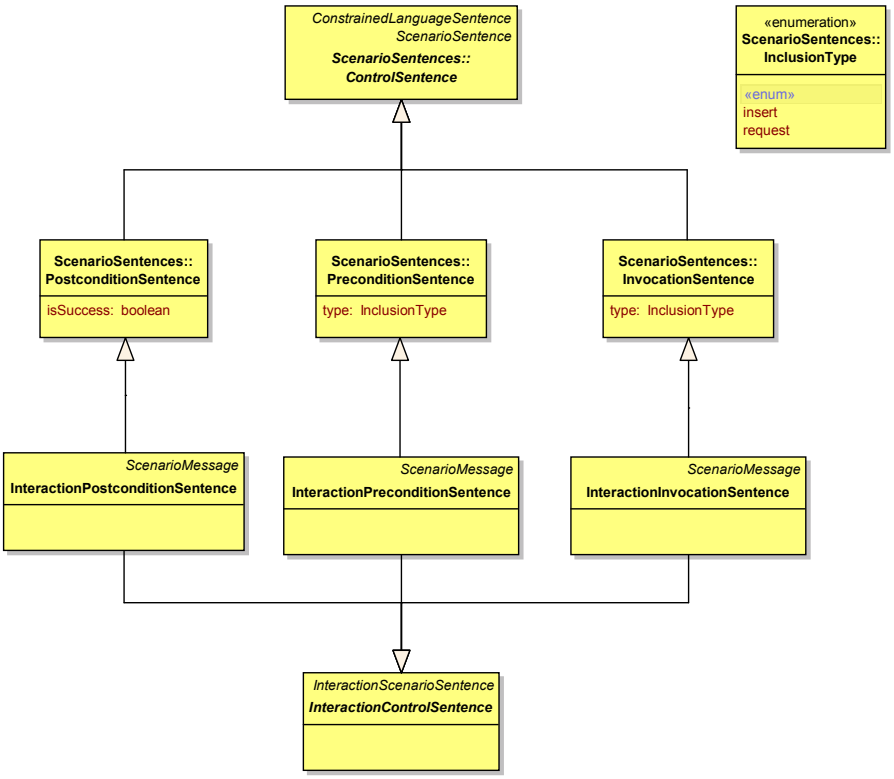


Figure 13.28: InteractionControlSentences

may occur in InteractionScenarios. As it derives from ControlSentence, it inherits all relations from this class.

InteractionInvocationSentence

Semantics. An InteractionInvocationSentence represents ScenarioSentences :: InvocationSentence in InteractionRepresentations :: InteractionScenario. It shows point of another use case invocation in a communication or sequence diagram. As invocation of another use case is triggered by system elements in all cases, InteractionInvocationSentences may start only at SystemElementLifelines.

Abstract syntax. An InteractionInvocationSentence is a kind of InteractionControlSentence. It also derives from ScenarioSentences :: InvocationSentence and inherits all relations from this base class. A InteractionInvocationSentence starts at a SystemElementLifeline with a SystemElementMessageEnd and ends at a InvokeLifeline with a InvokeMessageEnd.

InteractionPreconditionSentence

Semantics. An InteractionPreconditionSentence represents ScenarioSentences :: InteractionSentence in InteractionRepresentations :: InteractionScenario. It shows the entrypoint of the scenario represented by an interaction diagram. A constraint may be attached to a InteractionPreconditionSentence serving as a precondition of the scenario. It has semantics similar to ScenarioSentences :: PreconditionSentence.
Abstract syntax. An InteractionPreconditionSentence is a kind of InteractionControlSentence. It also derives from ScenarioSentences :: PreconditionSentence. It is associated with InteractionSentenceConstructs :: ActorMessageEnd.

InteractionPostconditionSentence

Semantics. An InteractionPostconditionSentence represents ScenarioSentences :: PostconditionSentence in a scenario description by InteractionRepresentations :: InteractionScenario. It shows the end point of a scenario represented by an interaction diagram. Similar to InteractionPreconditionSentence, a constraint may be attached to InteractionPostconditionSentence, serving as a postcondition of the described scenario. Additionally, InteractionPostconditionSentence shows, if the scenario ends with success or failure. Further semantics of InteractionPostconditionSentence is similar to ScenarioSentences :: PostconditionSentence.
Abstract syntax. An InteractionPostconditionSentence is a kind of InteractionControlSentence. It also derives from ScenarioSentences :: PostconditionSentence. It is associated with InteractionSentenceConstructs :: ActorMessageEnd.

13.7.3 Concrete syntax and examples

This section describes concrete syntax for the sentences described in the section above using the figures 13.29 and 13.30 known from section 12.6 as examples.

InteractionScenarioSentence This class is abstract, so there is no concrete syntax.

InteractionConditionSentence In interaction diagrams used for describing scenarios, conditional sentences are represented as a condition at the next message in the scenario. A example is shown in figure 13.29 at the message “acceptHelpRequest”. The condition for this sentence should be put in curly brackets nearby the line representing the message with role nextSentence

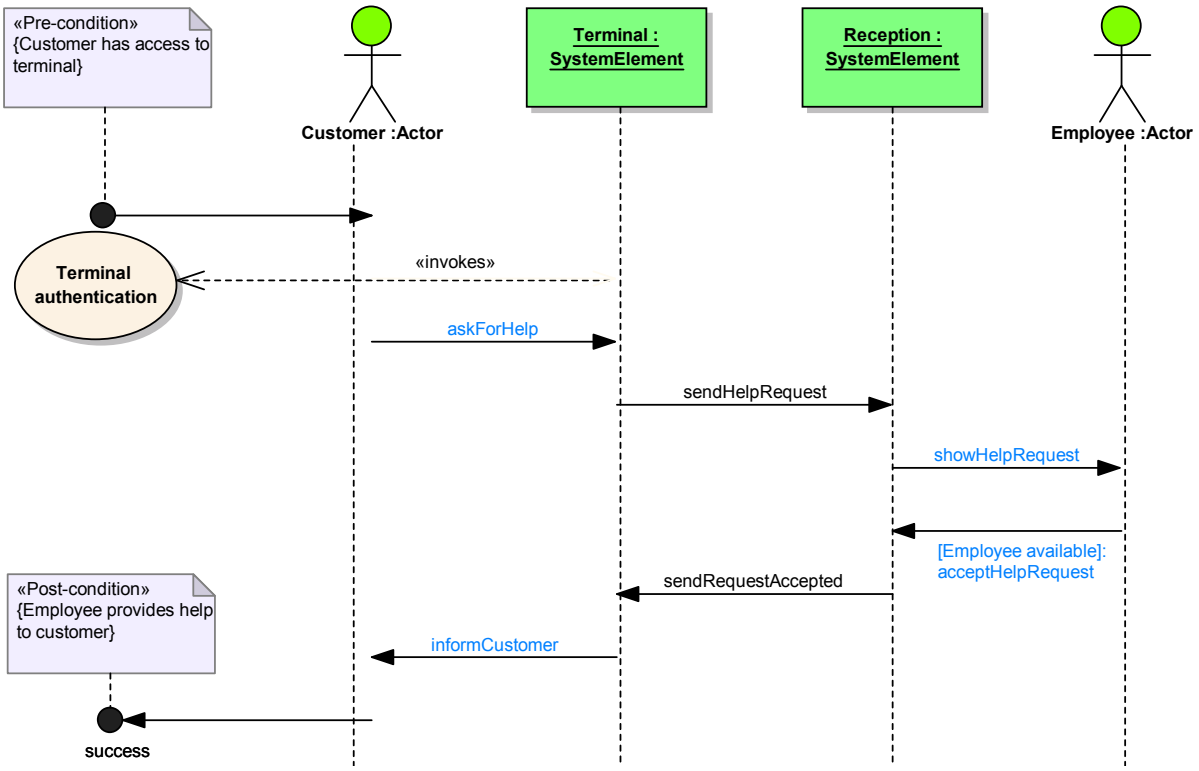


Figure 13.29: Concrete syntax of sequence diagram

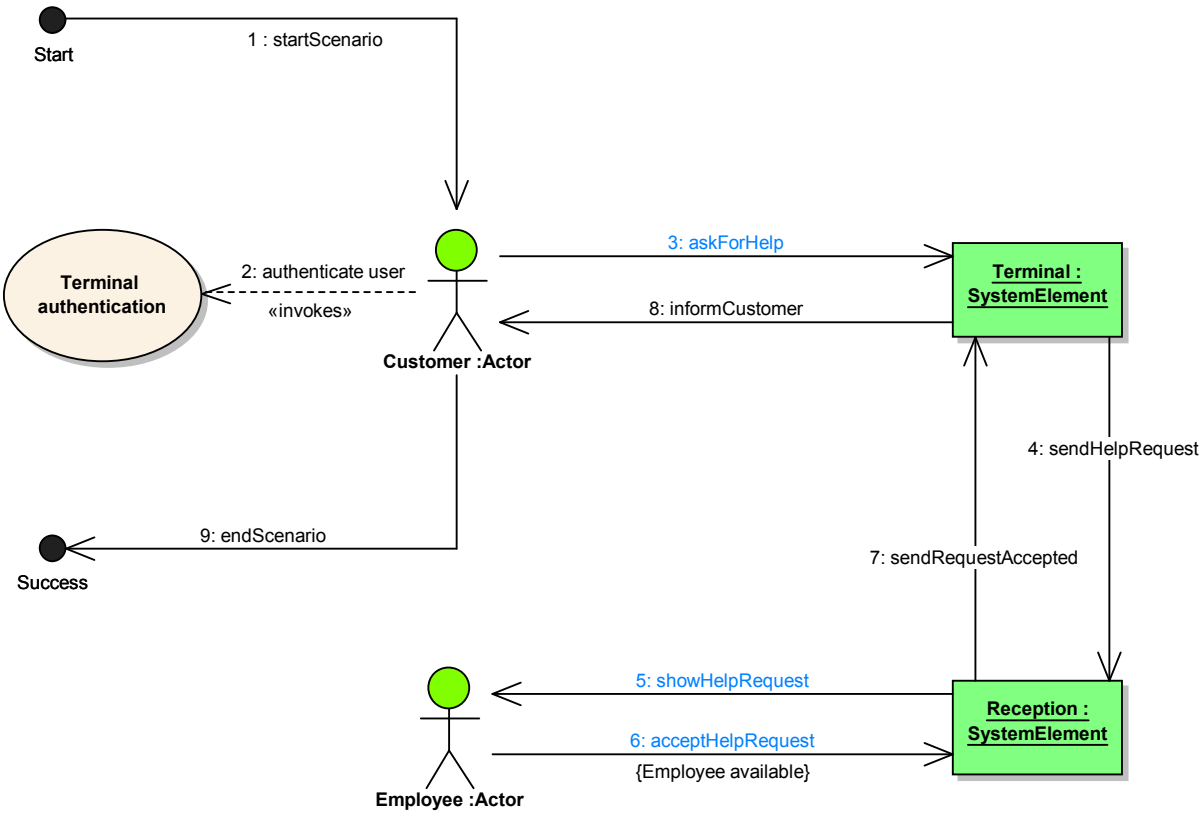


Figure 13.30: Concrete syntax of communication diagram

in the abstract syntax. In communication diagram the notation is similar to this, as message number 6 in Figure 13.30 shows.

InteractionControlSentence This class is abstract, so there is no concrete syntax.

InteractionInvocationSentence An InteractionInvocationSentence is modelled by a message starting at a lifeline and ending at a use case oval. This use case is the one the InteractionInvocationSentence invokes. The message left in figure 13.29 shows an example of InteractionInvocationSentence's concrete syntax. The use case called Terminal authentication is included in the scenario description using a message from the main actors lifeline to the use case diagram element. In figure 13.30, the same sentence is represented by message with number 2.

InteractionPreconditionSentence InteractionPrecondiationSentences are modelled as shown in the upper left corner of figure 13.29. An initial point represents the start of the scenario, the precondition is attached to this initial point as a UML constraint. In figure 13.30, the same sentence is represented by message with number 1.

InteractionPostconditionSentence The representation of InteractionPostconditionSentence is similar to the one of InteractionPreconditionSentence. A end point is used to represent the end of the scenario, the postcondiation is, analogous to the precondition of InteractionPreconditionSentence, modelled as a UML constraint attached to the end point. In figure 13.30, the same sentence is represented by message with number 10.

13.8 Interaction sentence constructs

13.8.1 Overview

Sections 12.6 and 13.7 have described the concept of InteractionScenario and the different sentence types that may be used in such scenario representations. This section explains the remaining elements of InteractionScenarios, such as different types of lifelines, messages and relations between them.

13.8.2 Abstract syntax and semantics

The abstract syntax for this package is shown in figures 13.31 to 13.35.

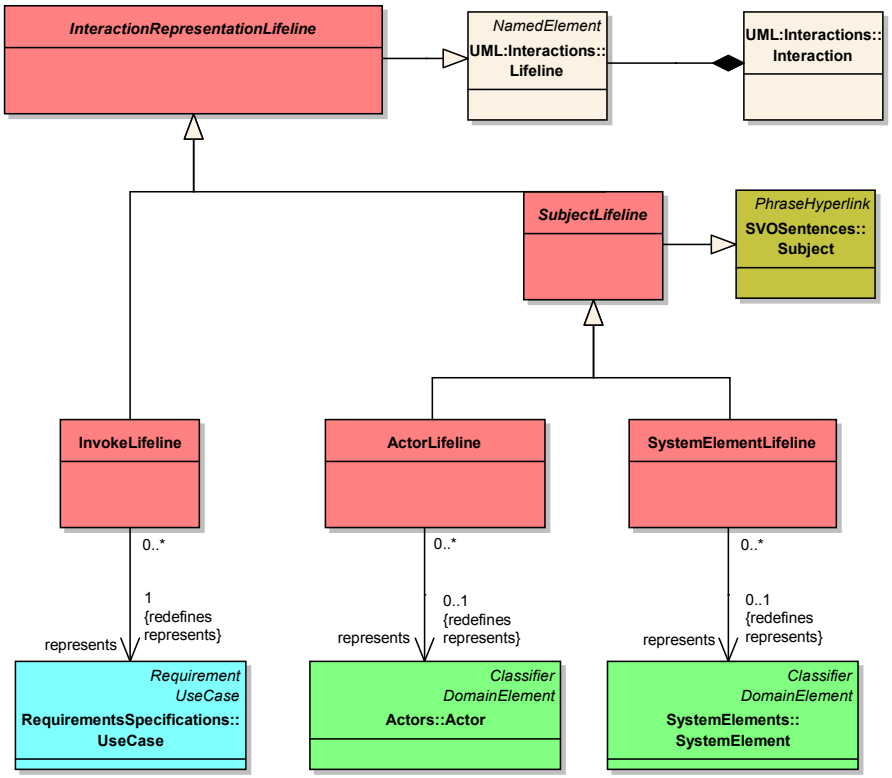


Figure 13.31: InteractionLifelines

InteractionRepresentationLifeline

Semantics. Lifelines in an InteractionScenario are always InteractionScenarioLifelines. They represent actors, components of the system under development or use cases that are invoked in the scenario. Communication between lifelines can be modeled with ScenarioMessages.
Abstract syntax. An InteractionRepresentationLifeline is the abstract base class for SubjectLifeline and InvokeLifeline. It is derived from the class UML::Interactions::Lifeline out of the UML2.0 superstructure and the class ElementRepresentation out of the package ElementRepresentations. It is a component of InteractionRepresentations :: InteractionScenario.

SubjectLifeline

Semantics. Acting subjects in the InteractionScenario are modelled as SubjectLifelines. They represent actors or components of the system under development.
Abstract syntax. A SubjectLifeline is the abstract base class for ActorLifeline and SystemElementLifeline. It is a kind of InteractionRepresentationLifeline and thus inherits all associations from it. Additionally, it is derived from SVOSentences :: Subject, so a SubjectLifeline acts as subject in a SVOSentences :: SVOSentence.

ActorLifeline

Semantics. Every actor who participates in a scenario described by a InteractionScenario is represented by an ActorLifeline. Since the scenario models an interaction between an actor and parts of the system, every ActorLifeline can have some outgoing and incoming messages.
Abstract syntax. An ActorLifeline is a kind of SubjectLifeline, so it may act as subject in a SVOSentence. An ActorLifeline may cover zero or more ActorMessageEnds, this association redefines the inherited association between Lifeline and MessageEnd. Each ActorLifeline is the representation of one Actors :: Actor, which is in represents role to ActorLifeline. An ActorLifeline can associated with InteractionRepresentations :: InteractionScenarios as a primaryActor. It is also a component of InteractionRepresentations :: InteractionScenario.

SystemElementLifeline

Semantics. A SystemElementLifeline is similar to ActorLifeline with the difference, that it represents components of the system under development instead of actors. Depending on the level of granularity the requirements engineer chose, the whole system can be modeled as one single component. Also communications between different system components or between a system component and an actor are represented as outgoing and incoming messages.
Abstract syntax. An SystemElementLifeline is a kind of SubjectLifeline, so it may act as subject in a SVOSentence similar to ActorLifeline. Each SystemElementLifeline is the representation of one SystemElements :: SystemElement, which is in represents role to SystemElementLifeline. It may cover SystemElementMessageEnds, the association describing this fact redefines the inherited association between Lifeline and MessageEnd in almost the same manner as it is done for ActorLifeline. SystemElementLifeline is also a component of InteractionRepresentations :: InteractionScenario.

InvokeLifeline

Semantics. InvokeLifelines represent use cases that are invoked in a scenario. So use cases that are needed more than once can be used in different scenarios easily and separation of concers is possible.
Abstract syntax. InvokeLifeline is derived from InteractionRepresentationLifeline and represents via association exact one RequirementsSpecifications :: UseCase, while a RequirementsSpecifications :: UseCase can be represented by more than one invoke lifelines, as the multiplicities

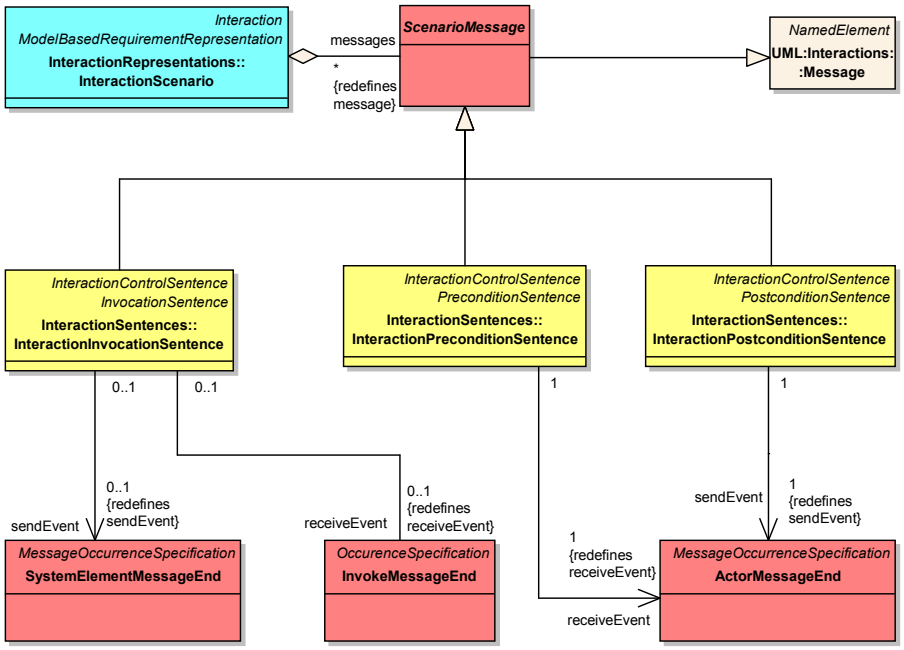


Figure 13.32: InteractionMessages

indicate. InvokeLifeline is also associated with InvokeMessageEnd as a redefinition of covered attribute.

ScenarioMessage

Semantics. A ScenarioMessage represents an interaction between different lifelines in a InteractionScenario. Depending on the interaction, one of the classes derived from ScenarioMessage is used to model the interaction.

Abstract syntax. A ScenarioMessage is derived from the class UML :: Interactions :: Message out of the UML 2.0 metamodel. Every ScenarioMessage belongs to a InteractionScenario, as the redefined aggregation with rolename message indicates. It is the abstract base class for InteractionInvocationSentence, InteractionPreconditionSentence, InteractionPostconditionSentence and PredicateMessage.

PredicateMessage

Semantics. The class PredicateMessage represents the predicate of an InteractionSVOScenarioSentence. Together with the lifeline related to it, every PredicateMessage builds up one sentence. Since it is derived from SVOSentences :: Predicate, it may contain a hyperlink which

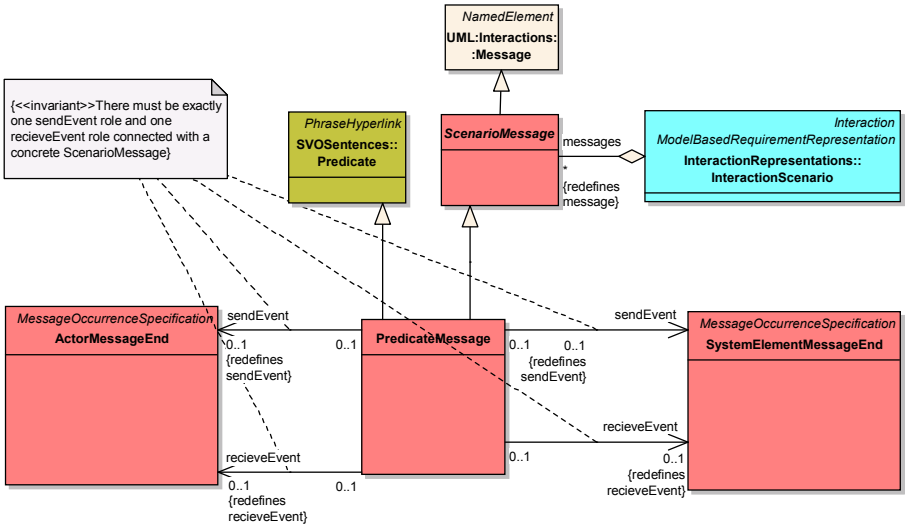


Figure 13.33: InteractionPredicateMessages

refers to elements in the vocabulary.

Abstract syntax. PredicateMessage has associations to SystemElementMessageEnd and ActorMessageEnd. These are described in detail at the class descriptions of SystemElementMessageEnd and ActorMessageEnd. To ensure that ScenarioMessage is associated to exactly one message end with role name sendEvent and exactly one with role name recieveEvent a constraint is added. PredicateMessage redefines verbWithObjects attribute for InteractionSVOScenarioSentence. PredicateMessage is also a redefinition of message attribute of InteractionRepresentations :: InteractionScenario, which is owning given PredicateMessage.

ActorMessageEnd

Semantics. A ActorMessageEnd models the connection between a PredicateMessage and a ActorLifeline. Every PredicateMessage may contain zero to two ActorMessageEnds, depending on what kind of communication is modelled with that message.

Abstract syntax. The base class of ActorMessageEnd is the class UML :: Interactions :: MessageOccurenceSpecification which is derived from UML :: Interactions :: MessageEnd. From this class the associations to Message are inherited, but because ActorMessageEnds may only be endpoints of PredicateMessages, the associations with the roles sendEvent and recieveEvent are redefined. The same applies for associated InteractionSentences :: InteractionPostconditionSentence and InteractionSentences :: InteractionPreconditionSentence. The constraint which holds for ActorMessageEnd, that also affects SystemComponentMessage, is described later in context of the class ScenarioMessage. The last class associated to ActorMessageEnd is ActorLifeline, the association between these two classes is inherited from Message and Lifeline,

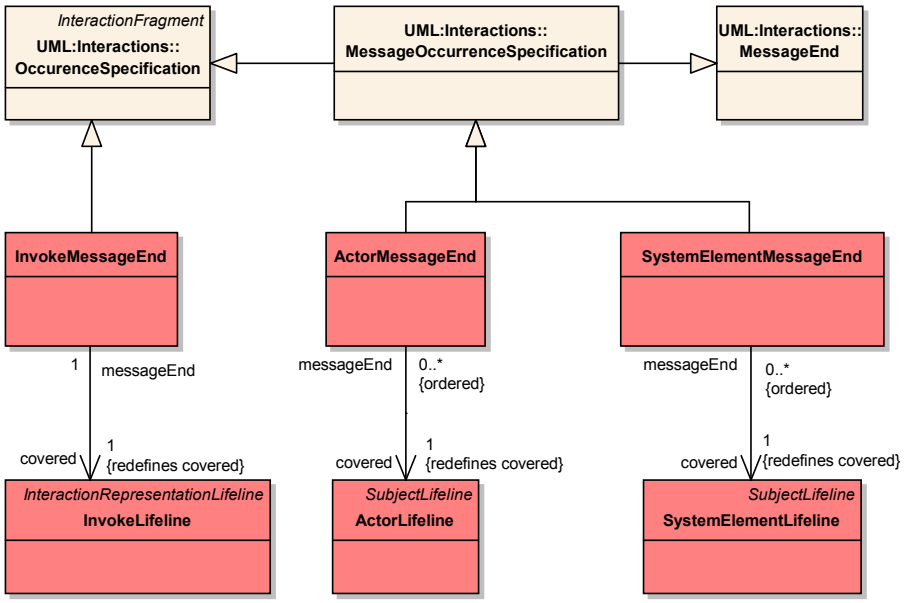


Figure 13.34: InteractionMessageEnds

but since ActorLifelines should be the only lifelines that may cover an ActorMessageEnd the association is redefined in the metamodel.

SystemElementMessageEnd

Semantics. Analogous to ActorMessageEnd, SystemElementMessageEnd models the connection between a PredicateMessage and a SystemElementLifeline.
Abstract syntax. The base class of ActorMessageEnd is the class UML :: Interactions :: MessageOccurenceSpecification which is derived from UML :: Interactions :: MessageEnd. Again analogous to ActorMessageEnd, the associations to Message are inherited from this class. Since SystemElementMessageEnds may only be endpoints of PredicateMessages, the associations with the roles sendEvent and recieveEvent are redefined here also. The same holds for the association to SystemElementLifeline, these associations are inherited from MessageEnd but must be redefined so that SystemElementMessageEnds may be covered only by SystemElementLifelines.

InvokeMessageEnd

Semantics. Analogous to ActorMessageEnd and SystemElementMessageEnd, the class InvokeMessageEnd models the connection between a ScenarioMessage and a InvokeLifeline.

Figure 13.35: InteractionSVOScenarioSentences

Abstract syntax. The base class of InvokeMessageEnd is the class UML :: Interactions :: MessageOccurenceSpecification which is derived from UML :: Interactions :: MessageEnd. Again analogous to ActorMessageEnd and SystemElementMessageEnd, the associations to Message are inherited from this class. Since InvokeElementMessageEnds may only be endpoints of InteractionInvocationSentences and their role may be only recieveEvent at the connected InteractionInvocationSentence, the appropriate association with the roles recieveEvent is redefined. The same holds for the association to InvokeLifeline, these associations are inherited from MessageEnd but must be redefined so that InvokeMessageEnds may be covered only by InvokeLifelines.

InteractionSVOScenarioSentence

Semantics. An InteractionSVOScenarioSentence represents ScenarioSentences :: SVOScenarioSentence in InteractionRepresentations :: InteractionScenario. It has similar semantics to its base class but, in contrast to its base class, it is composed of lifelines which act as subjects and objects and messages which act as predicates.

Abstract syntax. An InteractionSVOScenarioSentence is devired from ScenarioSentences ::

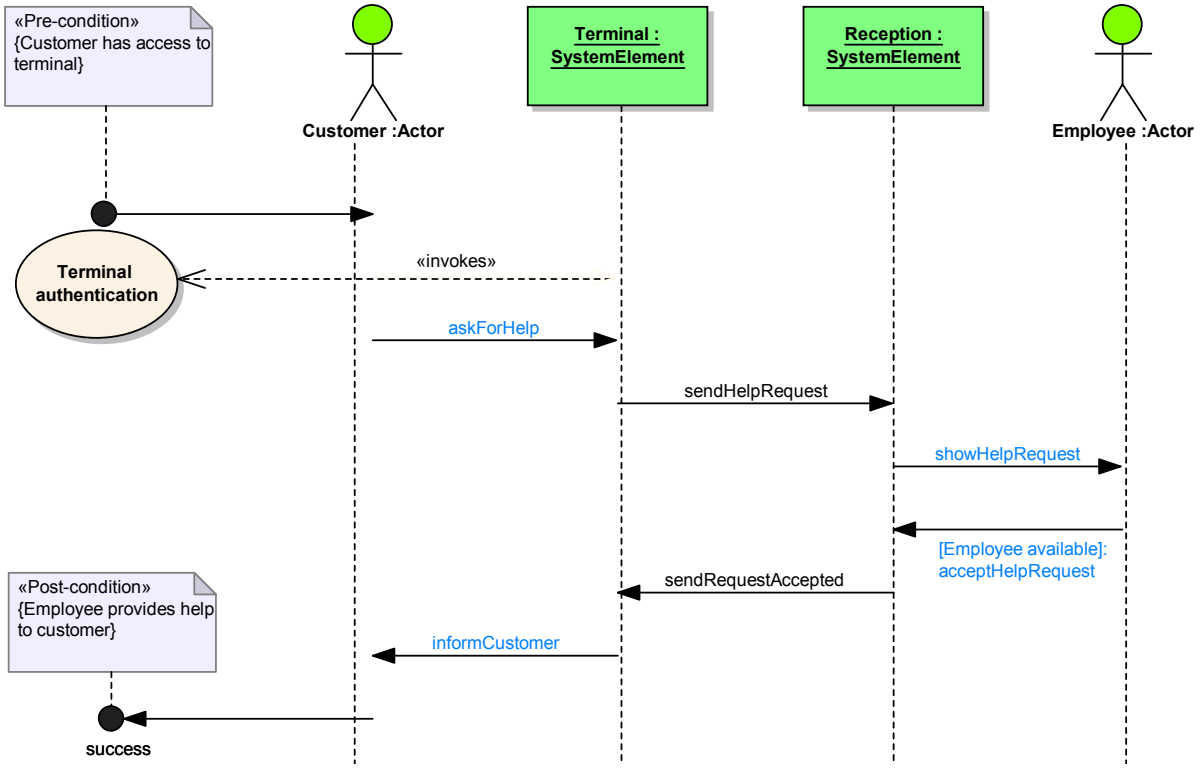


Figure 13.36: Concrete syntax of sequence diagram

SVOScenarioSentence. It redefines its subject and verbWithObjects with SubjectLifeline and PredicateMessage.

13.8.3 Concrete syntax and examples

This section describes concrete syntax for the sentence constructs described in the section above, again the figures 13.36 and 13.37 which were already used in sections 13.7 and 12.6, serve as examples.

InteractionRepresentationLifeline This class is abstract, so there is no concrete syntax.

SubjectLifeline This class is abstract, so there is no concrete syntax.

InvokeLifeline Both figures 13.36 and 13.37 contain an example for a InvokeLifeline. I both cases, this is the use case called Terminal authentication in the left part of the appropriate figure.

ActorLifeline In figure 13.36, the two outer lifelines which have a stick-figure as illustration, named “Customer” and “Employee”, are examples for a ActorLifeline. They are modelled as dotted lines as it is common in UML. In figure 13.37 these lifelines are also represented by a

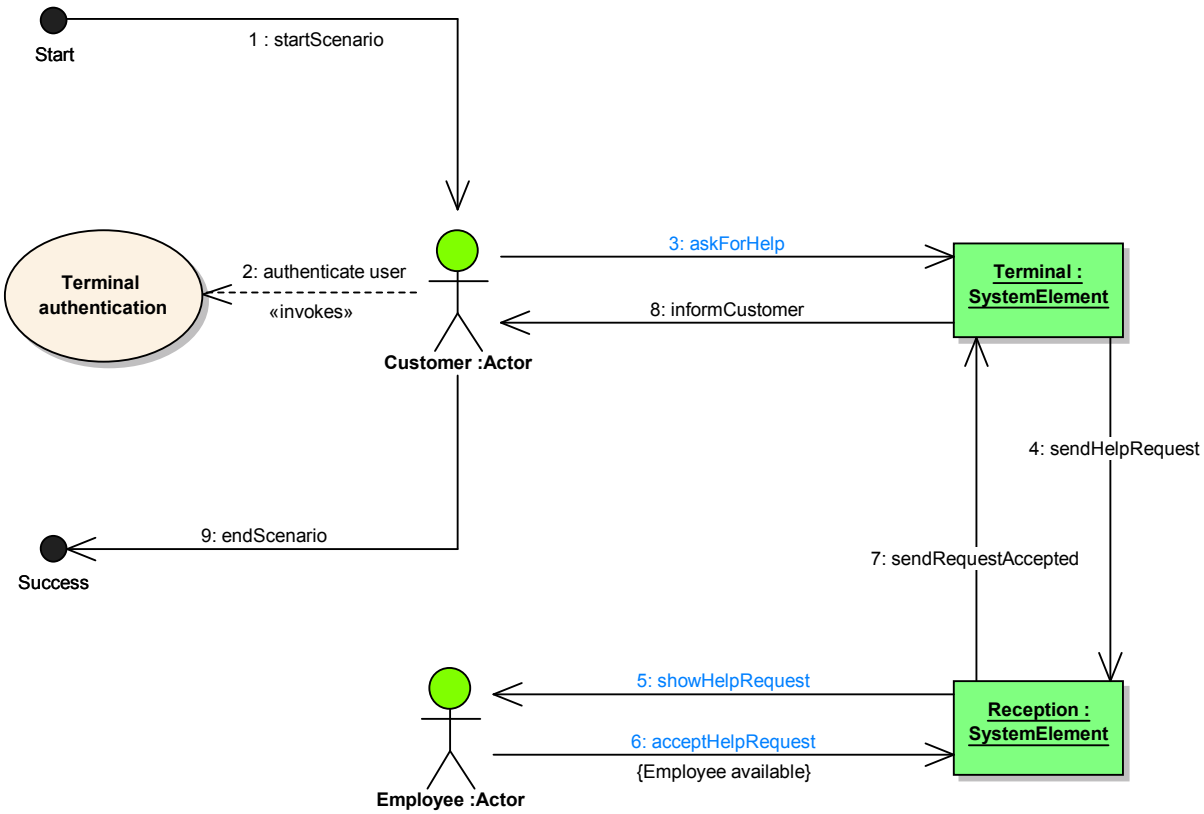


Figure 13.37: Concrete syntax of communication diagram

stick-figure, but their lifeline-nature is not explicitly modelled by a dotted line but by numbers at the incomming and outgoing messages.

SystemElementLifeline The SystemElementLifeline is represented as it is common for lifelines in UML 2.0. In sequence diagram example 13.36 the two inner lifelines named “Terminal” and “Reception” are such lifelines, in communication diagram example 13.37 the names are identical, but analogous to ActorLifeline, the lifeline-nature is not explicitly modelled by a dotted line but by numbers at the incomming and outgoing messages.

ScenarioMessage This class is abstract, so there is no concrete syntax.

PredicateMessage In the example figures, all messages between a ActorLifeline and a SystemElementLifeline or between two SystemElementLifelines are instances of PredicateMessage. That are the messages labeled with askForHelp, sendHelpRequest, showHelpRequest, acceptHelpRequest, sendRequestAccepted and informCustomer. The message from Employee-lifeline to itself is not a PredicateMessage but a InteractionConditionSentence, it’s concrete syntax is described in section 13.7.3. They are modelled as black lines between two different InteractionRepresentationLifelines. The messages in the centre of Figure 13.36, which connect the SystemElements Terminal and Reception, are PredicateMessages but do not contain a Hyperlink.

In this diagram, all messages between an Actor and a SystemElement contain a Hyperlink, as the blue font colour indicated. The whole blue part in the message name is the Hyperlink, it refers to the appropriate Phrase in the vocabulary. The points where a PredicateMessage and a InteractionRepresentationLifeline meet are ActorMessageEnds or SystemElementMessageEnds, their notation is explained at the description of the appropriate classes. The distribution of Hyperlinks in this example is not representative, there is no general restriction for a message between an Actor and a SystemElement to contain a Hyperlink neither a restriction for other messages to contain no Hyperlinks.

ActorMessageEnd The connections between the ScenarioMessages and the ActorLifelines are ActorMessageEnds. As it is common in UML sequence diagrams, the message whose end is modelled just starts or ends at the appropriate lifeline. If the ActorMessageEnd has the role of sendEvent, no additional graphical element is needed, if it has the role of receiveEvent the connection to the lifeline is modelled as a black arrow, as it is common in UML. By analogy to this, the messages in the communication diagram are represented as it is common in UML.

SystemElementMessageEnd The graphical representation of SystemComponentMessageEnd is analogous to ActorMessageEnd, just it connects not to ActorLifelines but to SystemComponentLifelines.

InvokeMessageEnd InvokeMessageEnds model the connections between an InteractionInvocationSentence and a InvokeLifeline. Since their role may be only recieveEvent to InteractionInvocationSentence, because a InvokeLifeline may only be invoked by another lifeline by may not invoke another lifeline itself, InvokeMessageEnds are modelled as a black arrow, as it is common in UML for the recieveEvent message end.

InteractionSVOScenarioSentence The InteractionSVOScenarioSentences are not modelled explicitly, but they are represented implicitly by the SubjectLifelines and PredicateMessages. For instance, the two lifelines called “Actor” and “Terminal” together with the PredicateMessage “askForHelp” build up the InteractionSVOScenarioSentence “Actor asks for help at terminal”.

Chapter 14

Domain elements

14.1 Overview

The DomainElements part defines the domain description aspects of a requirements specification. All terms that are domain related, for instance actors and components of the system under development as well as special actions of actors or the system will be stored in the domain vocabulary.

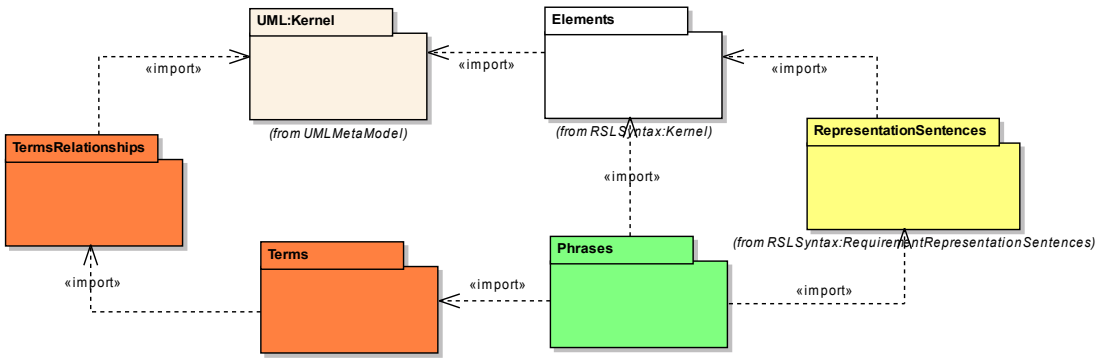


Figure 14.1: Overview of packages inside the DomainEntities part of RSL

While this part of the language has a focus on structured language, it also allows basic object-oriented representation of structured domain knowledge in the spirit of UML class diagrams. So, even a simple form of ontology can be represented in this language. Further work on the query mechanism in WP4 and the experiences from industrial applications during ReDSeeDS will show whether extensions of the requirements language would be desirable for supporting

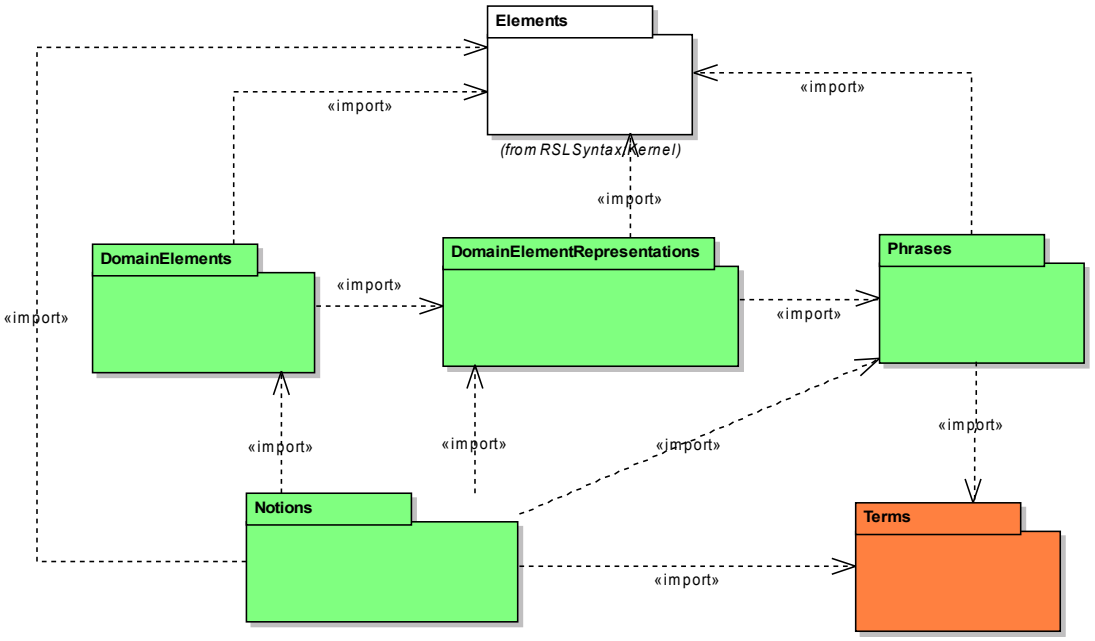


Figure 14.2: Overview of packages inside the DomainEntities part of RSL

a knowledge representation and reasoning approach, which would support the representation of ontologies even better.

The specification in this part of the Requirements Specification Language contains eight packages as shown in Figures 14.1, 14.2 and 14.3.

- The Terms package contains constructs defining terms, their structure, inflection and terms' hyperlinks that can be used in various requirements specifications. Out of these terms, more complex constructs, like phrases can be built. This is done through the use of hyperlinks to appropriate terms. This package «import»s from TermsRelationships package where relationships of different types between terms are defined.

- The TermsRelationships package contains meta-classes that define relationships between terms. Terms together with relationships form the basic structure for thesaurus and ontology. TermsRelationships package «import»s from UML :: Kernel package as term relationships are specialisations of relationship defined in UML.

- The Phrases package adds to the language important constructs that allow for building parts of sentences in a structured language. Phases of various type are constructed as sets of hyperlinks to appropriate terms. This package «import»s from RepresentationSentences as a class representing phrase is an extension of an abstract class representing a

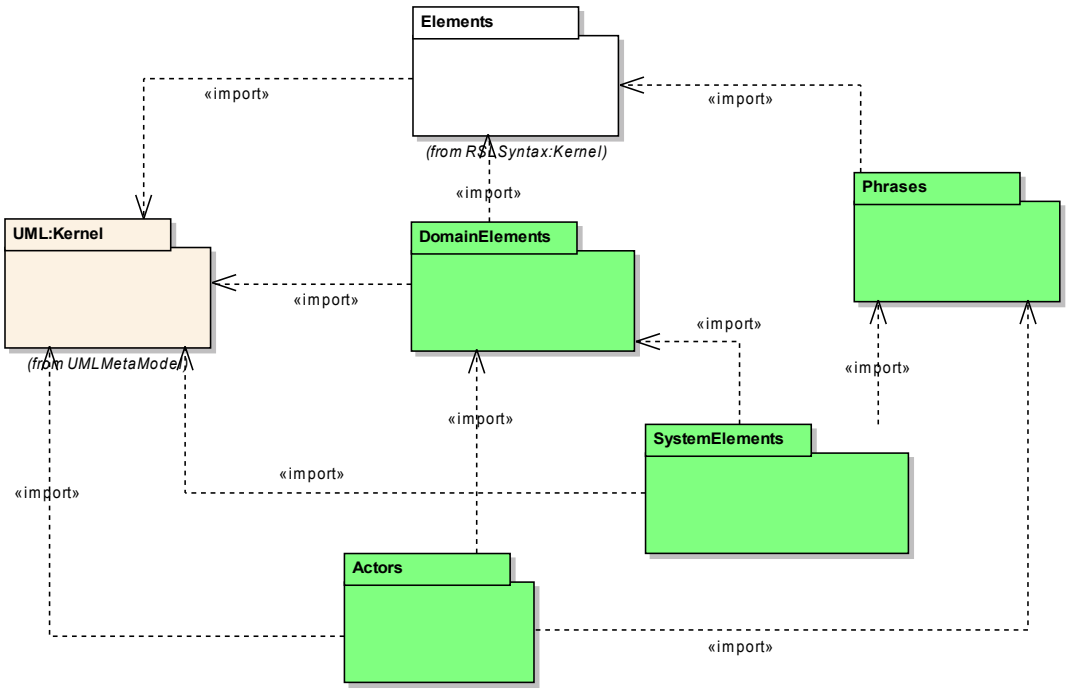


Figure 14.3: Overview of packages inside the DomainEntities part of RSL

sentence in constrained language. It also «import»s from Kernel :: Elements and Terms in order to define hyperlinks.

- The DomainElementRepresentations package supplies the language with definitions of representations for all kinds of domain elements. These are textual representations which contain sets of sentences with hyperlinks to phrases. Meta-classes from DomainElementRepresentations package extends abstract meta-classes from Kernel :: Elements.

- The DomainElements package supplies the language with abstract, high level constructs for elements which have their representations separated. These elements have names, and their representations contain sets of sentences. The names can contain hyperlinks to terms or phrases. This package also contain definitions for relationships that can exist between domain elements. Meta-classes from DomainElementRepresentations package extends abstract meta-classes from Kernel :: Elements as well as from UML :: Kernel.

- The Notions package contains definitions of notion elements which are part of domain vocabulary. Notions can express all real-world objects or entities from the problem domain of the system that requirements specification pertains to. Classes defined in this package extends more general classes from Kernel :: Elements and DomainElements. Notions package also «imports»s from DomainElementRepresentations, Phrases and Terms in order to use constructs defined in those packages.

- The Actors package allows for defining actors as part of the domain vocabulary. There can be shown relationships between actors. Actors are representable, and can have descriptions in hyperlinked text. Elements in this package extends general elements from DomainElements and UML :: Kernel. It also «import»s constructs from Phrases.
- The SytemElements package adds to the vocabulary the possibility to express the system and its general components. This does not allow for designing the system but allows for showing those elements of the system that might be used inside requirements specifications. Elements in this package extends general elements from DomainElements and UML :: Kernel. It also «import»s constructs from Phrases.

Generally, the vocabularies defined through our language consist of RepresentableElements which have names and HyperlinkedSentences as descriptions. Since these HyperlinkedSentences may contain Hyperlinks, RerpesentableElements resp. their descriptions that are related to each other are logically connected.

RepresentableElements can be actors, system components, special actions or entities that are domain-related, but not part of the system under development. Hence, the class RepresentableElement is a base class for some more special classes such as Actor, SystemComponent, and DomainElement. Everyone of these classes derived from RepresentableElement may have special associations to other RepresentableElements, for instance the DomainElement called wristband in the fitness-club is associated with the Actor customer as every customer wears a wristband. These associations are modelled with the class DomainElementAssociation and derived ones.

14.2 Domain elements

14.2.1 Overview

This package describes the general structure of domain elements as part of RequirementsSpecifications :: RequirementsSpecification. It consists of DomainSpecification class that defines the top level element holding a whole collection of DomainElements for a specific system grouped in DomainElementsPackages.
DomainSpecification and DomainElementsPackages can be presented in Package Diagrams that have their syntax derived from UML Package Diagrams. DomainElements are presented in Notions :: Notion Diagrams, SystemElements :: SystemElement Diagrams or in Actors :: Actor Diagrams, while DomainElement is abstract and superclass for these classes. DomainElements

can be related through DomainElementRelationship. This relationship can be constrained by DomainElementMultiplicity. All these elements can be placed in the Project Tree.

14.2.2 Abstract syntax and semantics

Abstract syntax for the DomainElements package is described in Figures 14.4, 14.5, 14.6.

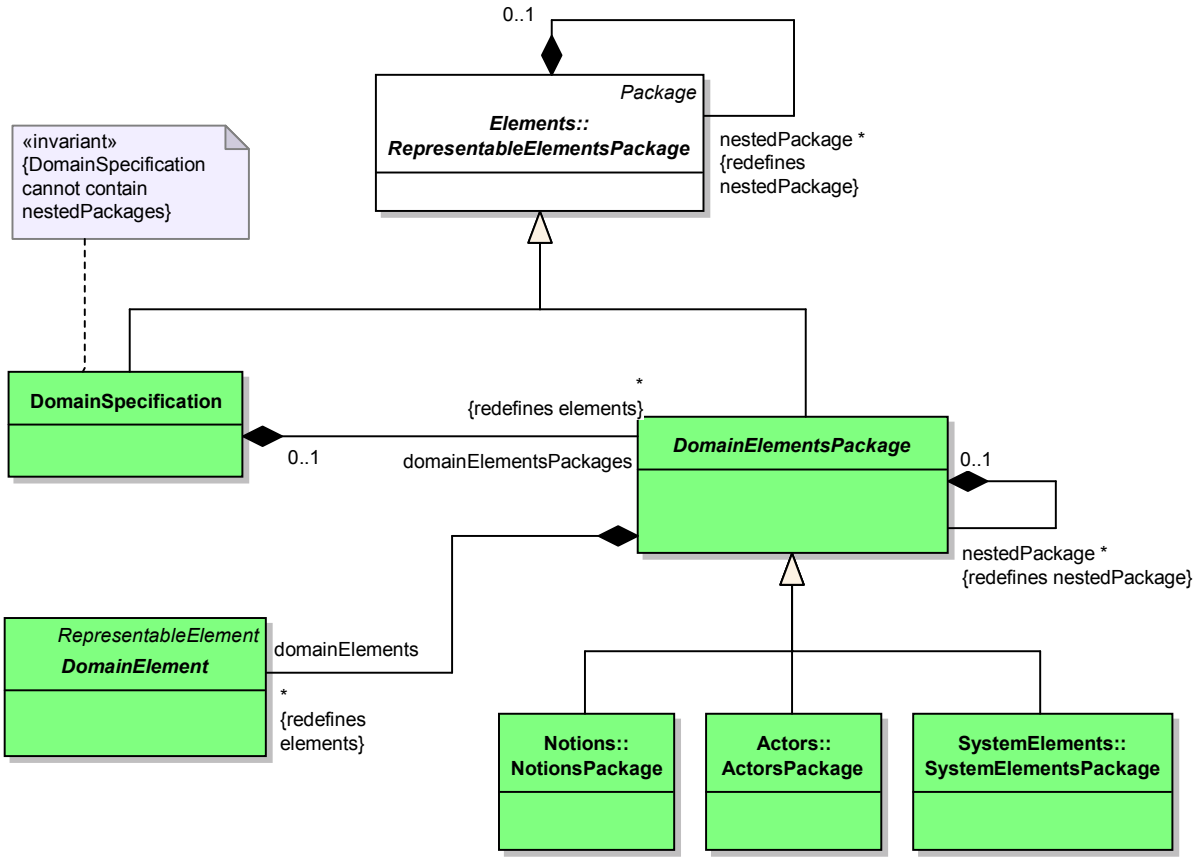


Figure 14.4: DomainSpecification

DomainSpecification

Semantics. DomainSpecification is a type of UML Package, i.e. a structure that groups elements and constitutes a container for these elements. It can contain only DomainElements, which specialisations are: Notions :: Notion, SystemElements :: SystemElement, Actors :: Actor. DomainElements have to be grouped in DomainElementsPackages' subclasses. DomainSpecifications is specific for only one RequirementsSpecification.

Abstract syntax. DomainSpecialisation is a kind of Elements :: RepresentableElementsPack-

age. It cannot contain any nested packages. It redefines elements with domainElementsPackages. It can contain many DomainElementsPackages.

DomainElement

Semantics. DomainElement denotes an abstract element of DomainSpecification. DomainElements are groupped in packages. They have its own representation, which is specific for all elements of domain. DomainElements can be related.
Abstract syntax. DomainElement is a specialisation of Elements :: RepresentableElement and is a superclass for Notions :: Notion, SystemElements :: SystemElement and Actors :: Actor. It is a component for DomainElementsPackage. DomainElement includes its own representation – DomainElementRepresentations :: DomainElementRepresentation with redefinition of representation. DomainElements can be related by DomainElementRelationship. DomainElement is a source and a target for this relation.

DomainElementsPackage

Semantics. DomainElementsPackage is a type of Elements :: RepresentableElementsPackage. It is an abstract package grouping DomainElements as well as nested DomainElementsPackages. It is a part of DomainSpecification.
Abstract syntax. DomainElementsPackage is a specialisation of Elements :: RepresentableElementsPackage. It redefines elements from the superclass with domainElements. It is composite for DomainElements. It also redefines nestedPackage, which can only be another DomainElementsPackage. Every DomainElementsPackage can be component of a DomainSpecification. DomainElementsPackage is a superclass for Notions :: NotionsPackage, SystemElements :: SystemElementsPackage and Actors :: ActorsPackage.

DomainElementRelationship

Semantics. DomainElementAssociation denotes relationships between two DomainElements. The relationship can be constrained by bounds of multiplicity.
Abstract syntax. DomainElementRelationship is a kind of Elements :: RepresentableElementRelationship. It connects two DomainElements by redefining source and target. This relationship is directed. Source of the relationship has to be different than its target – DomainElemen-

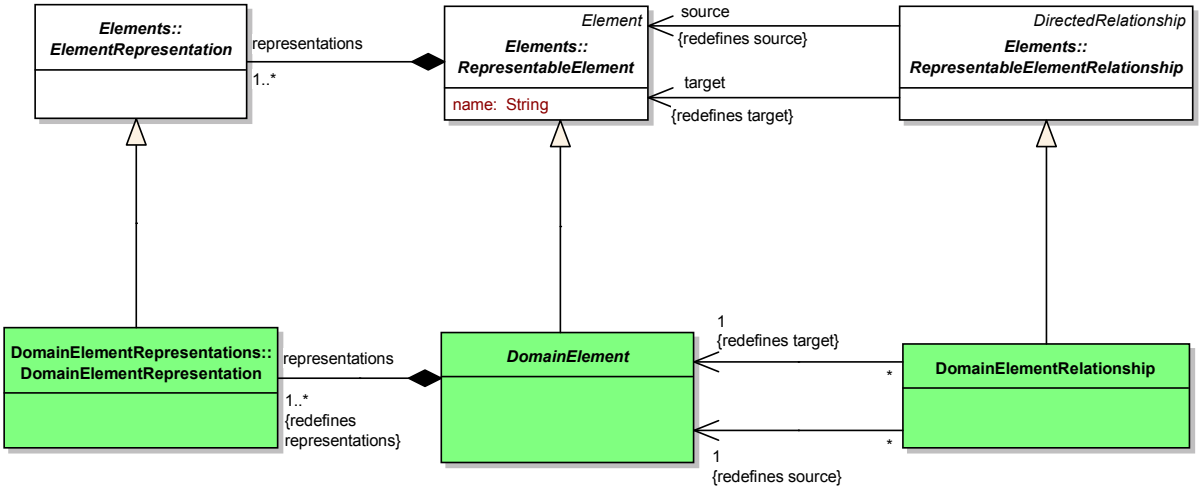


Figure 14.5: Relationship of domain elements

tAssociation cannot be associated with itself. DomainElementAssociation can have constrained multiplicity described by sourceMultiplicity and targetMultiplicity which are c.

DomainElementMultiplicity

Semantics. DomainElementMultiplicity is a type of UML :: Kernel :: MultiplicityElement. It is a definition of optional DomainElementRelationship's attributes for defining the bounds of a multiplicity.
Abstract syntax. DomainElementMultiplicity is a specialisation of UML :: Kernel :: MultiplicityElement. It is component of DomainElementRelationship in two roles: sourceMultiplicity and sourceMultiplicity.

14.2.3 Concrete syntax and examples

DomainSpecification. The concrete syntax is similar to UML :: Kernel :: Package, described in the UML Superstructure (in [Obj05b], paragraph 7.3.37, page 104): "A package is shown as a large rectangle with a small rectangle (a 'tab') attached to the left side of the top of the large rectangle. (...) Members may also be shown by branching lines to member elements, drawn outside the package. A plus sign (+) within a circle is drawn at the end attached to the namespace (package). (...)" In addition to the above UML :: Kernel :: Package description, name of DomainSpecification package is inside rectangle situated in the center of the large rectangle. It can also be presented in a tree structure with a minimized icon. See Figure 14.7 for examples

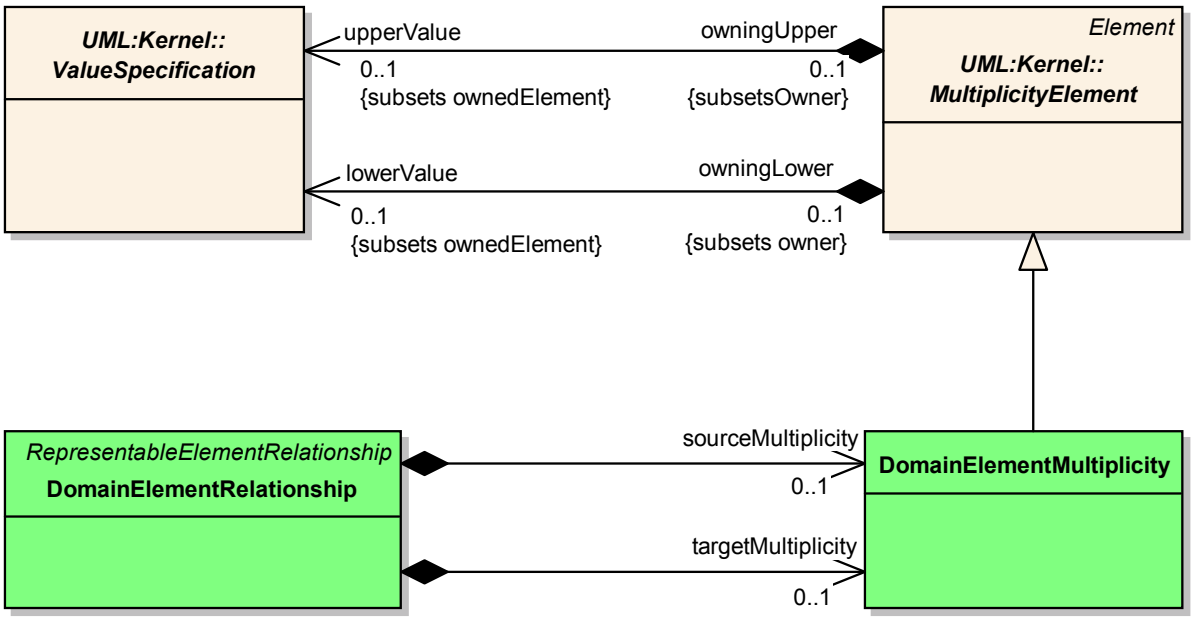


Figure 14.6: Multiplicities of domain elements' relationships

of concrete syntax in a Package Diagram and in a Project Tree structure with a minimized icon, respectively.

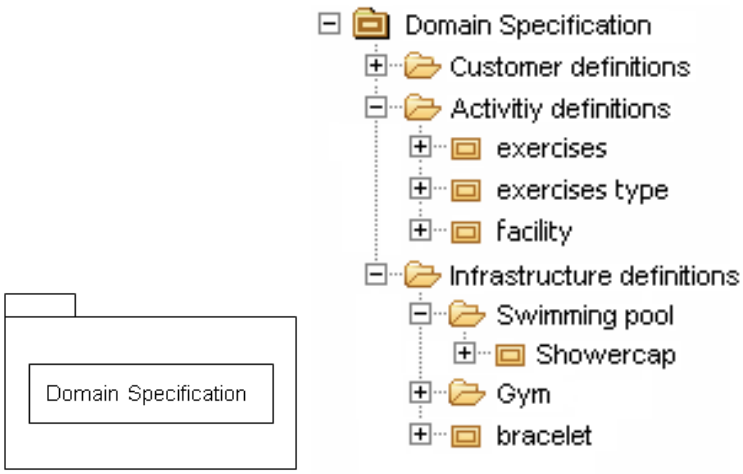


Figure 14.7: DomainSpecification example, normal and tree view

DomainElement. As an abstract meta-class, DomainElement does not have a concrete syntax.

DomainElementsPackage. As an abstract meta-class, DomainElementsPackage does not have a concrete syntax.

DomainElementRelationship. This class is presented as a line connecting two DomainElements' specialisations (Notion, SystemElement, Actor). The line can be directed from a source

to a target. See Figures 14.11, 14.12 for examples of concrete syntax in a Domain Element Diagram.

DomainElementsMultiplicty. The concrete syntax is similar to UML :: Kernel :: MultiplicityElement, described in the UML Superstructure (in [Obj05b], paragraph 7.3.32, page 90): "the notation will include a multiplicity specification shown as a text string containing the bounds of the interval, and a notation for showing the optional ordering and uniqueness specifications." See Figures 14.11, 14.12 for examples of concrete syntax in a Domain Element Diagram.

14.3 Notions

14.3.1 Overview

Notions package elements are extension of concepts from DomainElements (see section 14.2). The new concepts allow linking vocabulary elements through generalisations and attaching attributes to them. The role of core element of this package, a Notion, is to group all Phrases :: Phrases with Terms :: Noun representing given notion in domain vocabulary.

14.3.2 Abstract syntax and semantics

Notion

Semantics. A Notion is the core element of this package, which extends DomainElement by allowing generalising elements of vocabulary elements and assigning them properties (NotionAttributes). Notion is grouping all Phrases :: Phrases with Terms :: Noun representing given notion in domain vocabulary. This noun gives a name to this notion.
Abstract syntax. A Notion is the kind of DomainElement and UML :: Kernel :: Package. It consists of DomainStatements and NotionAttributes. A name of a Notion is redefined by Terms :: Noun linked by NounLink. Notion can be source and target of NotionGeneralisation. Notion is a component of NotionPackage. It is also a superclass for UIElements :: UIElement, UIElements :: InputOutputDevice and UIElements :: InputOutputType. Notion can be linked with RequirementsSpecifications :: Requirements by RequirementRelationships :: VocabularyRequieremenRelationship.

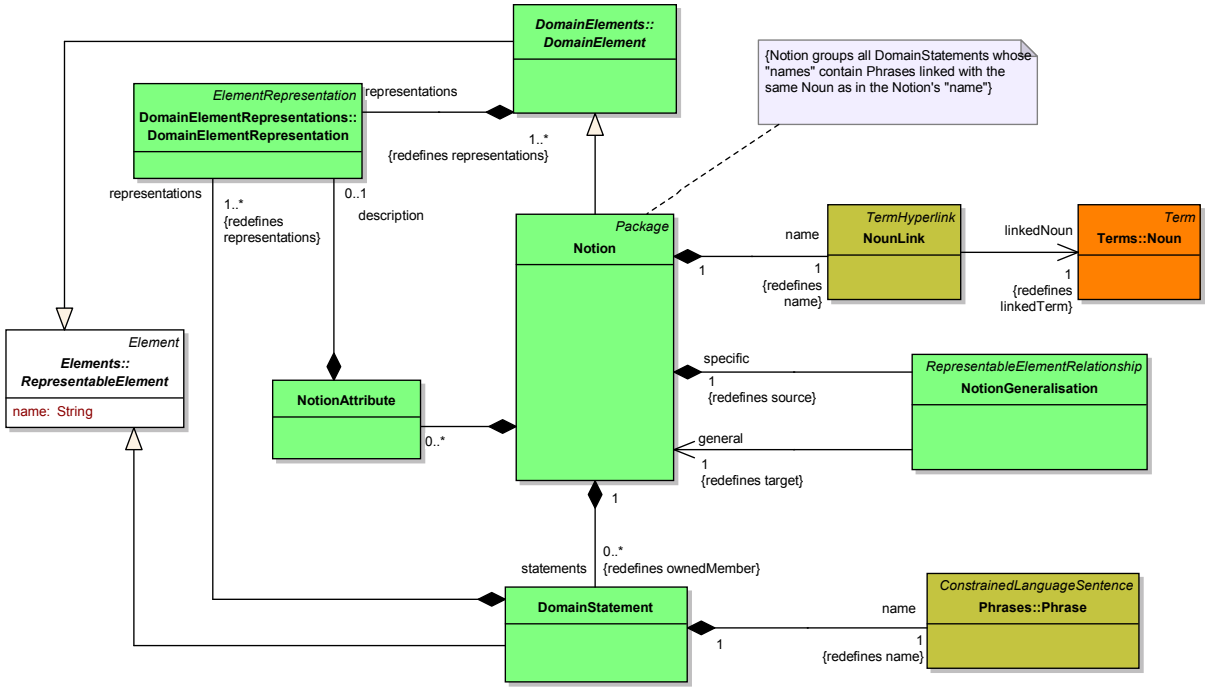


Figure 14.8: Notions

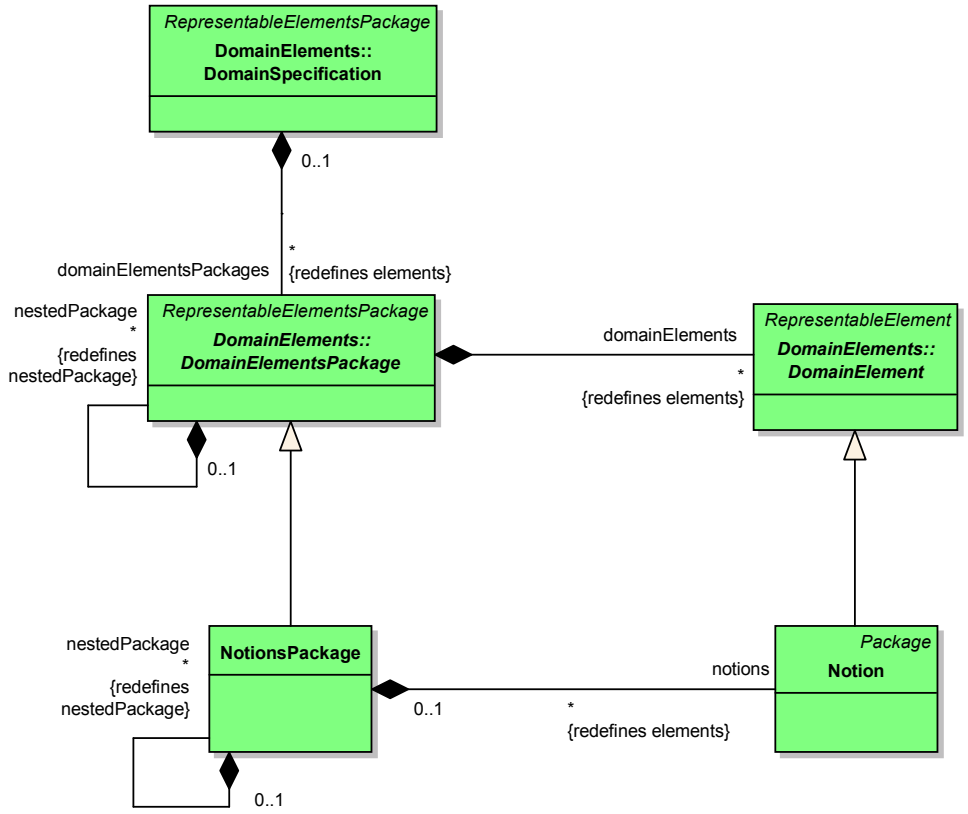


Figure 14.9: NotionsPackages

DomainStatement

Semantics. DomainStatement is a wiki-like description of an element of the domain of the system to be developed with its context - noun with modifiers, verbs and other nouns. DomainStatements are grouped in Notion in a role of statements, which are forming a container for all Phrases :: Phrases related to Notion that this statements are component of.
Abstract syntax. DomainStatement is kind of Kernel :: RepresentableElement. The inherited name attribute is redefined by Phrases :: Phrase. It consists of DomainElementRepresentations. DomainStatement is associated with Phrases :: PhraseHyperlink.

NotionAttribute

Semantics. A NotionAttribute is used for attaching Notion specific properties to domain vocabulary notions. Those properties have descriptions represented similarly to DomainElements.
Abstract syntax. A NotionAttribute is a component of a Notion and consists of DomainElementRepresentation as its description.

NotionGeneralisation

Semantics. A NotionGeneralisation allows parent-child relationships between two Notions: a more general Notion and more specific Notion.
Abstract syntax. A NotionGeneralisation is a kind of RepresentableElementRelationship. It links two Notions - specific and general.

NotionPackage

Semantics. A NotionPackage is used for grouping Notions in packages, which can be included in requirements specification.
Abstract syntax. NotionPackage is a kind of DomainElements :: DomainElementsPackage. It can have other NotionPackages included in it. NotionPackage consists of Notions.

NounLink

Semantics. A NounLink is a entity hyperlinking Notion to Terms :: Noun and therefore giving Notion a name.
Abstract syntax. NounLink is a kind of Terms :: TermHyperlink that, as a component of Notion, links it with Terms :: Noun.

14.3.3 Concrete syntax

Notion. The basic representation of Notion is denotated by a rectangle with its name inside it (see Figure 14.11). Another form of representation is a rectangle divided into three parts by two horizontal lines. The name of Notion is placed in the upper part. The middle part includes hyperlinked names of DomainStatements, each in its own rectangle (see Figure 14.12). The bottom part contains list of NotionAttrbiutes attached to the Notion (see Figure 14.14). Notions can be presented in diagram as both of their forms of representation. Notion can also be presented as a tree structure with a minimized icon. An example of concrete syntax for the tree view of Notion can be found in Figure 14.10.

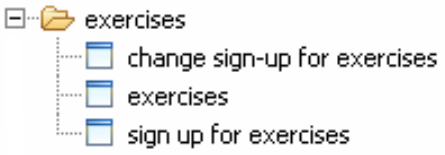


Figure 14.10: Notion's tree view example

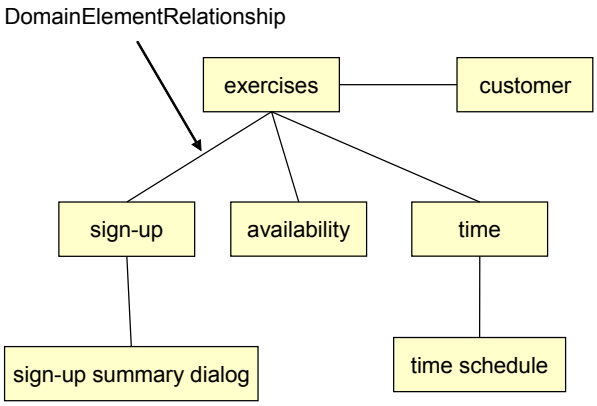


Figure 14.11: Notion's diagram example - notions and their associations

DomainStatement. Concrete syntax includes the name of DomainStatement as a hyperlink to Phrases :: Phrase or one of its subclasses and description as RequirementRepresentations :: NaturalLanguageHypertext. It can be represented in the form of a "source" or "view" of wiki-

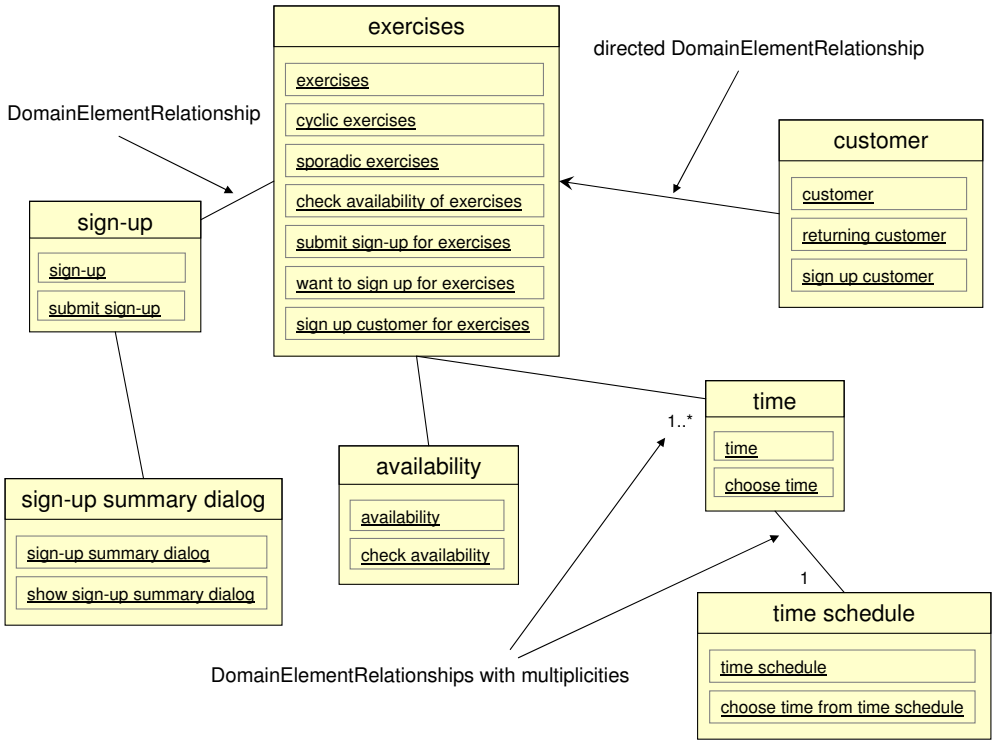


Figure 14.12: Notion's diagram example - extended view of notions

Wiki-like source

```
[[v:sign up n:customer p:for n:exercises]]

[[n:Customer]]'s interaction, when customer
sign up for exercises chosen from
[[m:available n:exercises list]].
```

Wiki-like view

sign up : customer : for : exercises

Customer's interaction, when customer sign up for exercises chosen from available : exercises list.

Figure 14.13: DomainStatement example

like hyperlinked sentence. Example of concrete syntax of DomainStatement can be found in Figure 14.13.

NotionAttribute. It can be presented along with notion it is attached to, in the bottom notion's compartment which is separated from phrases compartment by horizontal line. Attributes are listed in a form of

attribute name1 = value1
attribute name2 = value2
...
attribute nameN = valueN

where “attribute name” is a hyperlink (see Figure 14.14).

NotionGeneralisation. The concrete syntax is similar to UML :: Kernel :: Generalisation. NotionGeneralisation is shown as a line with hollow triangle as an arrowhead between symbols representing involved Notions. The arrowhead points to the symbol representing the general Notion – see Figure 14.14 (based on [Obj05b], p. 68).

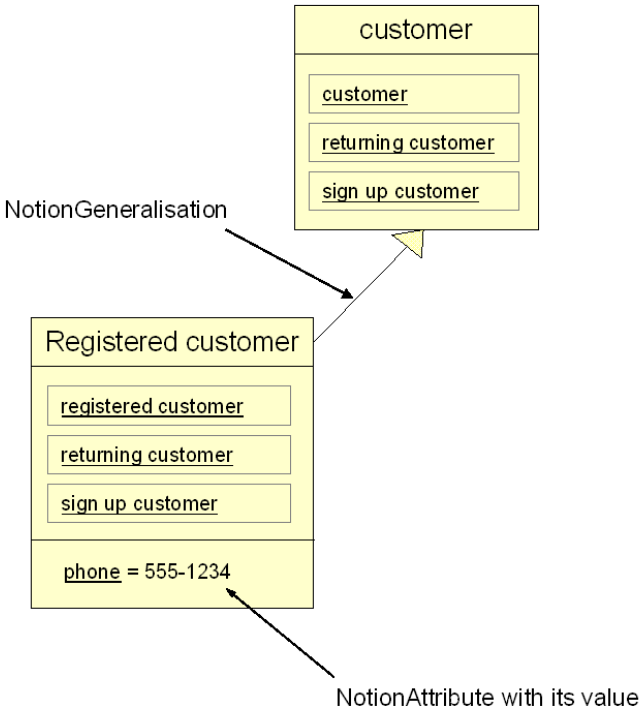


Figure 14.14: Notion’s diagram example - attributes and generalisations

NotionPackage. The concrete syntax of this class is inherited from UML :: Kernel :: Package (see [Obj05b], p. 104). In addition NotionPackages can be presented in a tree-view form (see Figure 14.10).

NounLink. Concrete syntax is inherited from the Elements :: Hyperlink meta-class.

14.4 System elements

14.4.1 Overview

This package contains the part of the RSL meta-model that deals with the representation of those domain elements that are not actors. If the system under development is the fitness club software system, its system elements are for instance “terminal” or “reception desk”.

14.4.2 Abstract syntax and semantics

The diagrams in Figures 14.15 and 14.16 describes the part of RSL that is related to the representation of the composite system under development. The classes introduced in this Figures are described in the following sections.

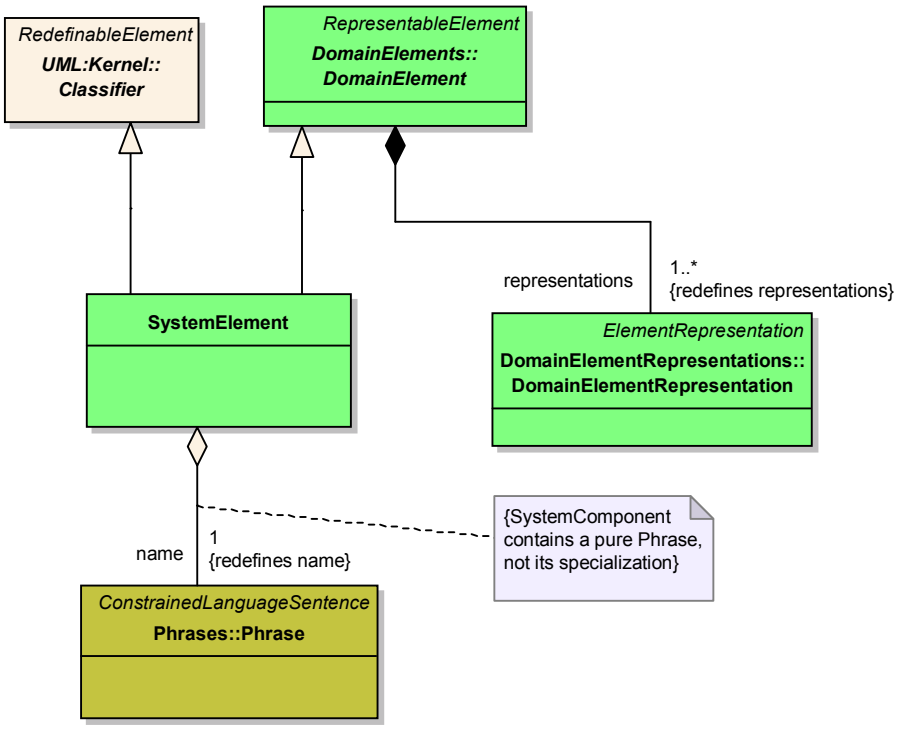


Figure 14.15: System elements

SystemElement

Semantics. This class is the most important class in this package. Every part of the application domain of the composite system that is referred to in the requirements specification as a “black

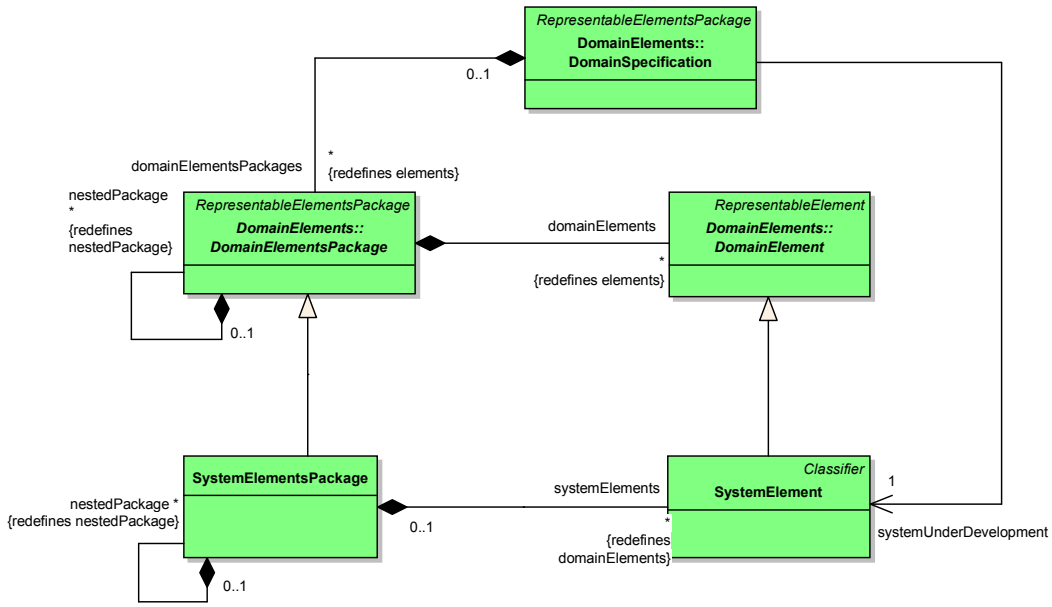


Figure 14.16: System package

box" is modelled as an instance of SystemElement. If requirements for a fitness club system are specified, system elements may for instance be "terminal", "database", or "reception computer". System components can be referred in functional and non-functional requirements.

Abstract syntax. The class SystemElement is derived from the classes Classifier from the UML 2.0 Superstructure and the class DomainElements :: DomainElement. The aggregation toElements :: HyperlinkedSentence, which specifies the name of an Elements :: Element, is redefined, thus the name of a SystemElement may only be a Phrases :: Phrase. The constraint is added to this redefined aggregation because a system's name should be for example "terminal" but not a Phrases :: VerbPhrase like "take".

SystemElementsPackage

Semantics. A SystemElementsPackage is used to group SystemElements in the requirements specification.

Abstract syntax. SystemElementsPackage is a kind of DomainElements :: DomainElementsPackage, it may contain other SystemElementsPackages as nestedPackages. Further, a SystemElementsPackage may contain SystemElements by the redefined aggregation derived from DomainElement :: DomainElementsPackage.

14.4.3 Concrete syntax

SystemElement. A SystemElement occuring in an interaction or a use case representation is depicted as a rectangular UML object (see Figure 14.17). The Phrases :: Phrase that defines the name of the SystemElement is written in the rectangle. If a SystemComponent is refered to in a textual description, it is represented only by the Phrases :: Phrase that defines its name.

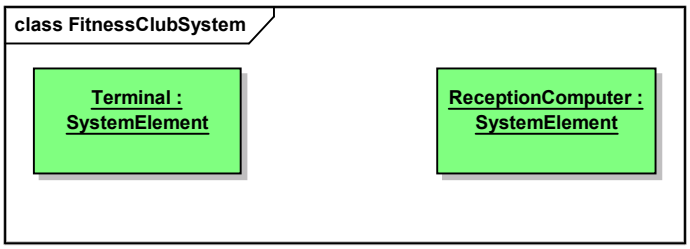


Figure 14.17: The concrete syntax of system elements and coresponding packages.

SystemElementsPackage. The concrete syntax of a SystemElementsPackage is similar to the concrete syntax of packages in UML 2.0, as illustrated by the package in Figure 14.17.

14.5 Actors

14.5.1 Overview

This package contains that part of the RSL metamodel that deals with the representation of actors in the requirements specification. Actors are for instance “customer” or “fitness club employee”, they can be refered in every type of requirement representation.

14.5.2 Abstract syntax and semantics

The diagrams in Figures 14.18 and 14.19 describe the RSL part that is related to actors and packages containign actors. The two classes Actor and ActorsPackage which are introduced in this Figure are described in the following sections.

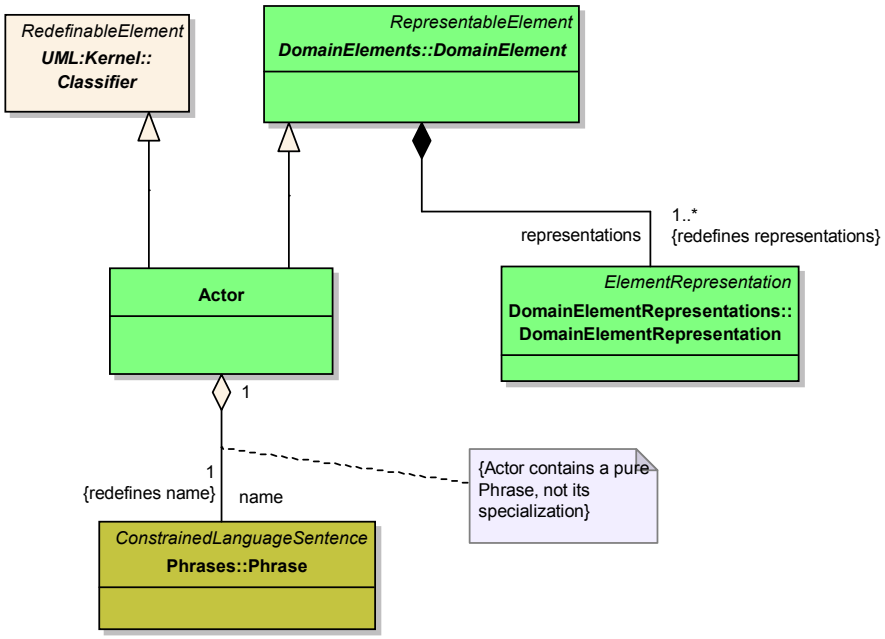


Figure 14.18: Actor metamodel part

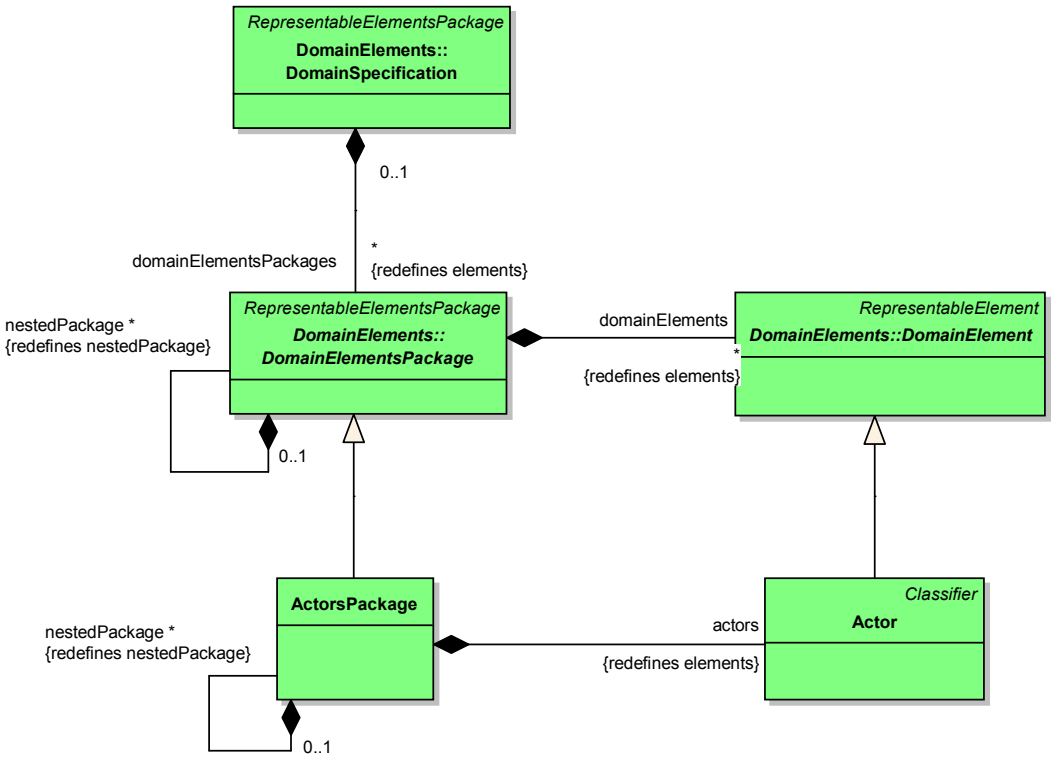


Figure 14.19: Actors package metamodel part

Actor

Semantics. This class is the most important class in this package. Every actor that is referred to in the requirements specification is modelled as an instance of Actor. If requirements for a fitness club system are specified, actors may for instance be "customer", "system administrator", or "staff member". Actors participate in scenarios and use cases on the one hand, but they may also be referred to in other functional and even non-functional requirements.
Abstract syntax. The class Actor is derived from the classes Classifier from the UML 2.0 Superstructure and the class DomainElements :: DomainElement.. The aggregation to Elements :: HyperlinkedSentence, which specifies the name of an Elements :: Element is redefined, thus the name of an Actor may only be a Phrases :: Phrase. The constraint is added to this redefined aggregation because an actor's name should be for example "a customer" but not a Phrases :: VerbPhrase like "take".

ActorsPackage

Semantics. A ActorsPackage is used to group Actors in the requirements specification.
Abstract syntax. ActorsPackage is a kind of DomainElements :: DomainElementsPackage, it may contain other ActorsPackages as nestedPackages. Further, a ActorsPackage may contain Actors by the redefined aggregation derived from DomainElement :: DomainElementsPackage.

14.5.3 Concrete syntax

Actor. An Actor occurring in an interaction or a use case representation is depicted as a stylised stick figure (see Figure 14.20), though not only a person can be an actor, but also external software systems interacting with the system in development. The actor's 'name' is written below the stick figure.

ActorsPackage. Analogous to DomainElementsPackage and SystemElementsPackage the concrete syntax of a SystemPackage is similar to the concrete syntax of packages in UML 2.0, as illustrated by the package in Figure 14.20.

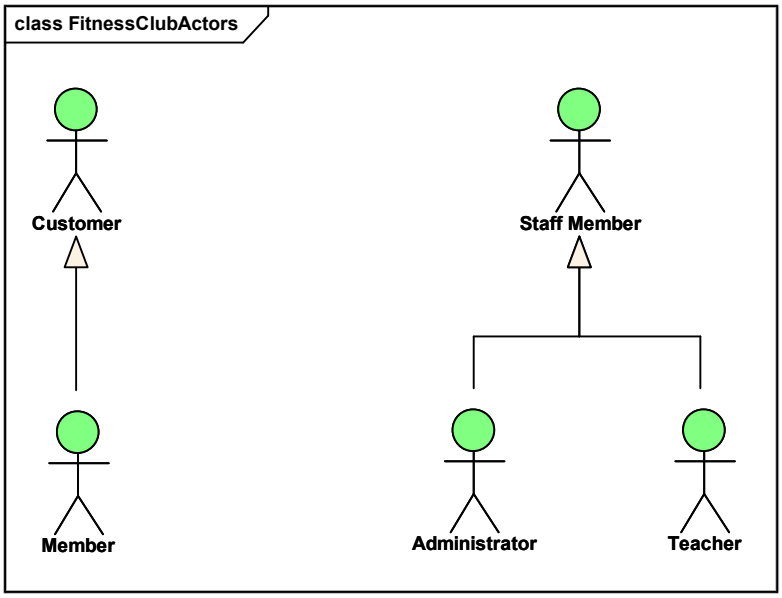


Figure 14.20: The concrete syntax of actors and actors packages.

14.6 Domain element representations

14.6.1 Overview

The package DomainElementRepresentations contains the classes DomainElementRepresentation and DomainElementHyperlinkedSentence that model the textual representation for all DomainElements :: DomainElements like Actors :: Actors, Notions :: Notions and SystemElements :: SystemElements.

14.6.2 Abstract syntax and semantics

The diagram in Figure 14.21 shows the abstract syntax of the classes in the package DomainElementRepresentations. The following sections describe semantics and abstract syntax for these classes.

DomainElementRepresentation

Semantics. This is a textual representation for all DomainElements :: DomainElements (Actors :: Actors, Notions :: Notions and SystemElements :: SystemElements) in form of wiki-like de-

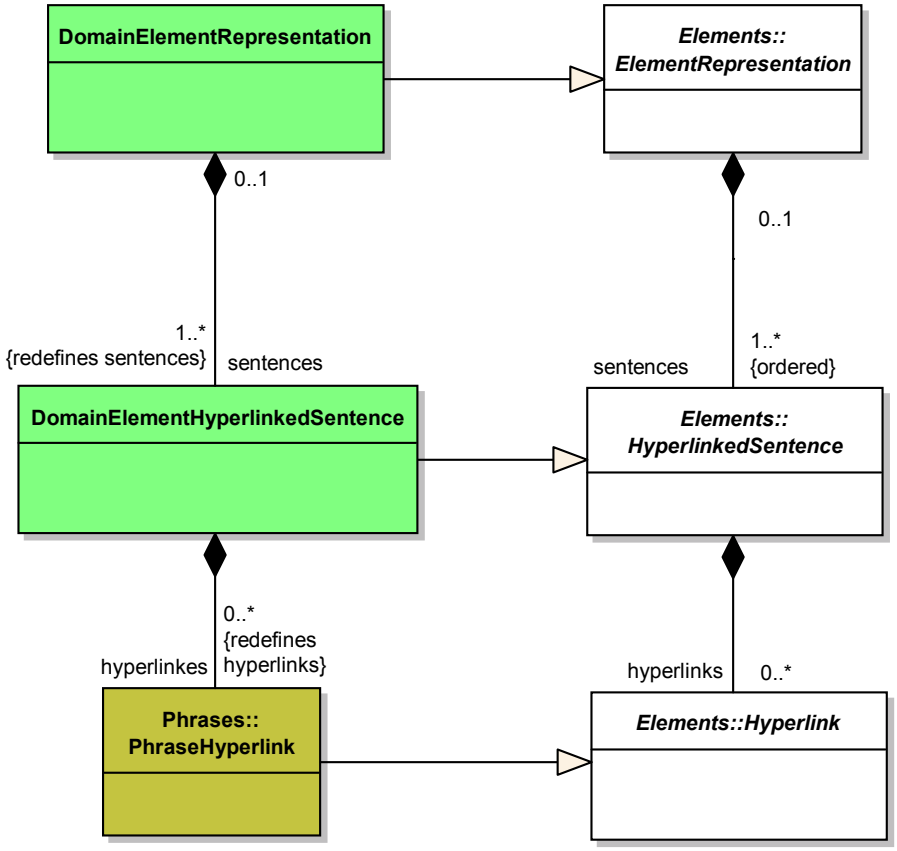


Figure 14.21: Domain element representations

scription (it can contain a set of hyperlinked sentences). DomainElementRepresentation can also build up a textual description of Notions :: NotionAttributes.

Abstract syntax. DomainElementRepresentation is concrete specialisation of abstract Elements :: ElementRepresentation. It redefines sentences derived from its superclass with DomainElementHyperlinkedSentence. Every DomainElements :: DomainElement contains at least one DomainElementRepresentation as its representations. DomainElementRepresentation can be also contained in Notions :: NotionAttributes as its description.

DomainElementHyperlinkedSentence

Semantics. A DomainElementHyperlinkedSentence is used in a description of a DomainElements' subclasses. DomainElementHyperlinkedSentence uses wiki-like hyperlinks in the sentence, pointing to other vocabulary elements. If the sentence does not contain any Hyperlink, it is simply free text.

Abstract syntax. DomainElementHyperlinkedSentence is kind of Elements :: HyperlinkedSentence. It redefines hyperlinks with Phrases :: PhraseHyperlink. It is also aggregated by DomainElementRepresentation in role of sentences.

RepresentableElement

Semantics. Every entity that is related to the system under development is represented by a RepresentableElement and has a name. Since such elements can be represented in different ways, every element has at least one representation.
Abstract syntax. RepresentableElement is the abstract base class for all elements related to the system under development that are represented in the requirements specification, such as RequirementSpecification :: Requirement, DomainElements :: DomainElement or Actors :: Actor. Every RepresentableElement has a name, which is a BasicRepresentations :: HyperlinkedSentence, so it may contain BasicRepresentations :: Hyperlinks that refer to Phrases :: Phrases and Terms::Terms in the terminology. In addition to the name, the RepresentableElement is represented by at least one ElementRepresentation, as the aggregation between these two classes indicates.

ElementRepresentation

Semantics. Every ElementRepresentation is one possible representation of a RepresentableElement. Due to this, a RepresentableElement may contain one or more representations. All those representations in the requirement specification, for instance InteractionRepresentations :: ActorLifeline, which is introduced in section 7.7 “ScenarioRepresentation” of D2.1 “Behavioural Requirements Language Definition”, are derived from ElementRepresentation.
Abstract syntax. The class ElementRepresentation is the base class for all representations such as e.g. BasicRepresentations :: RequirementsRepresentation or InteractionRepresentations :: ActorLifeline. The aggregation to BasicRepresentations :: HyperlinkedSentence shows that a ElementRepresentation contains BasicRepresentations :: HyperlinkedSentences, but as the way these aggregation is realized differes from representation to representation, it is redefined in most of them. These sentences are typically ordered and may be used to build up the textual description of the element the ElementRepresentation describes, but there are also other types of containment relations between ElementRepresentation and BasicRepresentations :: HyperlinkedSentence, for instance in the InteractionRepresentations :: InteractionScenario described in section 12.6.

14.6.3 Concrete syntax

RepresentableElement. ElementRepresentation. These classes are abstract, and they do not introduce any concrete syntax.

DomainElementRepresentation. DomainElementHyperlinkedSentence. DomainElementRepresentation is a description of DomainElement. Its concrete syntax depends on the context in which DomainElementRepresentation is presented to the user. It can be represented in the form of a purely textual "source", or in a "view" form of DomainElementHyperlinkedSentence with underlined wiki-like links. In source, DomainElementRepresentation consists of text with a double pair of square brackets ("[[]]") surrounding text to be hyperlinked in view mode. In view form, contained BasicRepresentations :: Hyperlinks are represented as coloured and underlined text (see Figure 14.22).

Source:	**View:**
[[Customer]]'s interaction, when customer signs up for exercises chosen from [[available exercises list]].	Customer's interaction, when customer signs up for exercises chosen from available exercises list.

Figure 14.22: DomainElementRepresentation's concrete syntax example

14.7 Phrases

14.7.1 Overview

The Phrases package contains language entities that allow for formulating phrases in a structured language. These Phrases represent Terms :: Nouns associated with other Terms (e.g. Terms :: Verbs). A generic Phrase is always put in the context of a Terms :: Noun and is (possibly) associated with a Determiner and/or Modifier. Another kind of Phrase is VerbPhrase which describes the context of a Verb.

14.7.2 Abstract syntax and semantics

The abstract syntax of this package is presented in figures 14.23 and 14.24.

Phrase

Semantics. A Phrase describes an expression involving a given Terms :: Noun.
Abstract syntax. Phrase is a kind of RepresentationSentences :: ConstrainedLanguageSentence. Phrase consists of an Object (a Terms :: TermHyperlink to a Terms :: Noun) and optionally of a Modifier or Determiner linking to Terms :: Modifier or Terms :: Determiner, respectively.

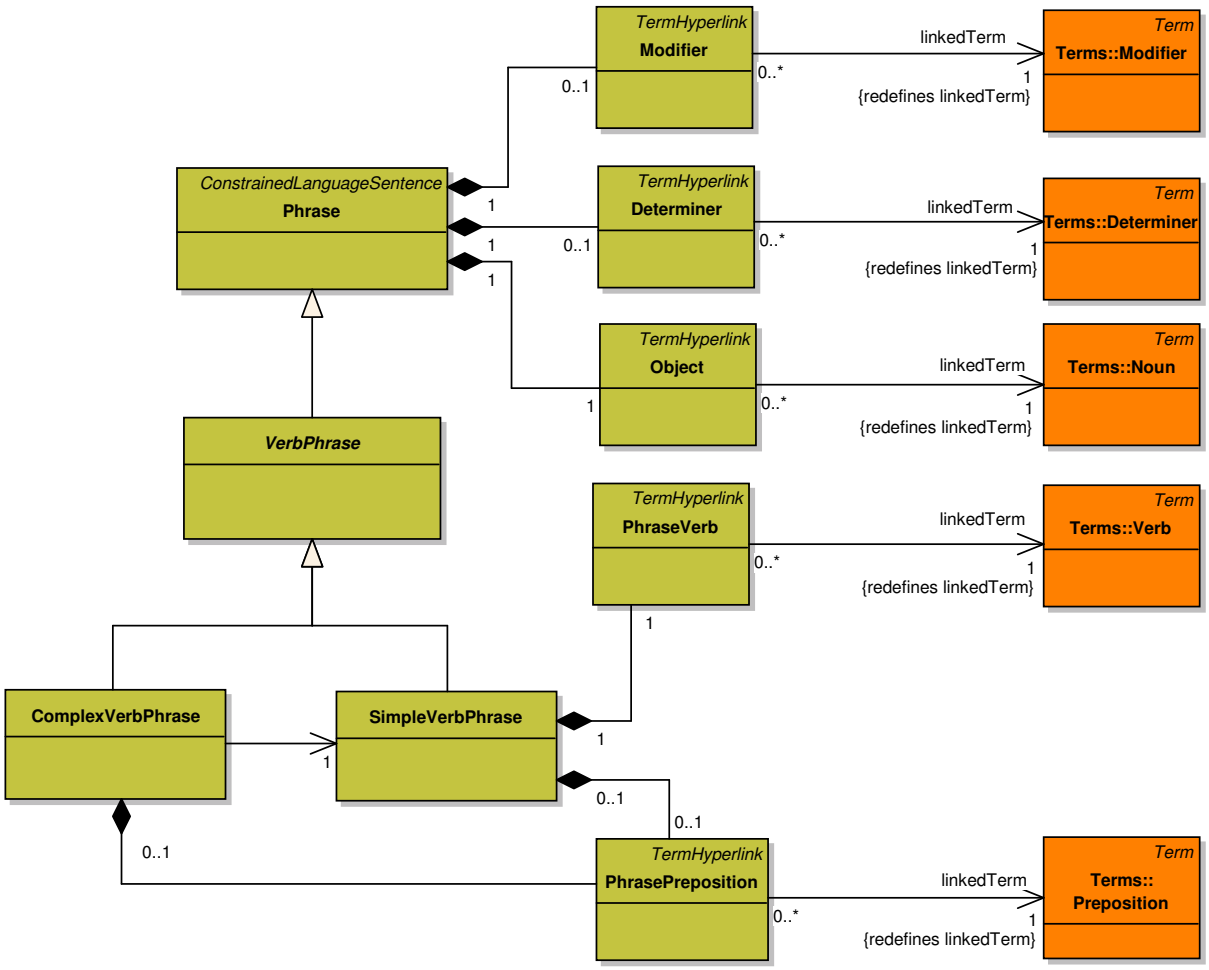


Figure 14.23: Phrases

A Phrase represents the "name" of a DomainElements :: DomainElement. Phrase contains a hyperlinked description. It is used for referencing the vocabulary of different requirement representations (controlled grammars, wiki-like descriptions).

VerbPhrase

Semantics. This expression describes an operation that can be performed in association with the Object described by a Terms :: Noun.

Abstract syntax. VerbPhrase is an abstract kind of Phrase. It exists in two concrete classes: SimpleVerbPhrase and ComplexVerbPhrase.

SimpleVerbPhrase

Semantics. SimpleVerbPhrase has the semantics of VerbPhrase and can be used as the **VO** part in an SVOSentences :: SVOSentence.
Abstract syntax. SimpleVerbPhrase, in addition to a Phrase, includes a PhraseVerb (a Phrases :: TermHyperlink to a Terms :: Verb). It may also contain a PhrasePreposition. SimpleVerbPhrase is a concrete subclass of VerbPhrase.

ComplexVerbPhrase

Semantics. ComplexVerbPhrase can be used as **VOO** (SVOSentences :: SVOSentence with direct and indirect object) part in an SVOSentences :: SVOSentence. ComplexVerbPhrase describes a behavioural relation between a direct and an indirect object.
Abstract syntax. ComplexVerbPhrase extends SimpleVerbPhrase with an additional Terms :: Noun (indirect object). It is a kind of VerbPhrase pointing to a SimpleVerbPhrase. It also includes a PhrasePreposition. ComplexVerbPhrase is a concrete subclass of VerbPhrase.

Modifier

Semantics. A Modifier combines with an Object and indicates how it should be interpreted in the surrounding context. In this way it creates a Phrase that distinguishes this Object's meaning from its main vocabulary entry.
Abstract syntax. Modifier is a kind of a Terms :: TermHyperlink. It points to the Terms :: Modifier used in a given Phrase.

Determiner

Semantics. A Determiner combines with Object and expresses their reference, e.g. "this" or "that". This includes quantity (e.g. "some", "a", "every", "two") and variability, that is the extent to which a Terms :: Noun holds over a range of things.
Abstract syntax. Determiner is a kind of a Terms :: TermHyperlink. It points to the Term :: Quantifier used in a given Phrase.

Object

Semantics. An Object points to the Terms :: Noun specific for this Object's Phrase.
Abstract syntax. Object is a kind of a Terms :: TermHyperlink.

PhraseVerb

Semantics. A PhraseVerb points to the Terms :: Verb specific for this Phrase.
Abstract syntax. PhraseVerb is a kind of a Terms :: TermHyperlink.

PhrasePreposition

Semantics. A PhrasePreposition points to the Terms :: Preposition used to connect a PhraseVerb with its direct and indirect Objects.
Abstract syntax. PhrasePreposition is a kind of a Terms :: TermHyperlink.

PhraseHyperlink

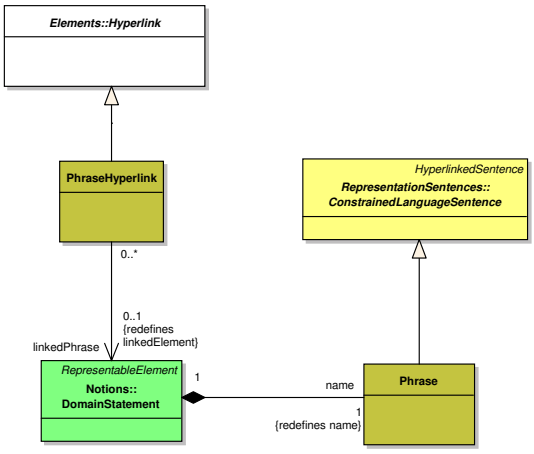


Figure 14.24: PhraseHyperlink

Semantics. PhraseHyperlink expresses a reference from a sentence in a DomainElementRepresentation to an element of the domain vocabulary. By using PhraseHyperlinks, domain vocabulary elements can be used in the content of DomainElementRepresentation without copying their names, but by pointing to their definitions.
Abstract syntax. PhraseHyperlink is a kind of Elements :: Hyperlink and can reference a single

Notion :: DomainStatement. It can be part of a DomainElementRepresentations :: DomainElementHyperlinkedSentence.

14.7.3 Concrete syntax and examples

Source:	View:
`[[n:customer]]`	customer
`[[m:registered n:customer]]`	registered : customer

Figure 14.25: Phrase concrete syntax examples

Source:	View:
`[[v:sign up n:customer]]`	sign up : customer

Figure 14.26: SimpleVerbPhrase concrete syntax examples

Source:	View:
`[[v:sign up n:customer p:for n:exercises]]`	sign up : customer : for : exercises

Figure 14.27: ComplexVerbPhrase concrete syntax examples

Object. Modifier. Determiner. PhraseVerb. PhrasePreposition. Their concrete syntax depends on the context in which they are presented to the user. They can be represented in source or view form. In the source form, they consist of the linked terms' names preceded by a letter with a colon (":") indicating the term type ("n:" for noun (Object), "m:" for Modifier, "d:" for Determiner, "v:" for PhraseVerb, "p:" for PhrasePreposition). In view form, they are represented as the linked terms' names separated by colons (see figures 14.25, 14.26, 14.27).

Phrase. SimpleVerbPhrase. ComplexVerbPhrase. These are represented by sequences of Terms :: TermHyperlinks. Their concrete syntax is described in the above paragraph (see Figures 14.25, 14.26, 14.27). Regular expressions for the order of the Terms :: TermHyperlinks in Phrases are depicted in figure 14.28

PhraseHyperlink. Its concrete syntax depends on the context in which PhraseHyperlink is presented to the user. It can be represented in the form of a purely textual "source", or in a "view" form of underlined wiki-like links. In source form, PhraseHyperlink consists of a double pair of square brackets ("[[]]") surrounding the hyperlinked text. In view form, PhraseHyperlink is represented as coloured and underlined text (see Figure 14.29).

VerbPhrase. As an abstract meta-class, VerbPhrase does not have a concrete syntax.

```
Phrase ::== [Determiner] [Modifier] Object
SimpleVerbPhrase ::== PhraseVerb [PhrasePreposition] [Determiner]
                      [Modifier] Object
ComplexVerbPhrase ::== SimpleVerbPhrase [PhrasePreposition]
                       [Determiner] [Modifier] Object
```

Figure 14.28: Regular expressions for Phrase and its subclasses. Optional elements are denoted by square brackets.

Source:

```
[[d:The n:Fitness Club]] a:should [[v:provide n:bracelets]].

c:If [[d:a n:customer]] [[v:signs up p:for d:a n:course]], [[d:the
n:system]] a:must [[v:bill d:this n:customer]].
```

View:

The : Fitness Club : should : provide : bracelets.

If : a : customer : signs up : for : a : course : , the : system : must : bill : this : customer.

Figure 14.29: PhraseHyperlink concrete syntax example

14.8 Terms

14.8.1 Overview

This package describes terms, its structure, inflection and terms' hyperlinks.

Terminology is the main package structure containing all Terms. Every Term has to have at least one Lexeme. Lexeme consists of its Lemma and Inflections. Such a structure allows for presenting Term in more than one national language. Terms are used by another parts of RSL through a TermHyperlink.

Every term is a distinguished part of speech. Note that "term" is not necessarily a single word (e.g. some modal verbs – see note on the Diagram 14.30). We also treat phrasal verbs as Verb class objects. Terms from Terminology are used for building Phrases.

14.8.2 Abstract syntax and semantics

Figure 14.30 shows the specialisation hierarchy of the different types of Terms, which are ConditionalConjunction, ModalVerb, Verb, Noun, Modifier, Determiner and Preposition.

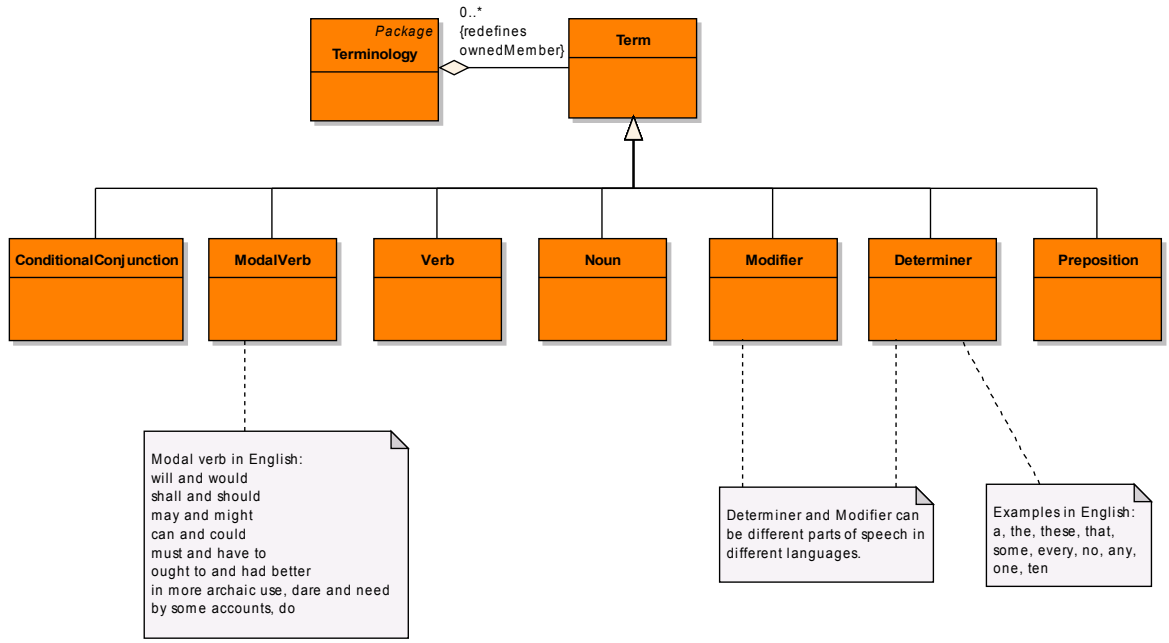


Figure 14.30: Term and its specialisations.

Figure 14.31 shows the structure for forms of the Term. This construction consists of Lexeme which includes only one Lemma and can have some Inflections.

Figure 14.32 shows the TermHyperlink and all its subclasses.

Terminology

Semantics. Terminology is a structure containing all the Terms, with their forms, inflections, cases etc. as well as their relations between each other (see section 14.9). These relations define the semantics of the Terms.
Abstract syntax. Terminology is a kind of Package. It contains Terms.

Term

Semantics. Term is a unit of language that native speakers can identify as a meaning-coherent notion. It is a block from which a phrase is made. A Term usually has different grammar forms. It can have forms for different national languages.
Abstract syntax. Term is a class, that contains at least one Lexeme. Terms are grouped in a Terminology.

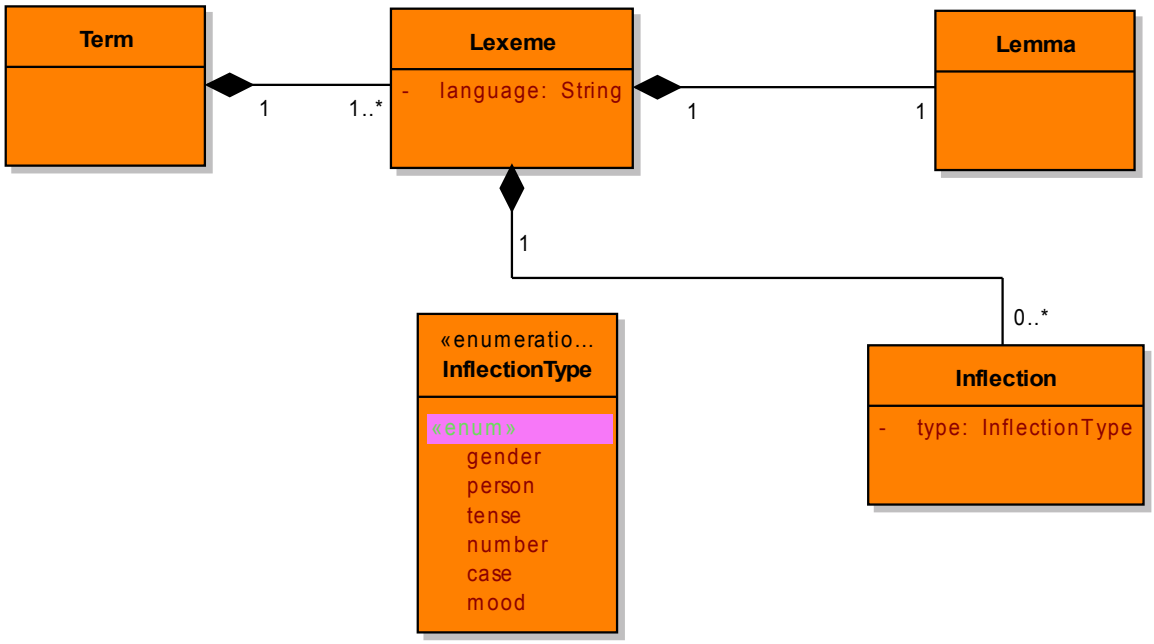


Figure 14.31: Inflections of the Term.

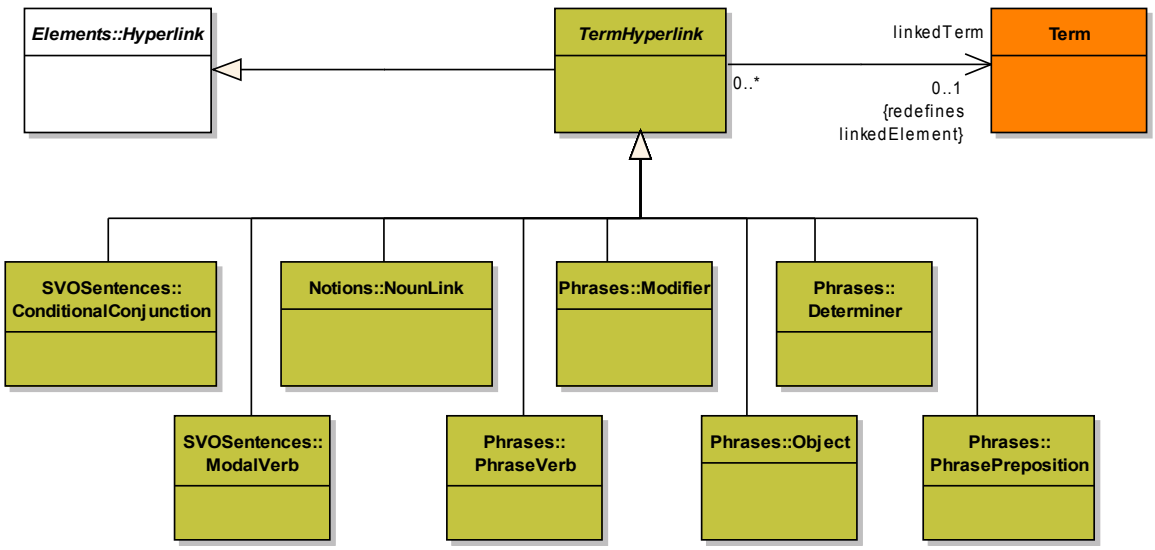


Figure 14.32: TermHyperlink and its subclasses

Lexeme

Semantics. Lexeme is an abstract unit of morphological analysis in linguistics. It is characteristic for national language. It can be understood as a set of forms, inflections of the same Term.
Abstract syntax. Lexeme is a component of Term. It consists of only one Lemma and some Inflections. This class has an attribute "language" of type String determining the national language of the Lexeme.

Lemma

Semantics. Lemma represents the canonical form of a Lexeme.
Abstract syntax. This class is a component of Lexeme. Lexeme can have only one Lemma.

Inflection

Semantics. Inflection is modification of a Term (or more precisely Lemma) reflecting a grammatical information. Grammatical form is constrained by InflectionType gender, tense, number, person, case, mood.
Abstract syntax. This class is a composite of Lexeme. Lexeme has at least one Inflection, the Lemma.

InflectionType

Semantics. InflectionType determines type of Term's inflection type. Possible types of inflections are: gender, tense, number, person, case, mood.
Abstract syntax. InflectionType is an enumerator which defines values: gender, tense, number, person, case, mood.

ConditionalConjunction

Semantics. A ConditionalConjunction is a Terminology element used for combining two sentences into a conditional or state descriptive structure.
Abstract syntax. ConditionalConjunction is a kind of a Term.

ModalVerb

Semantics. A ModalVerb (also modal, modal auxiliary verb, modal auxiliary) is a type of auxiliary verb that is used to indicate a provision of syntax that expresses the predication of an action, attitude, condition, or state other than that of a simple declaration of fact.
Abstract syntax. ModalVerb is a kind of Term.

Modifier

Semantics. A Modifier is a type of Term that combines with a Noun and indicates how it should be interpreted in the surrounding context. Modifier can be different parts of speech in different languages (e.g.Modifier in Lithuanian can be adjective, pronoun, pronoun+adjective, participle, pronoun+participle, in Polish it is an Adjective sourcing from a Noun).
Abstract syntax. Modifier is a kind of Term.

Determiner

Semantics. A Determiner is a type of Term that combines with a Noun and indicates how it should be interpreted in the surrounding context. This includes quantity and variability, that is the extent to which Noun holds over a range of things. Determiner can be different parts of speech in different languages (e.g. Determiner in Polish can be Zaimek (Pronoun)).
Abstract syntax. Determiner is a kind of Term.

Noun

Semantics. A Noun is a type of Term that names objects (e.g. a person, place, thing, quality, action or data).
Abstract syntax. Noun is a kind of Term.

Preposition

Semantics. A Preposition is a type of Term that combines with Phrases :: Phrases and indicates how they should be interpreted in the surrounding context.
Abstract syntax. Preposition is a kind of Term.

Verb

Semantics. A Verb is a type of a Term that expresses action or state of being.
Abstract syntax. Verb is a kind of a Term.

TermHyperlink

Semantics. A TermHyperlink denotes a reference to a Term. Using instances of the various subclasses of TermHyperlink, Terms can be used in Phrases, SVOSentences, Notions by pointing to them.
Abstract syntax. The abstract meta-class TermHyperlink is a kind of Elements :: Hyperlink. It can reference a single Term.

14.8.3 Concrete syntax and examples

Terminology. It is a semantic structure that holds Terms. This structure should allow for semantic-based reuse of organisation-specific Terms. The concrete syntax is equivalent to the concrete syntax of the DomainVocabulary (compare Figure 14.33 and Figure 14.7) which is based on Kernel :: Package. More important is the tree view presentation of the Terminology as

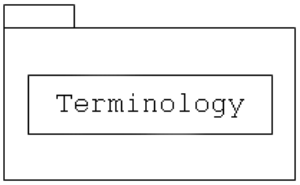


Figure 14.33: Package view: Terminology's concrete syntax.

depicted in Figure 14.34. Please note that the tree view depicts only the specialisation relations. Therefore relations between Terms are not shown.

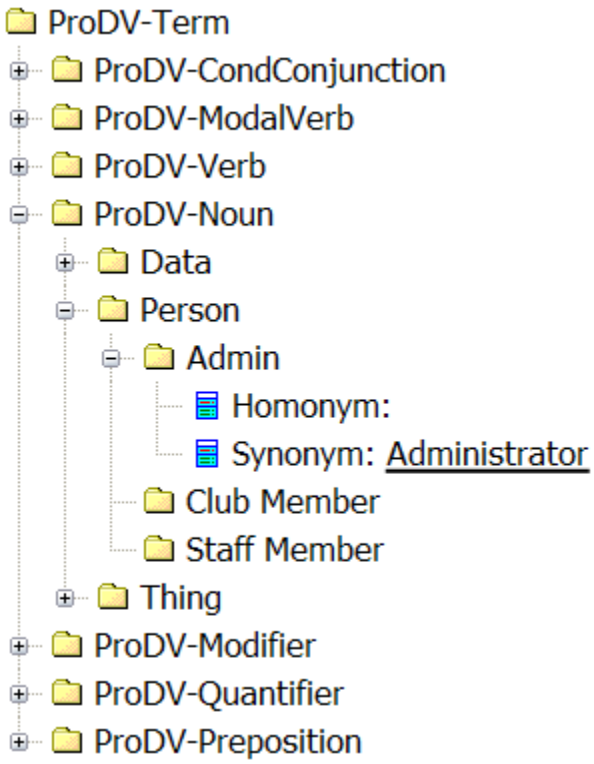


Figure 14.34: Terminology tree view example

Term. It is a string of letters and white spaces having a logical meaning in a specific natural language. The name of the Term is delivered by the Lexem. *Examples (for English):* "car", "buy", "look for", "buy ticket button", "at", "must", "every".

Lexeme. It is a string of letters and white spaces having a logical meaning in a specific natural language. The name of the Lexeme is delivered by the Lemma. Lexeme can be marked with the national language shortcut. *Examples (for English):* "EN: car", "EN: buy", "EN: look for", "EN: buy ticket button", "EN: at", "EN: must", "EN: every". *Examples (for Polish):* "PL: samochód", "PL: kupić", "PL: szukać", "PL: przycisk kupowania biletu", "PL: przy", "PL: musieć", ''PL: każdy".

Lemma. It is a string of letters and white spaces having a logical meaning in a specific natural language. Lemma should be marked with a key-word "basic form". *Examples (for English):* "basic form: car", "basic form: buy", "basic form: look for", "basic form: buy ticket button", "basic form: at", "basic form: must", "basic form: every".

Inflection. It is a string of letters and white spaces having a logical meaning in a specific natural language. Inflection should be marked with a key-word corresponding to InflectionType. *Examples (for English):* "singular: car", "plural: cars", "third person: buys", " infinitive: looking for".

InflectionType. It is a string of letters and white spaces finished with a colon ":", used in formulating a key-word used in describing Inflection. *Examples (for English):* "singular:", "plural:", "third person:", "infinitive:".

Conditional Conjunction. It is a string of letters and white spaces used in formulating conditional or state descriptive clauses. *Examples (for English):* "if", "when", "upon", "after", "during".

ModalVerb. It is a string of letters and white spaces used in formulating a modal form. *Examples (for English):* "will"/"would", "shall", "should", "may"/"might", "can"/"could", "must", "have to", "ought to".

Modifier. is a string of letters and white spaces used in describing (modifying the meaning of) a Noun. *Examples (for English):* "registered", "fast", "common".

Determiner. is a string of letters and white spaces used in formulating a quantity and a variability of a Noun. *Examples (for English):* "every", "one", "each".

Noun. It is a string of letters and white spaces used in formulating an objects' description. *Examples (for English):* "user", "system", "buy ticket button", "accessibility", "saving"

Preposition. It is a string of letters and white spaces used in formulating Phrases :: Phrases' context. *Examples (for English):* "on", "of", "to", "for", "inside", "next to", "in accordance with"

Verb. is a string of letters and white spaces used in formulating an action description. *Examples (for English):* "add", "show", "save", "start", "provide", "look for", "choose between"

TermHyperlink. As an abstract meta-class, TermHyperlink does not have a concrete syntax.

14.9 TermsRelations

14.9.1 Overview

This package describes subclasses, that form the basic structure for thesaurus and ontology. TermSpecialisationRelation as well as HasSynonym and HasHomonym relations are seen as the main semantic information for Terms :: Term.

The TermSpecialisationRelation structures the terms in a taxonomical hierarchy. This semantic definition is organisation-specific and is specified by extending the term structure (see Figures 14.37 and 14.38 for an example).

14.9.2 Abstract syntax and semantics

Figure 14.35 introduces two bi-directional relations that can be defined between two Terms, namely HasHomonym and HasSynonym.

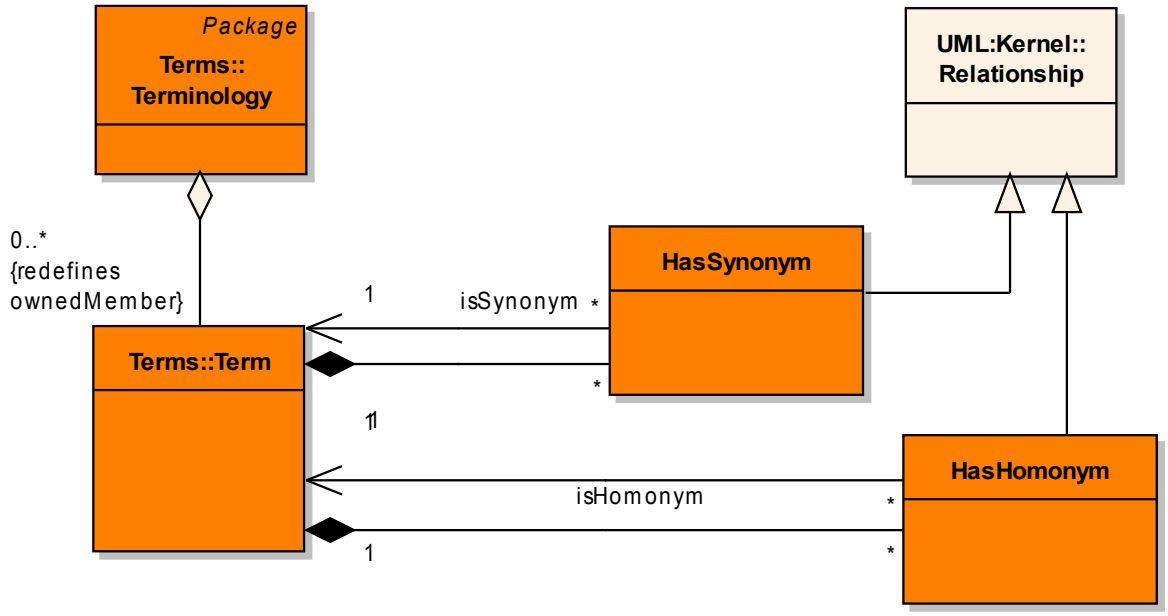


Figure 14.35: Synonym and homonym Terms's relations.

Figure 14.36 also contains all the specialisations of Term mentioned above. This figure focuses on the relationships that can be defined between two terms of the same type (only in between two Nouns, between two Verbs, etc.). All relations are specialisations of TermSpecialisationRelation and thus define a directed taxonomic relation.

HasHomonym

Semantics. A HasHomonym relates two Terms :: Terms that have the same character string as "name" attribute but that have different meanings.
Abstract syntax. HasHomonym is a kind of Relationship from UML :: Kernel package [Obj05b]. HasHomonym inherits concrete syntax from Relationship.

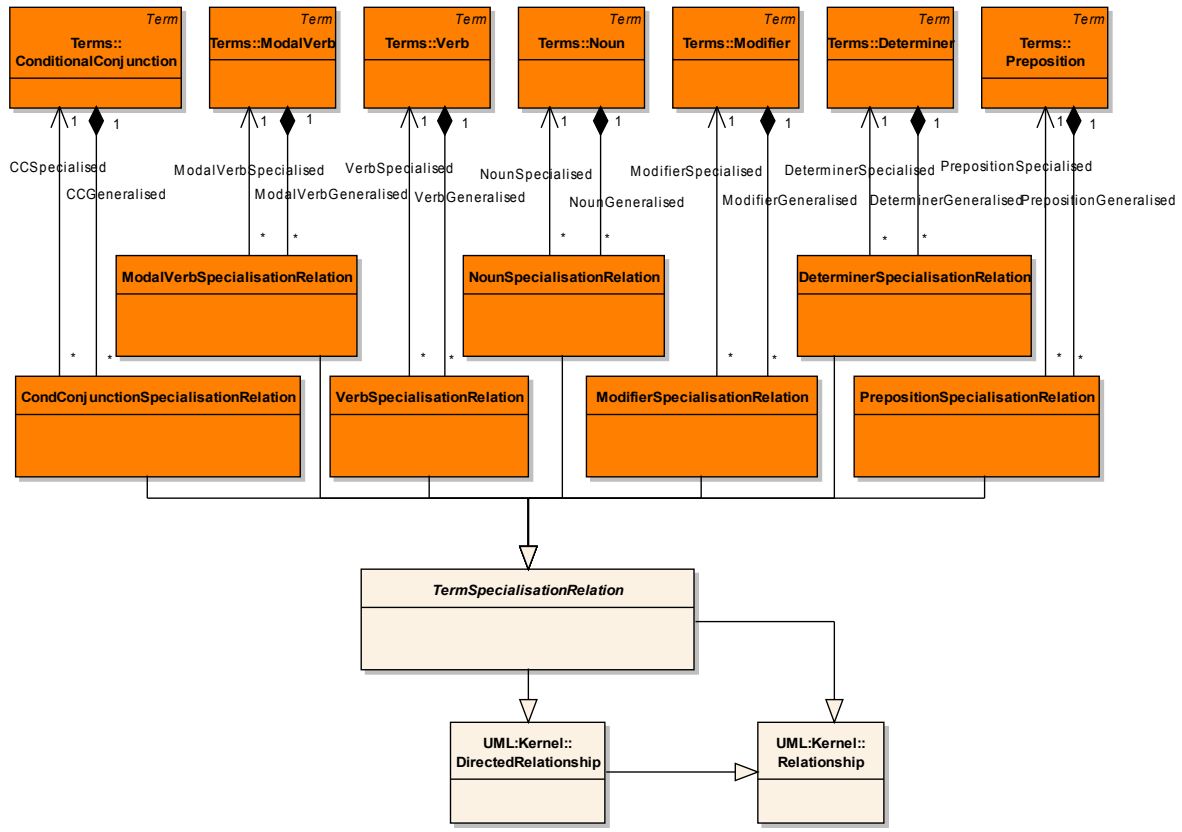


Figure 14.36: Specialisation Relations of terms from the same type.

HasSynonym

Semantics. A HasSynonym relates two Terms :: Terms that have the same meaning but a different character string as "name" attribute.
Abstract syntax. HasSynonym is a kind of Relationship from UML :: Kernel package [Obj05b]. HasSynonym inherits concrete syntax from Relationship.

TermSpecialisationRelation

Semantics. TermSpecialisationRelation is an abstract class that defines semantics for all other relation classes in this package other than HasHomonym and HasSynonym. TermSpecialisationRelation inherits concrete syntax from DirectRelationship. The subclasses of TermSpecialisationRelation are: CondConjunctionSpecialisationRelation, ModalVerbSpecialisationRelation, NounSpecialisationRelation, ModifierSpecialisationRelation, DeterminerSpecialisationRelation, PrepositionSpecialisationRelation. These relations define specific specialisation relations for the diverse types of terms. Those subclasses ensure that the source and the target of the relationship are of the same kind of Terms :: Term. The source of each relationship should be different from its target - Terms :: Term cannot be associated with itself.
Abstract syntax. TermSpecialisationRelation is a kind of DirectRelationship from the UML :: Kernel package [Obj05b].

CondConjunctionSpecialisationRelation

Semantics. A CondConjunctionSpecialisationRelation relates one Terms :: ConditionalConjunction (as the source of the relationship) to another Terms :: ConditionalConjunction (the target of the relationship).
Abstract syntax. CondConjunctionSpecialisationRelation is a kind of TermSpecialisationRelation.

ModalVerbSpecialisationRelation

Semantics. A ModalVerbSpecialisationRelation relates one Terms :: ModalVerb (as the source of the relationship) to another Terms :: ModalVerb (the target of the relationship).
Abstract syntax. ModalVerbSpecialisationRelation is a kind of TermSpecialisationRelation.

VerbSpecialisationRelation

Semantics. A VerbSpecialisationRelation relates one Terms :: Verb (as the source of the relationship) to another Terms :: Verb (the target of the relationship).
Abstract syntax. VerbSpecialisationRelation is a kind of TermSpecialisationRelation.

NounSpecialisationRelation

Semantics. A NounSpecialisationRelation relates one Terms :: Noun (as the source of the relationship) to another Terms :: Noun (the target of the relationship).
Abstract syntax. NounSpecialisationRelation is a kind of TermSpecialisationRelation.

ModifierSpecialisationRelation

Semantics. A ModifierSpecialisationRelation relates one Terms :: Modifier (as the source of the relationship) to another Terms :: Modifier (the target of the relationship).
Abstract syntax. ModifierSpecialisationRelation is a kind of TermSpecialisationRelation.

DeterminerSpecialisationRelation

Semantics. A DeterminerSpecialisationRelation relates one Terms :: Determiner (as the source of the relationship) to another Terms :: Determiner (the target of the relationship).
Abstract syntax. DeterminerSpecialisationRelation is a kind of TermSpecialisationRelation.

PrepositionSpecialisationRelation

Semantics. A PrepositionSpecialisationRelation relates one Terms :: Preposition (as the source of the relationship) to another Terms :: Preposition (the target of the relationship).
Abstract syntax. PrepositionSpecialisationRelation is a kind of TermSpecialisationRelation.

14.9.3 Concrete syntax and examples

HasSynonym. HasSynonym is presented in tree view as subelement (leaf) of Terms :: Term (see Figure 14.34).

HasHomonym. HasHomonym is presented in tree view as subelement (leaf) of Terms :: Term (see Figure 14.34).

TermSpecialisationRelation. As an abstract meta-class, TermSpecialisationRelation does not have a concrete syntax. All specialisation classes of TermSpecialisationRelation are depicted as a package with a subpackages in the tree view of the Terminology (see Figure 14.34).

Figures 14.37 and 14.38 show the abstract syntax of a terminology with domain-specific extensions (for ProDV related terms in this case). Figure 14.37 depicts the noun related part of the terminology, while Figure 14.38 shows the verb related part respectively.

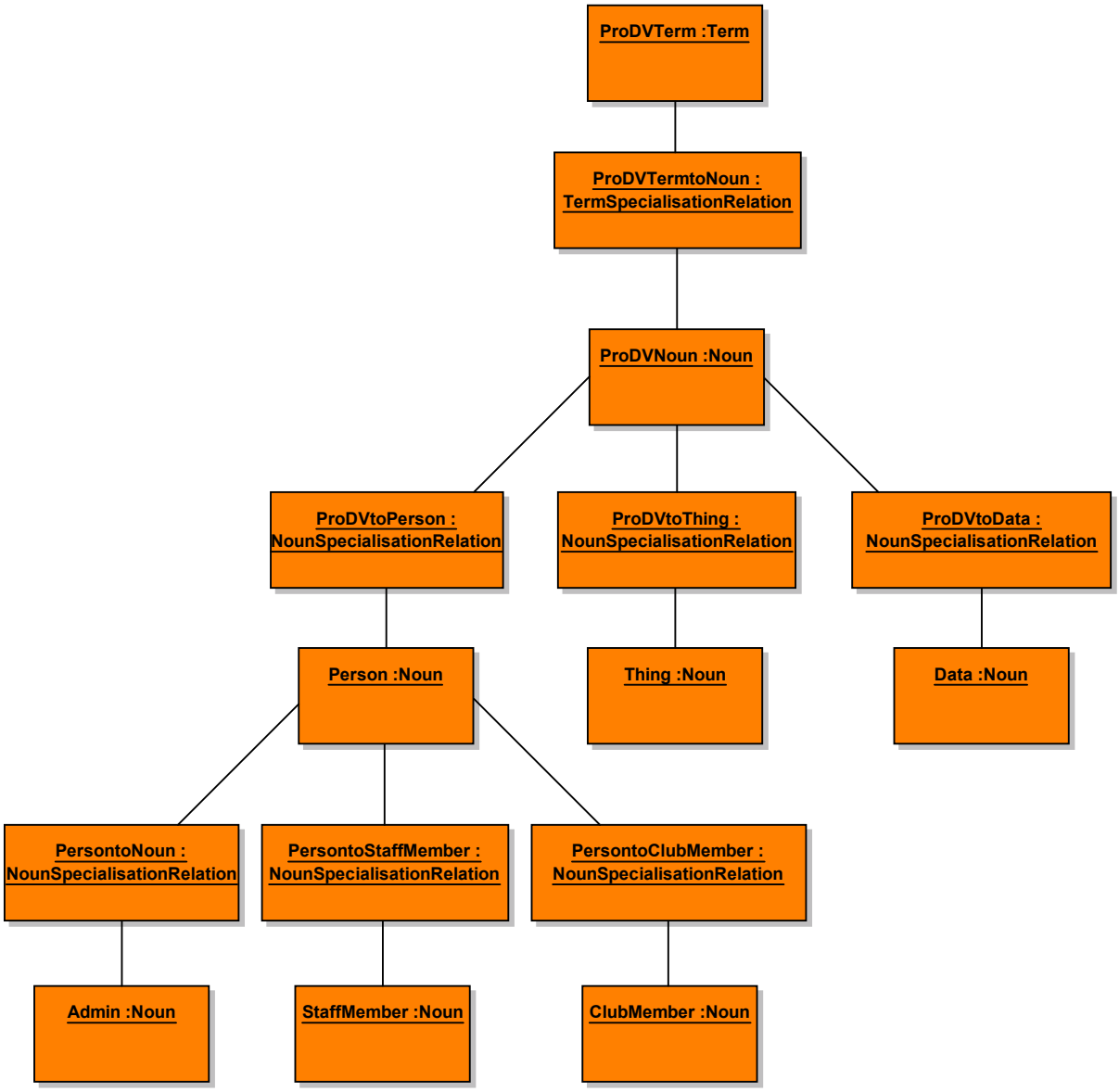


Figure 14.37: Part of a Terminology in abstract syntax with organisation-specific extension (only Nouns are presented in this figure.)

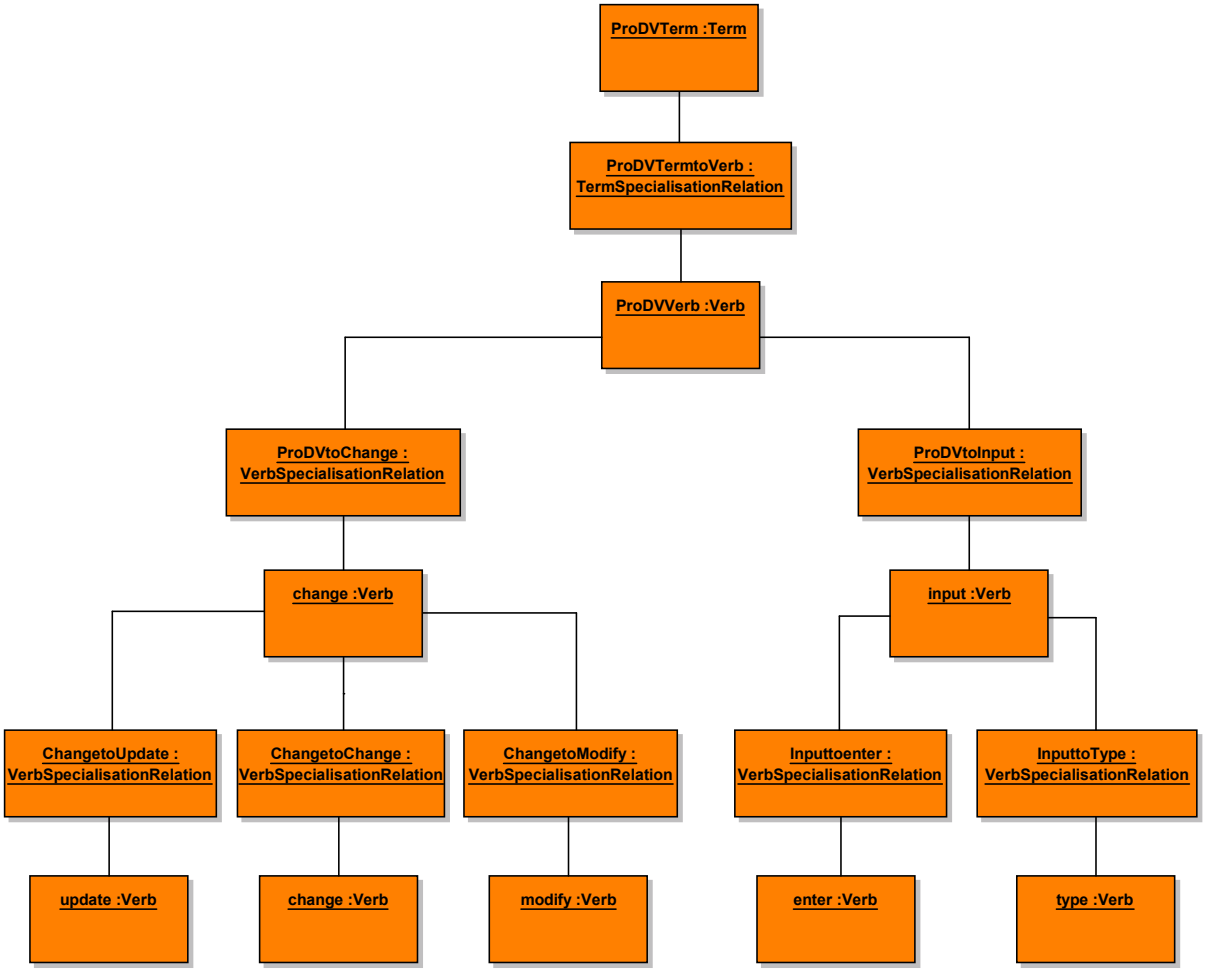


Figure 14.38: Part of a Terminology in abstract syntax with organisation-specific extension (only Verbs are presented in this figure).

Chapter 15

User interface elements

15.1 Overview

This chapter provides detailed description of the RSL elements for the representation of the user interface. Figure 15.1 shows dependencies of packages containing these elements. The main element for the UI structure is UIElement. The abstract syntax, the semantics and concrete syntax of these elements are given in the following sections. Examples of their concrete representation are also given.

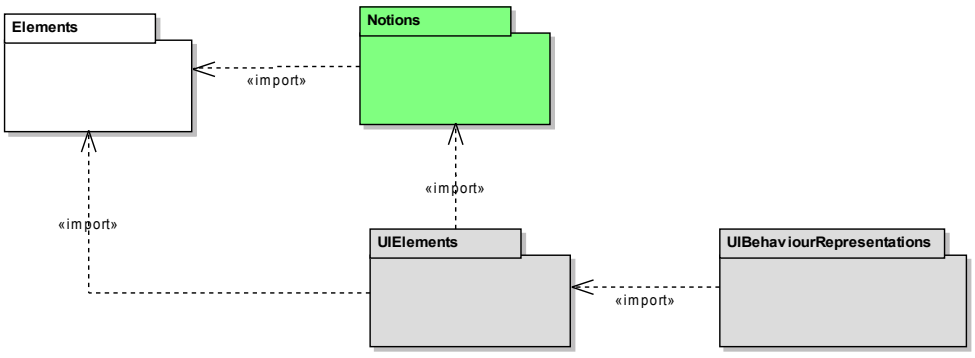


Figure 15.1: Overview of packages containing elements for the representation of the user interface

15.2 Abstract syntax and semantics

Figure 15.2 shows model-based elements for defining the user interface with the RSL. All user interface elements are subclasses of UIElement. These UIElements are general, meaning that they are not bound to any implementation toolkit.

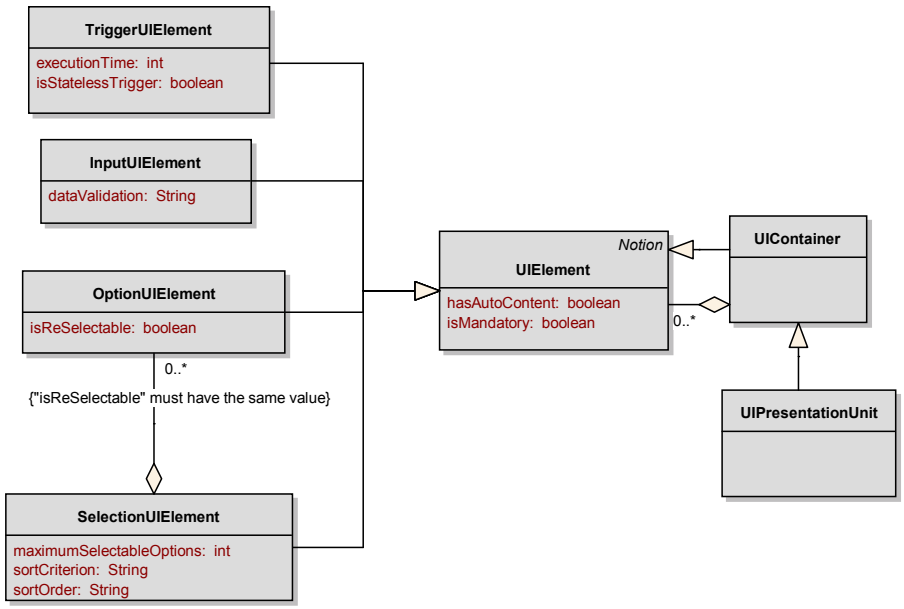


Figure 15.2: UIElements

UIElement

Semantics. UIElement is the general element for representable user interface elements. User interface elements are all elements that facilitate the interaction or communication between a user and the CBS to be developed. Concrete user interface elements may be visible, hearable, touchable etc. They can be implemented as software, e.g., widgets like *button*, *checkbox*, *table*, *window*, *form* etc., or hardware, e.g, gadgets or their components like *touch screen*, *keyboard*, *mouse*, *microphone* etc.

Abstract syntax. UIElement is a specialization of Notions :: Notion. It extends this with the Attributes

- *isMandatory* whose value indicates that a UIElement is mandatory (true) or optional (false) and

- *hasAutoContent* for determining whether the value of an Element can be generated automatically (true) or not (false).

that are common to all user interface elements. UIElement must have at least one representation. This can be defined by the class UIElementRepresentation. It is also associated with element PresentationOrder that defines the ordering of UIElements. UIElement can also associated with many InputOutputDevices for specifying hardware devices required for proper interaction with UIElement objects. The twofold association with UIBehaviourRepresentations :: UserAction reflects the role of UIElement in a user action; UIElement can be a sources of many UIBehaviourRepresentations :: UserActions and a UIBehaviourRepresentations :: UserAction can be a target of many UIElements. Regarding interaction, a user interface element can trigger a use case. For this purpose UIElement is associate with at least one RequirementsSpecifications :: UseCase. To allow for usage oriented definition of the user interface, UIElement is specialized into InputUIElement, TriggerUIElement, OptionUIElement, SelectionUIElement and UIContainer. The last one aggregates UIElements. Figure 15.3 shows UIElement and all its direct and extended relationships.

TriggerUIElement

Semantics. A TriggerUIElement is a user interface element with which users can trigger functions, operations etc. Example of concrete elements include buttons and toggle buttons most used in graphical user interfaces.
Abstract syntax. A TriggerUIElement is a special UIElement for direct triggering of operations. It extends UIElement by defining an integer attribute *executionTime* that can be used to specify the time that should elapse before the triggered operation starts and a boolean attribute *isStatelessTrigger* that can be used to specify whether the trigger element should maintain its state after triggering or not (see Figure 15.2).

InputUIElement

Semantics. A InputUIElement is a user interface element for data input. Text field and text area are examples of such elements on graphical user interfaces.
Abstract syntax. A InputUIElement is a special UIElement for entering new data into the system. It extends UIElement by defining an String attribute *dataValidation* that holds data validation constraints or other conditions as informal text (see Figure 15.2).

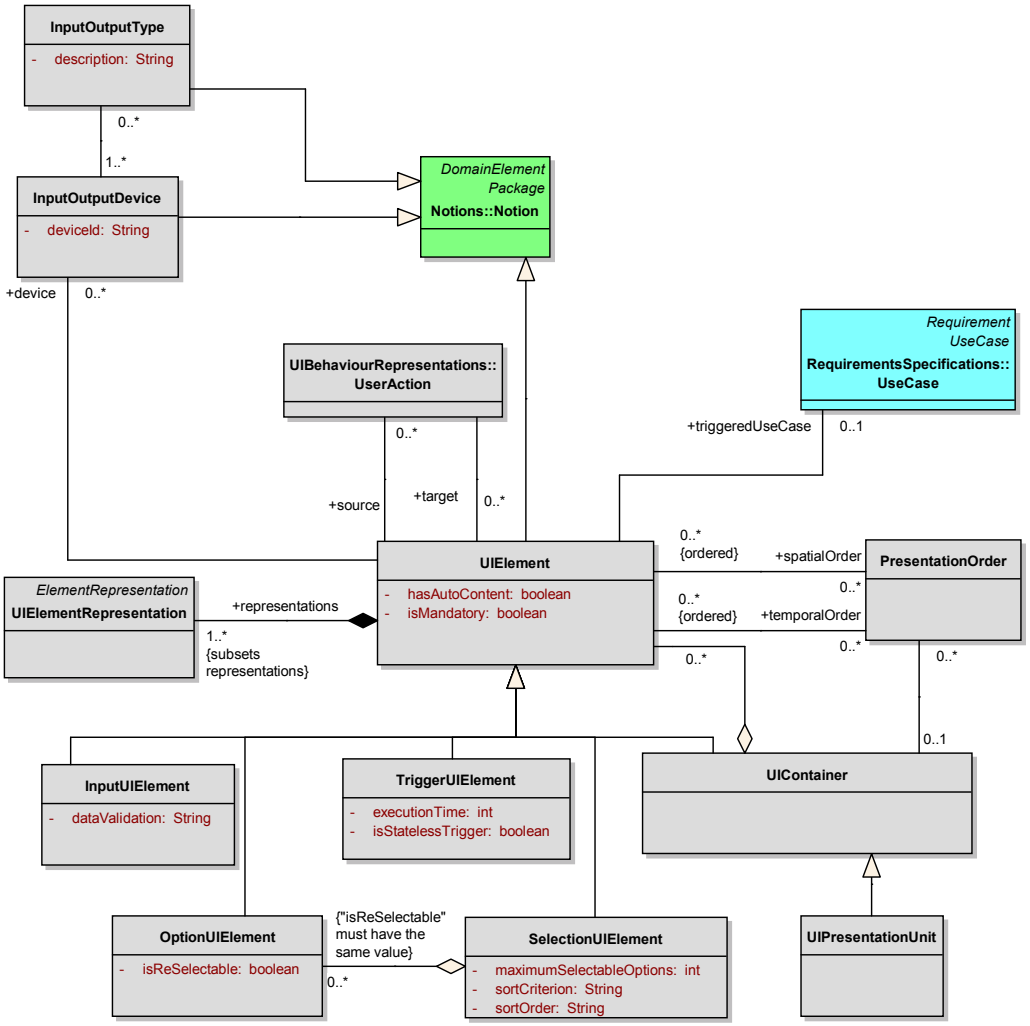


Figure 15.3: Relationships with UIElement

OptionUIElement

Semantics. A OptionUIElement is a user interface element for defining a single selectable option. This can be for example a checkbox or a radio button used for graphical user interfaces. The main difference between the two examples is that while a check box can be directly selected and deselected, a radio button can only be directly selected.

Abstract syntax. OptionUIElement is a specialization of UIElement. It extends UIElement by defining a boolean attribute *isReSelectable* for determining whether the option can be directly deselected after selection and vice versa (see Figure 15.2).

SelectionUIElement

Semantics. A SelectionUIElement is use interface element that presents a list of options to the user for selection. Depending on the settings, the user can select one or multiple options. Concrete examples include drop down lists or list.
Abstract syntax. SelectionUIElement is a specialization of UIElement. It extends UIElement by defining an Aggregation of options out of which some or all can be selected at once. The maximum number of options allowed can be specified in an integer attribute *maximumSelectableOptions*. All OptionUIElements in the aggregation must have the same value for the attribute *isReSelectable*. Moreover SelectionUIElement defines a String Attribute sortCriterion for specifying the criterion for sorting the options and sortOrder for specifying the order for sorting these options (see Figure 15.2).

UIContainer

Semantics. A UIContainer groups other UIElements. Such a group can be used as a structuring element or it can facilitate the manipulation and presentation of its elements. When a container is deleted from the user interface, all its contents are also deleted. A concrete examples for a UIContainer is a JPanel, from the Java Swing component.
Abstract syntax. A UIContainer is a special UIElement that can be used to define a group of other UIElements. This is indicated by the aggregation. The elements in a container may be organized in a certain order by using the element **PresentationOrder** (see Figure 15.2).

UIPresentationUnit

Semantics. A UIPresentationUnit groups elements that logically belong together and should be presented as a unit to the user. It is equivalent to a logical window that presents a view of information to a user [Lau05]. Therefore elements in a UIPresentationUnit should always be kept intact. Of course a PresentationUnit can contain elements organized in several UIContainers as subgroups. Concrete examples include dialogue windows, application windows or tabs windows.
Abstract syntax. A UIPresentationUnit is a special UIContainer that can be component of UIScene.

PresentationOrder

Semantics. User interface elements might need to be arranged in a certain order either in a container or as a flow of containers. This class defines occurrence order of the UIElement in parent one. Through PresentationOrder, UIElements can be ordered according to time (time sequence) or space (layout). PresentationOrder can not be used with root UIElement (element which is not included by any other).

Abstract syntax. PresentationOrder is associated with UIContainer, which includes same UIElements. It also has two associations to UIElement for defining the time sequence (temporalOrder) and layout (spatialOrder). See Figure 15.4.

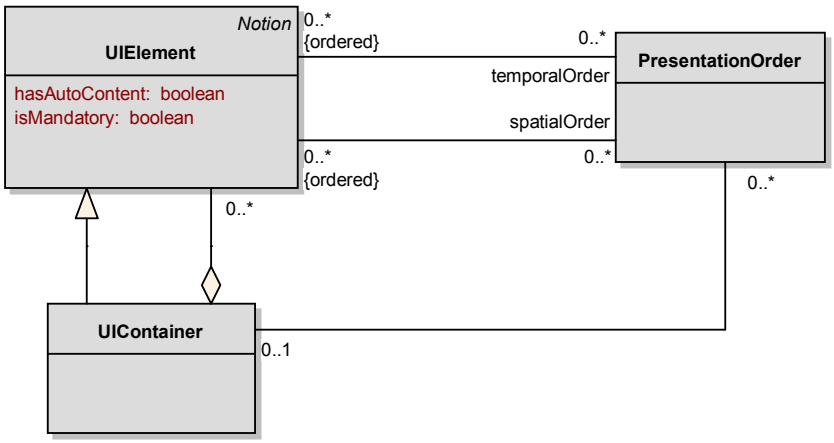


Figure 15.4: Ordering Element for UI Elements

UIElementRepresentation

Semantics. UIElementRepresentation is a possible representation of the UIElement (another is DomainElementRepresentations :: DomainElementRepresentation, while UIElement is kind of Notions :: Notion). It may be expressed in most appropriate manner for concrete UIElement. Basically it is exhibiting an UIElement in some visible image (screenshot) or another form (icon, file, voice).

Abstract syntax. UIElementRepresentation is a kind of Elements :: ElementRepresentation. It is a part of UIElement and forms its "representations". See Figure 15.5.

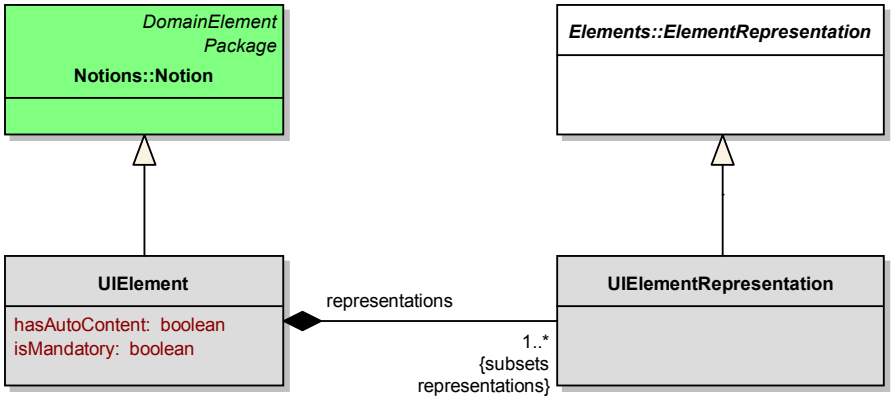


Figure 15.5: Representatinos of UI elements

InputOutputDevice

Semantics. InputOutputDevice gives a possibility to express a external device connected with the UIElement. Such device facilitates the interaction of the user with UIElements. For example user interface elements with drop mechanism like "drop down list", require pointing interaction devices like the "mouse" for proper interaction [ZK99].
Abstract syntax. InputOutputDevice is a specialisation of Notions :: Notion. Due to this specialisation, InputOutputDevice has representation of Notions :: Notion and is stored in Notions :: NotionsPackage. InputOutputDevice can be associated with many UIElements by “device” attribute and InputOutputTypes. It should have a unique identifier “device_id”that can be stored as a String. See Figure 15.6.

InputOutputType

Semantics. InputOutputType represents the style of interaction between user and element of user interface, which is supported by InputOutputDevice. A device may have some specific interaction styles. For example a mouse can best support the interaction types “selection”, “drag and drop”, “scrolling”, etc.
Abstract syntax. InputOutputType is a specialisation of DomainElement. Due to this specialisation, InputOutputType has DomainElement’s representation and is stored in DomainVocabulary. InputOutputType is a class of InputOutputDevice types. Attribute *description* gives details of InputOutputDevice. An InputOutputType should be supported by at least one InputOutputDevice (see Figure 15.6).

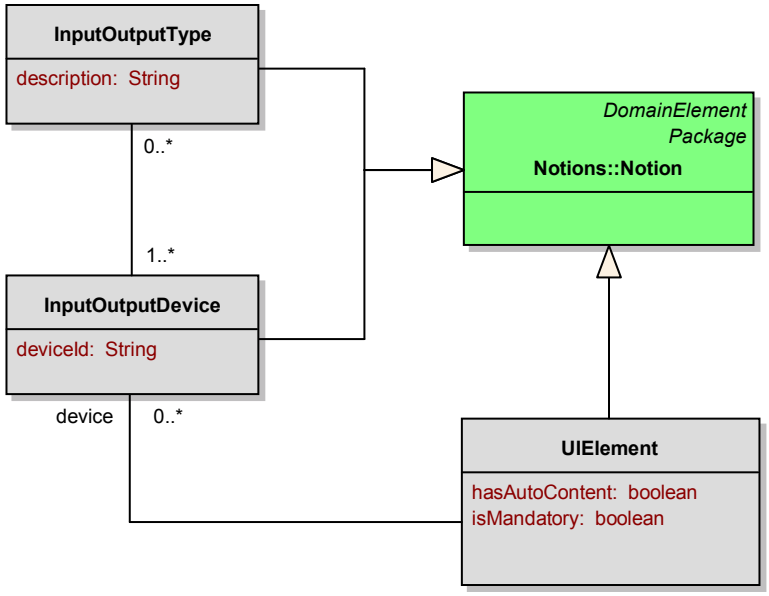


Figure 15.6: Devices for UI elements

15.3 Concrete syntax and examples

The concrete syntax of UIElements should consist of prototype oriented domain specific presentations that can be easily understood by users. This facilitates their communication the requirement analyst and allows early evaluation. This can be done by defining domain specific stereotypes and associating them with UIElements.

UIElement. The concrete syntax of UIElement depends on the context in which it is presented to the user. It can be represented like other Notions :: Notions at DomainElement's diagram, in tree view with nodes categorised in related UserActions, InputOutputDevices, UIElementRepresentations and PresentationOrders, as a wiki-like description (see Section 14.3) or as a stand alone icon. Figure 15.7 shows two examples of the concrete syntax of UIElement. The representation is an icon consists of the element name and the symbol indicating the element type. Moreover the symbol can indicate whether the UIElement is mandatory, critical etc. The tree view representation shows all elements associated with the UIElement in categeries described above. Additionally, each category has an icon for easy identification.

Other UIElements. Since TriggerUIElement, InputUIElement, OptionUIElement, SelectionUIElement, UIContainer and PresentationUnit are special UIElements, they can also be represented by using the concrete syntax shown above. However an other icons indicated in Figure 15.8 are used in order to visually differentiate them from each other.

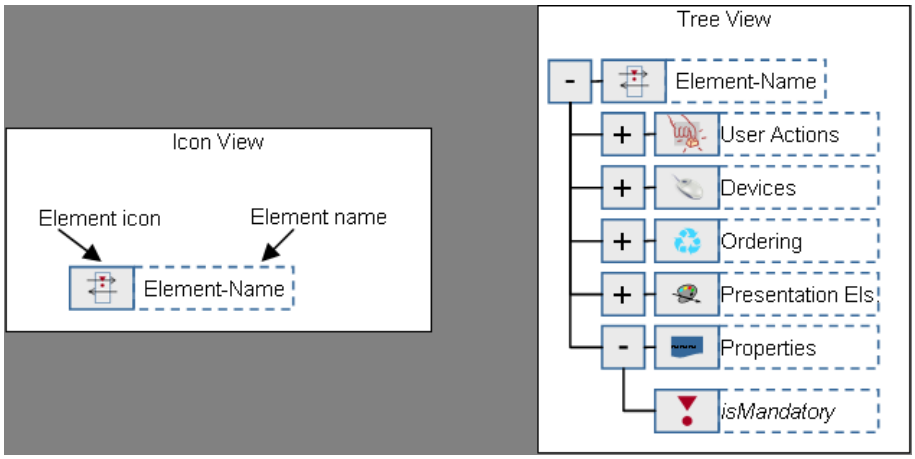


Figure 15.7: Examples of UIElement Concrete Syntax

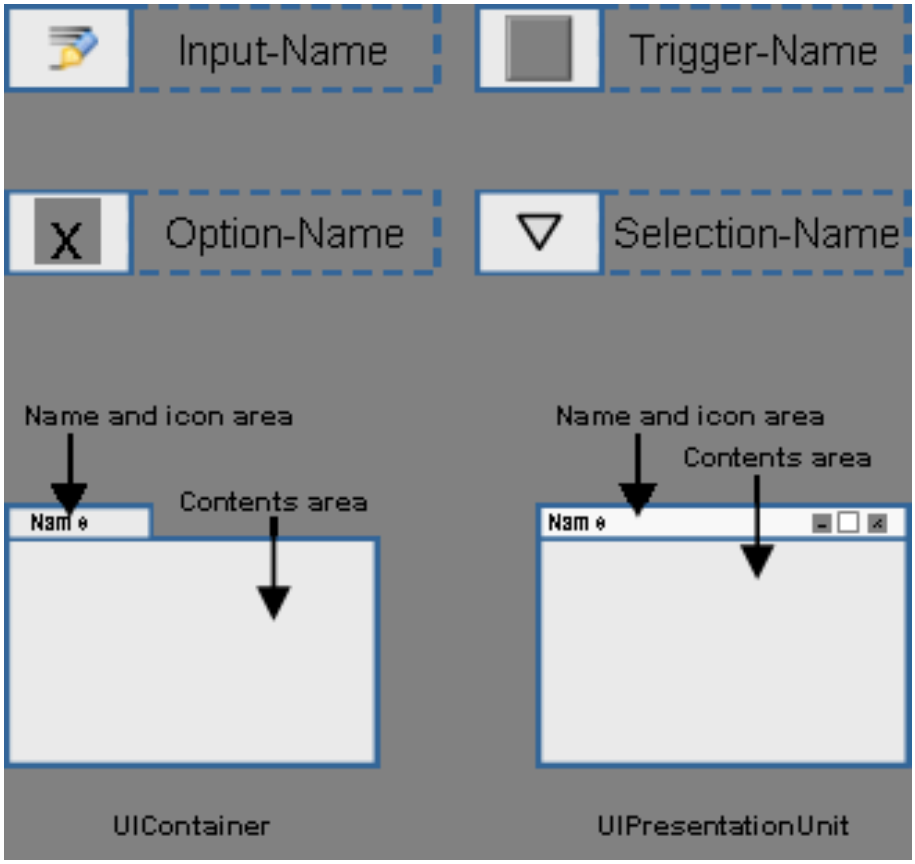


Figure 15.8: Examples of Concrete Syntax for other UIElements

PresentationOrder. This class is presented in the form of table. It has three columns:

- child UIElement's name,
- spatial order,
- temporal order.

The first row of this table includes names of columns and further rows includes concrete data. Example of concrete syntax for this class can be found at the Figure 15.9.

UIElement name	**Spatial order**	**Temporal order**
LoginTextField	1,2	0
PasswordTextField	2,2	0
AcceptButton	3,3	0

Figure 15.9: UIPresentationOrder concrete syntax example

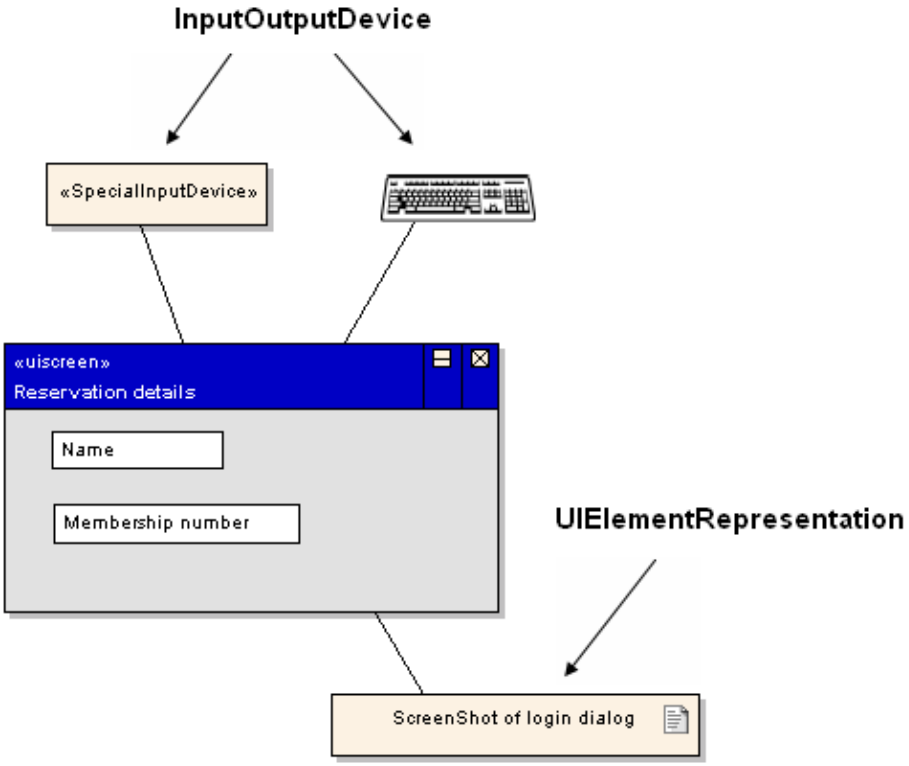


Figure 15.10: UIElementRepresentation concrete syntax example

InputOutputDevice. Its concrete syntax depends on the context in which InputOutputDevice is presented to the user. It can be represented like Notions :: Notion at DomainElement's diagram,

in tree view or as a wiki-like description (see Section 14.3). In user interface preview diagram it is presented as rectangle with a stereotype name describing device (see Figure 15.10). For predefined stereotypes, InputOutputDevice is presented as appropriate icon (see A.2).

InputOutputType. Its concrete syntax depends on the context in which InputOutputType is presented to the user. It can be represented like Notions :: Notion at DomainElement's diagram, in tree view or as a wiki-like description (see Section 14.3).

UIElementRepresentation. UIElementRepresentation is presented as rectangle with the upper right corner "note symbol" and name of the representation in the center. It is an external item, which is used for additional description UIElement. Composition relation between UIElementRepresentation and UIElement is expressed by simple line. For example see Figure 15.10.

Chapter 16

User interface behaviour representation

16.1 Overview

This package describes the general structure of UI elements that are used to describe behaviour of user interface. UIBehaviourRepresentations :: UIStoryboard as the main element presenting behaviour of user interface includes UIScenes connected to ScenarioSenteces :: SVOScenarioSentence. Each UIScene is built from UIElements :: UIPresentationUnit enriched by description and scene number. UIScenes are related through UserActions which represent the actions of users in interaction with the system.
UIPresentationUnit is a snapshot of the user interface of the system (in form of graphic, text, etc.) at each specific step of interaction between the user and the system. This is a part of the concrete syntax of UIScene. The other part is the textual description and the number of the scene. A connection between UIScenes is presented as an arrowed line. All these elements together create the concrete syntax of UIBehaviourRepresentations :: UIStoryboard.

16.2 Abstract syntax and semantics

Abstract syntax for the UIBehaviourRepresentations package is described in Figure 16.1.

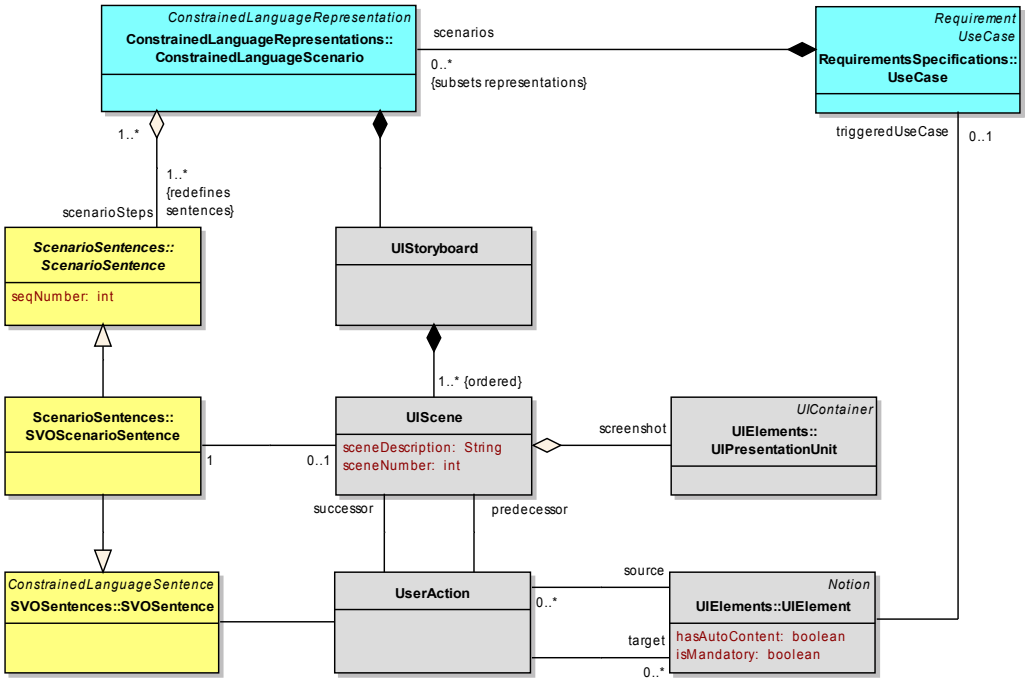


Figure 16.1: UIBehaviour representation

UIStoryboard

Semantics. UIStoryboard is a series of scenes displayed in sequence for the purpose of previsualizing user interface (UI) appearance from the perspective of behaviour.
Abstract syntax. UIStoryboard is a component of ConstrainedLanguageRepresentations :: ConstrainedLanguageScenario. UIStoryboard contains ordered set of one or more UIScenes.

UIScene

Semantics. UIScene represents the state of the user interface of the system corresponding to ScenarioSenteces :: SVOScenarioSentence. Transitions between UIScenes are adequate to a user's actions in the use case scenario. UIScenes structured in an ordered list are part of the UIBehaviourRepresentations :: UIStoryboard.
Abstract syntax. UIScene includes one UIElements :: UIPresentationUnit and can be pointed to by no more than one ScenarioSenteces :: SVOScenarioSentence. It is a component of UIStoryboard. Each UIScene has its own number sceneNumber of type integer and can have a description sceneDescription of type string. UIScenes related through UserAction form an ordered list.

UserAction

Semantics. UserAction represents user activity results in transition from one UIScene to another (predecessor and successor). Interaction of UserAction is triggered by connected ScenarioSenteces :: SVOScenarioSentence. This action is performed on one source UIElement in predecessor UIScene and can influence some target UIElements in successor UIScene.
Abstract syntax. UserAction is associated with SVOSentence :: SVOSentence and two UIScenes in roles of predecessor and successor. It has also associations with source and target UIElements.

16.3 Concrete syntax and examples

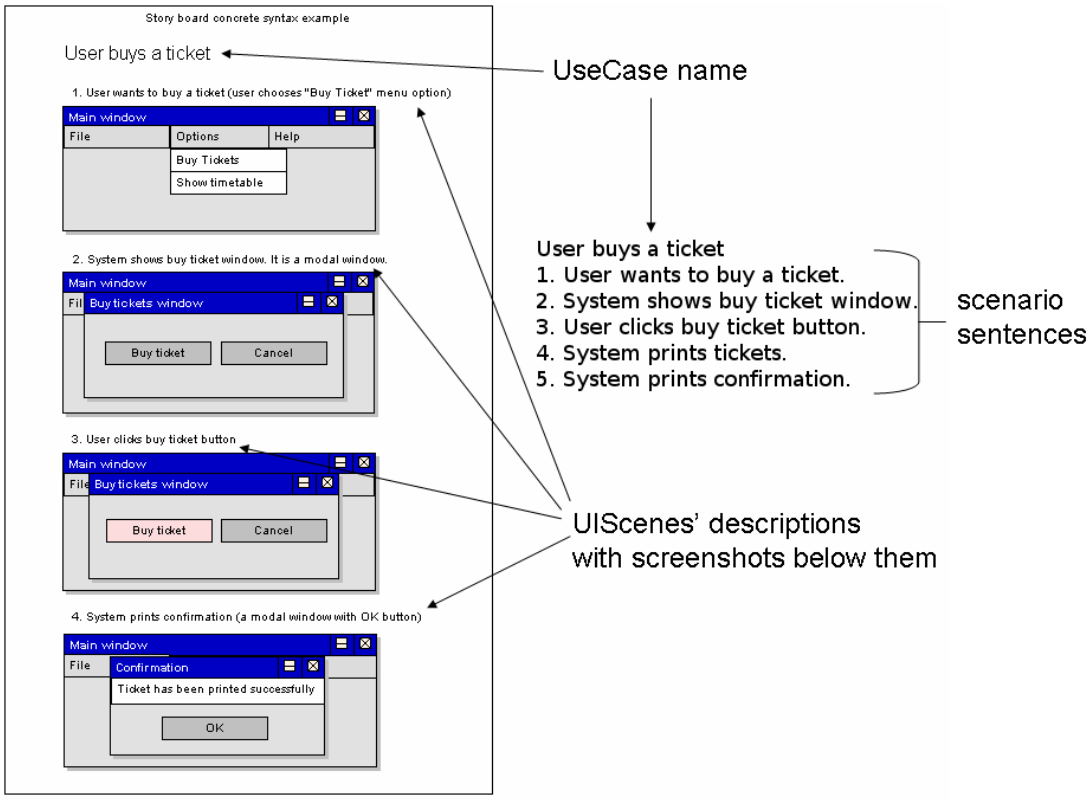


Figure 16.2: UIStoryboard concrete syntax example

UIStoryboard. A Storyboard is a series of screenshots with additional textual information about them (UIScenes). A UIStoryboard representation must contain the name of a RequirementsSpecifications :: UseCase, which scenario it is illustrating. The description of contained UIScenes should consist of a scene number and short textual descriptions of scenes (they may include scenario sentences expressions with additional information specific to the UI concepts).

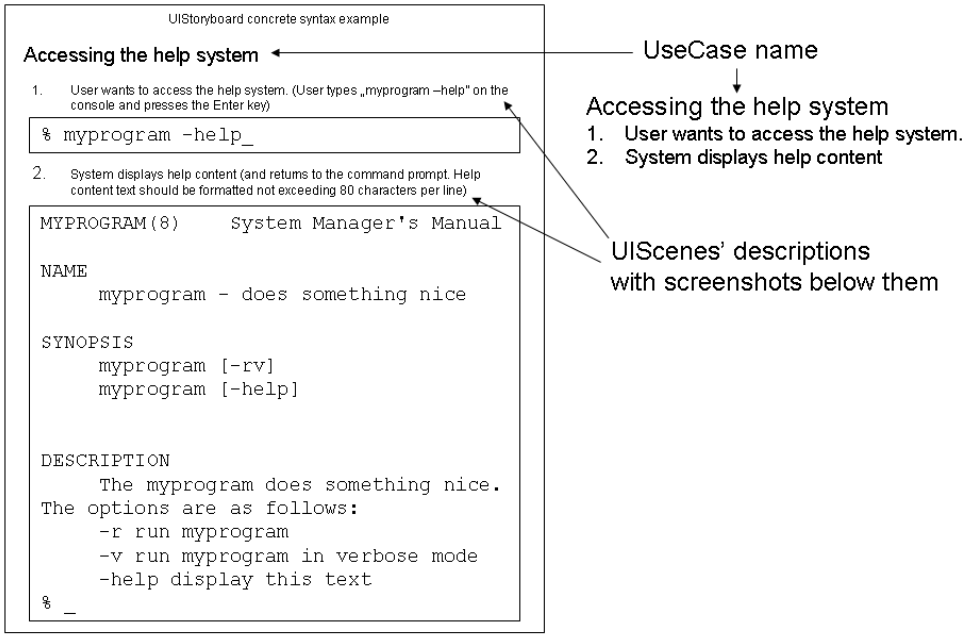


Figure 16.3: UIStoryboard concrete syntax example (2)

Figure 16.2 depicts how a scenario containing 5 sentences is illustrated with a storyboard. All but one of the sentences have screenshots in the storyboard (for sentence number 4 no scene is needed as it does not describe any user – UI interaction). Note that numbers of scenes does not correspond with UseCase sentences numbers. The descriptions of UIScenes include partial or full text of sentences scenarios, completed with additional UI-specific information (e.g. last scene's description depicts a confirmation window and the first scene's description specifies how the user can initiate the interaction).

Storyboards do not necessarily relate to Graphical User Interfaces (GUIs). Figure 16.3 shows an example of UIStoryboard presenting textual user interface.

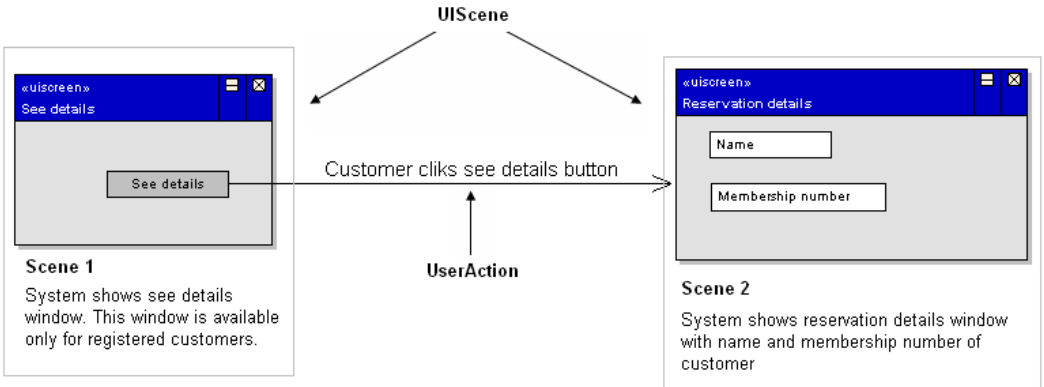


Figure 16.4: UIScene concrete syntax example

UIScene. The concrete syntax of this class consists of two main parts. One is a figure presenting the state of the system - UI elements. The other part is a textual description of this figure with the number of the UIScene's in a UIStoryboard. Ilustration of concrete syntax of a UIScene can be found in Figure 16.4 which presents a consolidated example for UIBehaviour elements.

UserAction. The concrete syntax of this class is an arrowed line between two UIScene's in the UIStoryboard with ScenarioSenteces :: SVOScenarioSentence text above (see Figure 16.4).

Chapter 17

Conclusion

This deliverable presents a new requirements specification language named RSL. It integrates a behavioural and a structural part, and even a part for user-interface specifications.

The behavioural part of this language is special because of its clear distinction between Functional and Behavioural Requirements as well as its precise definition of their relationships. Its conceptual definition is new in its clear distinction between requirements and *representations* of requirements. This distinction is important for the use of this language as a basis for reuse based on requirements, since only representations can actually be reused. This language is also unique through its explicit distinction between *descriptive* and *model-based* requirements representations.

The structural part of this language is special because of its explicit inclusion of objects existing in the domain (environment) of the software system to be built — *domain objects*. A conceptual Domain Model (to-be) can be represented in newly defined domain entity diagrams. Additionally, descriptions are possible in a newly defined representation of vocabulary, which is organised as a terminology representation that integrates a dictionary with a thesaurus. Both kinds of domain representations facilitate a better understanding of the requirements per se.

The part of this language for user-interface specifications is special because of its explicit binding between behavioural representations of requirements (e.g., scenarios in Use Cases) with user-interface elements. It can be used to describe user interfaces in a platform-independent way. The user interfaces can be textual or graphical, and with various types of access (limited, voice-triggered etc.). Features specific to culture and region are also supported.

The user-interface specifications may contain textual descriptions as well as screenshots or drawings illustrating the appearance of user-interface elements. With the envisioned tool support, the graphical representation will allow visualising interaction of a user with a software system even before any prototype is available.

A major contribution of our work is the coherent integration of all these parts of RSL. In particular, textual descriptions in user-interface specifications can be written according to the same grammars as the textual descriptions in structural and behavioural specifications.

Our language is the first requirements specification language intimately integrated with UML and defined using the same metamodelling approach as used for UML itself (using MOF).

This deliverable also presents and explains the complete language definition, from abstract down to concrete syntax.

Appendix A

Profiles for User Interface Representation

This annex describes the predefined standard stereotypes for User Interface Representation. The standard stereotypes are specified in two separate profiles. These profiles can be applied to a user model elements just like any other profile.

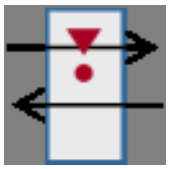
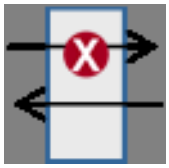
The stereotypes belonging to the profile are described using a compact tabular form rather than graphically. The first column gives the name of the stereotype label corresponding to the stereotype. The actual name of the stereotype is the same as the stereotype label except that the first letter of each is capitalised. The second column identifies the metaclass to which the stereotype applies and the third column provides a description of the meaning of the stereotype. The last column contains a figure that symbolises redefined concrete syntax of class that stereotype applies to.

A.1 Profile for user interface elements

This profile provides concrete syntax for various types of graphical user interface elements in information systems. The images are suggestions and they can be changed accordingly. The first two lines are general symbols that can be used to explicitly mark mandatory and critical

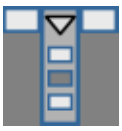
visual user interface elements.

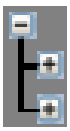
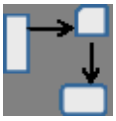
Name	Applies to	Description	Concrete syntax
Mandatory symbol	UIElements :: UIElement	A symbol used to indicate that a ui element is mandatory on the user interface. Valid input must be provided before moving to another ui element.	The mandatoryness of an element can be represented by a red exclamation mark: 
Critical symbol	UIElements :: UIElement	A symbol used to indicate that the interaction with such a ui element has critical effects to the system. For example shutting down the system, deleting data, etc. Therefore, the user should be careful before interacting with such an element.	A red circle with a white **X** indicated that a ui element is critical: 
User interface element	UIElements ::UIElement	A user interface element is a mediator between the user and the system. It takes user inputs into the system and makes system outputs available to the user.	A generic ui element is represented by a quad that is penetrated with two lines in both directions to indicate the interaction nature of ui elements: 

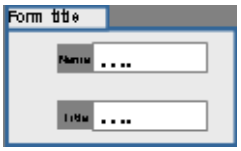
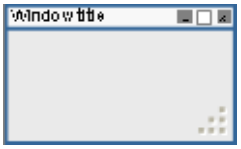
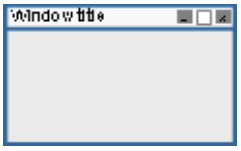
Name	Applies to	Description	Concrete syntax
Mandatory user interface element	UIElements :: UIElement	see above.	Adding a mandatory symbol to a ui symbol indicates that it is mandatory: 
Critical user interface element	UIElements :: UIElement	see above.	Adding a critical symbol to a ui symbol indicates that it is critical: 
Button	UIElements :: Trigger	A button is used to trigger an operation. It immediately returns in its normal state thereafter.	A squared labelled "OK" represents a button: 
Toggle Button	UIElements :: Trigger	A toggled button maintains its triggered state after triggering. Its is therefore possible to visually know if it button has been pressed or not.	A squared with two parts with different labels and colours symbolises the states of a toggled button. The transition between the states is indicated by two opposite but equal arrows: 

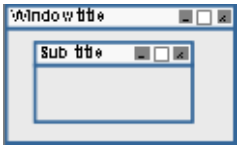
Name	Applies to	Description	Concrete syntax
Radio Button	UIElements :: Option	A radio button allows only selecting an option. Once an option has been selected, it can not be deselected directly.	A radio button is symbolized by a circle with a black point at the centre. The label "Opt" emphasises that a radio button should be labelled: 
Check Box	UIElements :: Option	A check box allows unlimited and direct selecting and deselecting of an option. This means that the actions of selecting and deselecting an option can be repeated unlimitedly.	A check box is indicated by a symbol with two boxes. One box contains an x indication the selected state and another box is empty indicating the unselected state. The arrows between the boxes symbolise the sate change. The label "Opt" emphasises that a check box should be labelled: 
Combo Box	UIElements :: Selection	A combo box is used to present a list of options to the user. The user can only select one option at a time. Only the selected option is visible after selection. Other options are hidden in the drop down list. That's why a combo box can also be referred to as drop down list. Should the user want to change her/his selection, she/he has to open the drop down list.	An icon with a label "Opt" followed with an arrow pointing down symbolises a combo box. An additional to that, a labelled dotted box is attached to this icon to indicate a hidden drop down list of options: 

Name	Applies to	Description	Concrete syntax
List	UIElements :: Selection	A list displays an open list of more that one option that it contains. Depending on its size, all options can be always visible to the user. Depending on the configuration, more than one option can be selected from a list at a time.	A list is represented by a boy containing bullets as indicators for list options: 
Radio List	UIElements :: Selection	A radio list is a group of radio buttons. Only one option can be selected at a time. The selection of a different element automatically deselects a previously selected option.	An icon of a box containing radio buttons symbolises a radio list. One option is selected to indicate that only one option can be selected at a time: 
Check List	UIElements :: Selection	A radio list is a group of check boxes. Depending on the configuration, more than one option can be selected from a check list at a time.	An icon of a box containing check boxes symbolises a check list. Two options is selected to indicate that more than one option can be selected at a time: 

Name	Applies to	Description	Concrete syntax
Menu	UIElements :: Selection	A menu is used to structure system functions and make them visible to the user as a list of options. Menus are commonly presented as drop down list that are displayed after opening a menu (see combo box). Contrary to combo boxes, a drop down menu hides all its options (also the selected ones) after selection has been performed.	An icon with three horizontal buttons and a drop down list symbolizes a menu. The open list indicates a selected and hence rolled out menu: 
Text Field	UIElements :: Input	A text field is used to enter single line text input into the system. Default text can be provided, that can then be overwritten by the new input.	Since a text field is for entering text input, a box with a T, dotted points and a pencil is used for symbolizing this element: 
Text Area	UIElements :: Input	A text area is used to enter multiple line text input into the system. Default text can be provided, that can then be overwritten by the new input.	The same icon like for text field but with two dotted lines for symbolizing multiple lines is used: 
Label	UIElements :: UIElement	A label is used as a caption for other ui elements. The text contained in a label can not be edited.	An icon containing the text "Tt" and with a grey background colour symbolizes a label: 

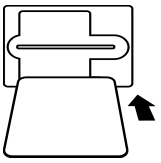
Name	Applies to	Description	Concrete syntax
Panel	UIElements :: UIContainer	A panel is used to group (logically related) user interface elements. The type and ordering of elements in a panel is irrelevant.	A box containing different elements symbolizes a panel: 
Tree	UIElements :: UIContainer	A tree is a container of hierarchically ordered user interface elements, which are its nodes. The topmost node is a root of a tree. All other nodes are siblings of the root node. The deepest node of each tree path (i.e., a way from the root to any other node) is a leaf. A leaf has not siblings. A tree node can be expanded to display its siblings or collapsed to hide them.	An icon of a tree of nodes with the root node open symbolizes this ui element: 
Layout	UIElements :: UIPresentationOrder	Layout is not a ui element. It is the result of ordering ui elements in a spatial order.	Layout is represented by three elements aligned on a line to symbolize their spatial order: 
Flow	UIElements :: UIPresentationOrder	Flow is not a ui element. It is the result of ordering ui elements in a temporal order.	Flow is represented by three elements connected with arrows to symbolize their time order: 

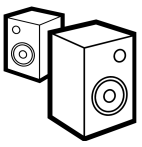
Name	Applies to	Description	Concrete syntax
Form	UIElements :: UIContainer	A form is used to collect and display. It does so by using controls like text fields, combo buttons etc (see above).	A form is represented by titled tab that contains elements to symbolize the controls in the form: 
None arbitrary Resizable Window	UIElements :: UIPresentationUnit	A window is a floatable, movable and closable container of ui elements. A none arbitrary resizable window has one standard size and can only be maximized to occupy the whole allocated space on the screen or minimized to hide it from the screen.	This window is represented by a titled window icon with buttons for minimize, maximize and close: 
Resizable Window	UIElements :: UIPresentationUnit	A resizable window has a standard size but it can be resized to any arbitrary size allowed. Moreover maximizing and minimizing it is possible.	This window is represented by an icon resembling that of none arbitrary resizable window. A dotted triangle is added at the bottom left corner to indicate the possibility for resizing (for example by pulling with a mouse): 

Name	Applies to	Description	Concrete syntax
Multi Document Interface Window	UIElements :: UIPresentationUnit	A multi document interface window can display other windows as its children. This is very applicable for Multi Document Interface (MDI) Applications. It is therefore possible to open many documents of the same type within the same application window. The opposite are Single Document Interface (SID) Applications that can have only one document opened at a time. The open document must be closed before opening another document.	A MDI-Window is represented by a window icon containing another window icon as a symbol for the parent window and child window: 
Dialog	UIElements :: UIPresentationUnit	A dialog is a small window for presenting messages to the user. It can not be resized. System dialogs are system modal, i.e., no user interaction can be performed with other windows in the system as long as a dialog is displayed. Most application dialogs are application modal, meaning that no user interaction can be performed within the application other than with a displayed dialog.	A Dialog is represented by a titled window containing buttons for "Ok", "Cancel" and a close symbol: 

A.2 Profile for devices

This profile allows changing concrete syntax for various types of input and output devices used in computer based systems. Note that images in table below are only suggestions of depicting devices for UI.

Name	Applies to	Description	Concrete syntax
card reader	UIElements :: InputOutputDevice	Used for devices communicating with various types of cards (magnetic, memory). This kind of device can be used as a system input (e.g. read data from a magnetic card) or output (e.g. store data on memory card).	An icon depicting a card reader: 
joystick	UIElements :: InputOutputDevice	A joystick is a system input device consisting of a handheld stick that pivots about one end and transmits its angle in two or three dimensions to a computer based systems. Joystick can also contain a number of buttons and switches.	An icon depicting a joystick: 
keyboard	UIElements :: InputOutputDevice	Keyboard is an input device that is an arrangement of buttons, which allow triggering commands in computer system or correspond to letters of alphabet and other written symbols.	An icon depicting standard computer keyboard: 
microphone	UIElements :: InputOutputDevice	A microphone is a voice input device.	An icon depicting a microphone: 

Name	Applies to	Description	Concrete syntax
mouse	UIElements :: InputOutputDevice	A mouse is a computer pointing device. It is designed to detect two-dimensional motion relative to its supporting surface. Mouse consists of a small case, to be held under one of the user's hands, buttons (typically one or two) and/or other elements (like scrolling wheel). The mouse's motion typically translates into the motion of a pointer on a display.	An icon depicting a mouse: 
printer	UIElements :: InputOutputDevice	A printer is an input-output device. Printer used as output device allows producing hard copies of data from computer system. Printer used as input allows transmitting commands to computer system (mainly concerning printing process).	An icon depicting a printer: 
screen	UIElements :: InputOutputDevice	A screen is a system output device. It consists of monitor (CRT, LCD) which allows display of graphics, text etc. for the user.	An icon depicting a computer monitor – CRT display or LCD screen: 
speaker	UIElements :: InputOutputDevice	A speaker is an output voice device of computer system.	An icon depicting a speaker or speaker set: 

Name	Applies to	Description	Concrete syntax
touchscreen	UIElements :: InputOutputDevice	A touchscreen (also touch screen, touch panel or touchscreen panel) is input output device. A touchscreen is display overlay which have the ability to display and receive information on the same screen.	An icon depicting a touch screen: 

Appendix B

List of abbreviations

ACE	Attempto Controlled English
AIO	Abstract Interaction Objects
ATM	Automatic Teller Machine
CASE	Computer Aided Software Engineering
CBS	Computer-Based System
CHI	Computer Human Interaction
CIO	Concrete Interaction Objects
DM	Domain Model
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
INTERACT	International Conference on Human Computer Interaction
IUI	International Conference on Intelligent User Interfaces
MB-UID	Model Based User Interface Development
MOF	Meta Object Facility
OMG	Object Management Group
OOA	Object Oriented Analyses
RE	Requirements Engineering
RSL	Requirements Specification Language
RUP	Rational Unified Process
SVO	Subject Verb Object
SysML	System Modelling Language
TORE	Task and Object Oriented Requirements Engineering

UbiComp	International Conference on Ubiquitous Computing
UC	Use Case
UI	User Interface
UIDL	User Interface Description Language
UML	Unified Modeling Language
XML	Extensible Markup Language
XUL	XML User Interface Language

Bibliography

[AH02] M Abrams and J Helms. *UIML v3.0 Draft Specification*, 2002.

[Ant96] Annie I. Antón. Goal-based requirements analysis. In *ICRE '96: Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96)*, page 136, Washington, DC, USA, 1996. IEEE Computer Society.

[BI96] Barry Boehm and Hoh In. Identifying quality-requirement conflicts. *IEEE Software*, 13(2):25–35, 1996.

[BMT90] Richard Beckwith, George A Miller, and Randee Tengi. Design and implementation of the wordnet lexical database and searching software. *International Journal on Lexicography*, 3(4):62–77, 1990.

[BPG+01] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Modeling early requirements in tropos: A transformation based approach. In *AOSE*, pages 151–168, 2001.

[Bro03] Der Brockhaus Computer und Informationstechnologie. Brockhaus, Mannheim, 2003.

[CdPL04] Luiz Marcio Cysneiros and Julio Cesar Sampaio do Prado Leite. Nonfunctional requirements: from elicitation to conceptual models. *IEEE Transactions on Software Engineering*, 30(5):328–350, May 2004.

[CEE+04] Kai-Uwe Carstensen, Christian Ebert, Cornelia Endriss, Susanne Jekat, and Ralf Klabunde, editors. *Computerlinguistik und Sprachtechnologie*. Elsevier Spektrum Akademischer Verlag, 2004.

[CKM01] Jaelson Castro, Manuel Kolp, and John Mylopoulos. A requirements-driven development methodology. *Lecture Notes in Computer Science*, 2068, 2001.

[CKM02] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: The tropos project. *To Appear in Information Systems, Elsevier, Amsterdam, The Netherlands*, 2002.

[Coc97] Alistair Cockburn. Structuring use cases with goals. *Journal of Object-Oriented Programming*, 5(10):56–62, 1997.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[EK02] Gerald Ebner and Hermann Kaindl. Tracing all around in reengineering. *IEEE Software*, 19(3):70–77, 2002.

[FHK+05] Norbert E. Fuchs, Stefan Höfler, Kaarel Kaljurand, Fabio Rinaldi, and Gerold Schneider. Attempto controlled english: A knowledge representation language readable by humans and machines. *Lecture Notes in Computer Science*, 3564, 2005.

[Fow04] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*. Addison-Wesley, 2004.

[Gli05] Martin Glinz. Rethinking the notion of non-functional requirements. In *Proceedings of the Third World Congress for Software Quality, Munich, Germany*. Department of Informatics, University of Zurich, September 2005.

[GM90] Derek Gross and Katherine J. Miller. Adjectives in wordnet. *International Journal on Lexicography*, 3(4):265–277, 1990.

[GMP01] Fausto Giunchiglia, John Mylopoulos, and Anna Perini. The tropos software development methodology. *Technical Report No. 0111-20, ITC - IRST. Submitted to AAMAS '02. A Knowledge Level Software Engineering 15*, 2001.

[GPM+01] Paolo Giorgini, Anna Perini, John Mylopoulos, Fausto Giunchiglia, and Paolo Bresciani. Agent-oriented software development: A case study. In *Proceedings of the Thirteenth International Conference on Software Engineering & Knowledge Engineering (SEKE01)*, 2001.

[GPS01] Fausto Giunchiglia, Anna Perini, and Fabrizio Sannicolo. Knowledge level software engineering. *In Springer Verlag, Editor, In Proceedings of ATAL 2001, Seattle, USA. Also IRST TR 011222, Istituto Trentino Di Cultura, Trento, Italy*, 2001.

[Hoe04] Stefan Hoefler. The syntax of attempto controlled english: An abstract grammar for ace 4.0, technical report. Technical Report ifi-2004.03, Department of Informatics, University of Zurich, 2004.

[Kai93] H. Kaindl. The missing link in requirements engineering. *ACM Software Engineering Notes (SEN)*, 18(2):30–39, 1993.

[Kai95] H. Kaindl. An integration of scenarios with their purposes in task modeling. In *Proceedings of the Symposium on Designing Interactive Systems: Processes, Practices, Methods, & Techniques (DIS '95)*, pages 227–235, Ann Arbor, MI, August 1995. ACM.

[Kai96] H. Kaindl. Using hypertext for semiformal representation in requirements engineering practice. *The New Review of Hypermedia and Multimedia*, 2:149–173, 1996.

[Kai97] H. Kaindl. A practical approach to combining requirements definition and object-oriented analysis. *Annals of Software Engineering*, 3:319–343, 1997.

[Kai00] H. Kaindl. A design process based on a model combining scenarios with goals and functions. *IEEE Transactions on Systems, Man, and Cybernetics (SMC) Part A*, 30(5):537–551, Sept. 2000.

[Kai05] Hermann Kaindl. A scenario-based approach for requirements engineering: Experience in a telecommunication software development project. *Systems Engineering*, 8(3):197–209, 2005.

[KGM02] Manuel Kolp, Paolo Giorgini, and John Mylopoulos. A goal-based organizational perspective on multi-agent architectures. In *ATAL '01: Revised Papers from the 8th International Workshop on Intelligent Agents VIII*, LNCS 2333, pages 128–140. Springer, 2002.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction, 3rd ed.* Addison Wesley, 2003.

[KS91] H. Kaindl and M. Snaprud. Hypertext and structured object representation: A unifying view. In *Proceedings of the Third ACM Conference on Hypertext (Hypertext '91)*, pages 345–358, San Antonio, TX, December 1991.

[Kur04] Dominik Kuropka. *Modelle zur Repräsentation natürlichsprachlicher Dokumente*. Logos Verlag Berlin, 2004.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, Englewood Cliffs, NJ, second edition, 2004.

[Lau02] Søren Lauesen. *Software Requirements: Styles and Techniques*. Addison-Wesley, Reading, MA, 2002.

[Lau05] Soren Lauesen. *User Interface Design: A Software Engineering Perspective*. Addison Wesley, 2005.

[Lim04] Q Limbourg. *Multi-Path Development of User Interfaces*. PhD thesis, Université catholique de Louvain, Louvain, 2004.

[MBF+90] George A Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J Miller. Introduction to wordnet: An on-line lexical database. *International Journal of Lexicography*, 3(4):235–244, 1990.

[MC00] John Mylopoulos and Jaelson Castro. Tropos: A framework for requirements-driven software development. *In J. Brinkkemper and A. Solvberg, Editors, Information Systems Engineering: State of the Art and Research Themes. SpringerVerlag*, 2000.

[MCL+01] John Mylopoulos, Lawrence Chung, Stephen Liao, Huaiqing Wang, and Eric Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1):92–96, /2001.

[MCY99] John Mylopoulos, Lawrence Chung, and Eric Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, 1999.

[Mil90] George A. Miller. Nouns in wordnet: a lexical inheritance system. *International Journal of Lexicography*, 3(4):245–264, 1990.

[MKC01] John Mylopoulos, Manuel Kolp, and Jaelson Castro. UML for agent-oriented software development: The tropos proposal. In *UML 2001 - The Unified Modeling Language.Modeling Languages, Concepts, and Tools: Fourth International Conference*, LNCS 2185, pages 422–441. Springer, 2001.

[MKG02] John Mylopoulos, Manuel Kolp, and Paolo Giorgini. Agent-oriented software development. In *Methods and Applications of Artificial Intelligence: Second Hellenic Conference on AI, SETN*, LNCS 2308, pages 3–17. Springer, 2002.

[MM03] Joaquin Miller and Jishnu Mukerji, editors. *MDA Guide Version 1.0.1, omg/03-06-01*. Object Management Group, 2003.

[Mol04] P. Molina. A review to model-based user interface development technolgy. In *Proceedings of the first Workshop on Making Model-Based User Interfaces Practical*, 2004.

[MOW01] Pierre Metz, John O'Brien, and Wolfgang Weber. Against use case interleaving. *Lecture Notes in Computer Science*, 2185:472–486, 2001.

[Moz02] Mozilla. *XUL Programmer's Reference*, 2002.

[Muk06] Kizito S Mukasa. *Model-Based Generation of User Interface Prototypes - A Design Tool for the formal Description of generic and consistent User Interfaces with XML.* PhD thesis, Kaiserslautern University of Technology, Kaiserslautern, Germany, 2006.

[Obj03a] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification, Final Adopted Specification, ptc/03-10-04*, 2003.

[Obj03b] Object Management Group. *OCL 2.0, Final Adopted Specification, ptc/03-10-14*, 2003.

[Obj05a] Object Management Group. *Unified Modeling Language: Infrastructure, version 2.0, formal/05-07-05*, 2005.

[Obj05b] Object Management Group. *Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04*, 2005.

[Obj06a] Object Management Group. *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*, 2006.

[Obj06b] Object Management Group. *Systems Modeling Language (SysML) Specification, version 1.0, ptc/2006-05-03*, 2006.

[PK03] B Paech and K Kohler. Task-driven requirements in object-oriented development. volume 753 of *The International Series in Engineering and Computer Science*, chapter 3, pages 45–68. Springer, 2003.

[SBNS05a] Michał Śmiałek, Jacek Bojarski, Wiktor Nowakowski, and Tomasz Straszak. Scenario construction tool based on extended UML metamodel. *Lecture Notes in Computer Science*, 3713:414–429, 2005.

[SBNS05b] Michał Śmiałek, Jacek Bojarski, Wiktor Nowakowski, and Tomasz Straszak. Writing coherent user stories with tool support. *Lecture Notes in Computer Science*, 3556:247–250, 2005.

[Sim99] A J H Simons. Use cases considered harmful. In *Proceedings of the 29th Conference on Technology of Object-Oriented Languages and Systems-TOOLS Europe'99*, pages 194–203, Nancy, France, June 1999. IEEE Computer Society Press.

[SK94] M. Snaprud and H. Kaindl. Types and inheritance in hypertext. *International Journal of Human-Computer Studies (IJHCS)*, 41(1/2):223–241, July/August 1994.

[Sze96] Pedro Szekely. Retrospective and challenges for model-based interface development. *Design, Specification and Verification of Interactive Systems.*, 1996.

[vLL00] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26(10):978–1005, 2000.

[VPG99] Piek Vossen, Wim Peters, and Julio Gonzalo. Towards a universal index of meaning. In *Proceedings of SIGLEX (Special Interest Group on the Lexicon)*, 1999.

[ZK99] D. Zuehlke and L. Krauss. *Human adapted design of machines and process user interfaces based on WINDOWS.* Shaker, 1999.