UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Institute for Software Technology

Institut für Softwaretechnik

Development of a Glossary System for Artifacts of the Requirements Engineering

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Wirtschaftsinformatik

vorgelegt von

Michael Merz

`www.telekobold.de`

Erstgutachter: Dr. Volker Riediger
Institut für Softwaretechnik

Zweitgutachterin: M.Sc. Katharina Großer
Institut für Softwaretechnik

Koblenz, im Januar 2020

Note

This document differs from the version issued in January 2020 in that my matriculation number has been replaced by a link to my personal homepage.

Hinweis

Dieses Dokument unterscheidet sich von der im Januar 2020 abgegebenen Version dadurch, dass meine Matrikelnummer durch einen Link zu meiner privaten Homepage ersetzt wurde.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

....

(Ort, Datum) (Unterschrift)

Kurzfassung

Im Rahmen der Anforderungserhebung im Requirements Engineering sowie in Softwareprojekten werden häufig *Glossare* erstellt, um projektspezifische Begriffe und ihre Definitionen zu hinterlegen. Solche Glossare werden dann als Nachschlagewerke verwendet, sobald beim Lesen eines projektspezifischen Artefaktes (z.B. einer Anforderung oder einer Dokumentation) Unklarheit über die Bedeutung eines Begriffes auftritt.

Solche Glossare sind jedoch häufig nur als *Begriff-Definitions-Listen* realisiert. Dies hat mehrere Nachteile: *Homonyme* (Begriffe mit mehreren Bedeutungen) und *Synonyme* (Begriffe, die in ihrer Bedeutung übereinstimmen), welche beispielsweise in Anforderungen verwendet werden, können zu Missverständnissen und Inkonsistenzen führen. Beispielsweise kann es sich bei dem Term *bug* entweder um einen Programmfehler, um ein verstecktes Mikrofon oder um ein Insekt handeln. Homonyme und Synonyme lassen sich in als Begriff-Definitions-Listen realisierten Glossaren jedoch nur schwer darstellen. Zudem ist es mit solchen Glossaren nicht oder nur unleserlich durch textuelle Anmerkungen möglich, für das Verständnis nützliche *Relationen* zwischen Begriffen und ihren Definitionen darzustellen, wie beispielsweise Teil-Ganzes-Beziehungen (z.B. zwischen *Stuhl* und *Stuhlbein*). Außerdem werden *thematische Bezüge* zwischen Begriffen und ihren Definitionen häufig streng hierarchisch mithilfe von Überschriften dargestellt und kategorisiert. Solche Überschriften zerstören aber eine einheitliche, alphanumerische Sortierung von Begriffen, welche in den meisten Glossaren zur einfacheren Auffindbarkeit verwendet wird. Diese Probleme erschweren die Erstellung, Nutzbarkeit und Pflege von projektspezifischen Glossaren.

Um die genannten Probleme zu verbessern, wurde im Rahmen dieser Arbeit auf Grundlage von Literaturrecherchen in den Bereichen *Requirements Engineering*, *Linguistik* und *Controlled Vocabulary- und Thesaurus-Standards* ein Metamodell entwickelt, das sogenannte *GlossarySchema*. Das GlossarySchema ist als UML-Klassendiagramm realisiert und definiert eine formale Struktur für Glossare für das Requirements Engineering. Mithilfe des GlossarySchemas lassen sich explizit und eindeutig viele nützliche Eigenschaften von Begriffen und Definitionen sowie nützliche Relationen zwischen Begriffen und ihren Definitionen darstellen.

Ein komplett neuer Ansatz des entwickelten GlossarySchemas ist die Möglichkeit, *Kontexte* zu definieren und zu verwenden. Kontexte sind Gruppierungen von im GlossarySchema als Konzepten abstrahierten Begriffs-Definitionen und anderen Kontexten und ermöglichen eine formale und flexible thematische Gruppierung von Begriffs-Definitionen, welche wesentlich flexibler verwendbar ist als Überschriften.

Zudem wurde das sogenannte *GlossarySystem* entwickelt, ein in der Programmiersprache Java geschriebenes Programm, welches das Erstellen, Editieren und Suchen von auf dem GlossarySchema basierenden Glossaren ermöglicht. Das GlossarySystem stellt seine Funktionaliäten für externe Systeme über eine *Programmierschnittstelle* zur Verfügung.

Ziel des GlossarySchemas bzw. des GlossarySystems ist die Erstellung von Glossaren von einer höheren Qualität, welche einfacher und projektübergreifend verwendet und gewartet werden können.

Abstract

In the context of requirements engineering and software projects, *glossaries of terms* are often created to store project-specific terms and their definitions. Such glossaries are then used as reference works when the meaning of a term becomes unclear when reading a project-specific artifact (e.g. a requirement or documentation).

However, such glossaries are often only realized as *term-definition lists*. This has several disadvantages: *Homonyms* (terms with several meanings) and *synonyms* (terms that have the same meaning), which are used in requirements, for example, can lead to misunderstandings and inconsistencies. For example, the term *bug* can be either a programming bug, a hidden microphone or an insect. However, homonyms and synonyms are difficult to represent in glossaries realized as term definition lists. Moreover, with such glossaries it is not possible, or only illegibly through textual annotations, to represent *relations* between terms and their definitions that are useful for understanding, such as part-whole relations (e.g. between *chair* and *chair leg*). Furthermore, *thematic references* between terms and their definitions are often represented and categorized in a strictly hierarchical manner using headings. However, such headings destroy a uniform, alphanumeric sorting of terms, which is used in most glossaries for easier retrieval. These problems complicate the creation, usability and maintenance of project-specific glossaries.

In order to improve the problems mentioned above, a metamodel, the so-called *GlossarySchema*, was developed within the scope of this thesis on the basis of literature research in the fields of *requirements engineering*, *linguistics* and *controlled vocabulary and thesaurus standards*. The GlossarySchema is implemented as a UML class diagram and defines a formal structure for glossaries for the requirements engineering. With the help of the GlossarySchema many useful properties of terms and definitions as well as useful relations between terms and their definitions can be represented explicitly and unambiguously.

A completely new approach of the developed GlossarySchema is the possibility to define and use *contexts*. Contexts are groupings of term definitions abstracted as concepts in the GlossarySchema and other contexts and allow a formal and flexible thematic grouping of term definitions, which can be used much more flexibly than headings.

In addition, the so-called *GlossarySystem* was developed, a program written in the Java programming language that enables the creation, editing and searching of glossaries based on the GlossarySchema. The GlossarySystem makes its functionalities available to external systems via an *application programming interface* (API).

The objective of the GlossarySchema and the GlossarySystem is to create glossaries of higher quality that are easier to use and maintain across projects.

Contents

List of Figures

Chapter 1

Introduction and problem description

In in the requirements engineering and in software projects, a *glossary of terms* should be defined to store project-specific terms and definitions that are used in the different artifacts of this project. Every term that is used in a project-specific artifact (e.g. a requirement in a requirement list, a wiki entry, or a textual note made during a project meeting) and whose definition deviates from a generally clear definition (such as the definition *a seat for one person, with a support for the back* of the term *chair* [33]) should be defined in that glossary. This serves to make the meaning of the term clear to all those involved in the project and to avoid misunderstandings about the meaning of terms.

A well-defined glossary of terms is especially important in the process of requirements engineering: The use of ambiguous and/or not clearly defined terms in requirements can lead to different understandings of this requirements. A consequence of this can be an implementation of a requirement that is not as intended by the requirements engineer. Moreover, requirements in a project have contractual character. The ambiguity of a requirement which may be caused by an ambiguous term can lead to a different understanding between the contracting parties and to a legal dispute in the worst case. It would therefore be important that glossaries are as comprehensible as possible and are as easy to maintain as possible.

1.1 Problems with current glossaries

However, in most cases, such glossaries are only realized as *term definition lists*, which raises a number of problems:

Homonyms (terms with several meanings) and *synonyms* (terms that have the same meaning) are only difficult to represent in combination with a good findability and maintainability. For example, in the *ECSS Glossary of Terms* [6], to represent the synonymous terms *component* and *part*, the definition of one term is used to refer to the other term or is noted as NOTE. But especially homonyms are critical in glossaries and should receive increased attention: For example, in the requirement *The system shall enable to find the bug*, the term *bug* could refer to a programming error, a hidden microphone, or an insect, which is a source of ambiguity.

Another problem is that useful *relations* between term definition pairs are also hard to represent in a term definition list-like glossary. For example, if a part-whole relation between two terms should be represented, this would also have to be annotated in the textual definition, which makes the definition larger and more incomprehensible. This becomes clear, for example, when looking at the NOTE notes of the ECSS Glossary of Terms [6]: these are at least outside the textual definition, but informal and rather difficult to read. This problem increases with an increase in relations that are represented. This is not as desired since these relations should actually help to increase comprehensibility.

Finally, in conventional glossaries it is difficult to create *thematic groupings* (which are also called *contexts*). There are different approaches to this, but all of them are rather disadvantageous. One approach to this is the thematic grouping using headings, another approach is to write the context to the term in the glossary. Both approaches are used in the ECSS Glossary of Terms [6]:

In section 2.2 of the ECSS Glossary of Terms, terms that correspond to specific contexts (what is referred to as *breakdown of the overall Space System* there) are grouped using *headings*. These terms are repeated in section 2.3 in order to include all terms of the ECSS Glossary in an alphanumerical listing. This illustrates that grouping by headings and alphanumeric sorting contradict each other. In addition, the repetition of the term definition pairs creates redundancy, which impairs the readability and maintainability of the glossary.

In section 2.3 of the ECSS Glossary of Terms, contexts are noted in angle brackets to annotate different meanings of the same term in different contexts. E.g., the term *acceptance* has two different meanings in the "contexts" *act* and *process*. However, this is not a good solution, because on the one hand there are two parallel representations of contexts (headings and the bracket notation), contexts themselves cannot be given properties in this way and this annotation is rather informal and can lead to mistakes and inconsistencies.

These restrictions reduce the maintainability and cross-project usability of glossaries.

1.2 Short analysis of glossaries of ESA System Requirements Documents (SRDs)

To gain further insights about glossaries from real projects, a few requirements documents (SRDs) from the European Space Agency (ESA) were analyzed regarding the structure of their glossaries. It turned out that the glossaries in these documents were realized very differently:

For example, [21] and [27] make a distinction between the listing of term definition pairs and the listing of abbreviations and acronyms, [23] and [26] refer to external documents regarding the glossary, and in [22] there is no overarching reference to glossaries at all (only in section 6.8, a reference is made in a single requirement to the paragraphs 3.1 - 3.3 of [19]).

In [25] again, term definition pairs are annotated in the document in section 1.6 in the same way as requirements, with these requirements being both references to other documents and bundles of definitions. In some cases, requirements are even mixed up with requirements in that a requirement stipulates that several terms are to be used in a certain context, which are then defined under the same requirement ID.

In [24], annexes B, C, D and F correspond most closely to a glossary because of their content. Terms to be defined are mostly noted there as headings, the definitions contain relatively many formulas due to the nature of [24].

The scope of the glossary contents of the examined requirements engineering documents also differs considerably. While e.g. chapter *2.2 Definition* of [27] consists of only 7 short term definition pairs, which take up a quarter of a DIN A4 page, [24] contains 16 pages of term definition content (Appendix B-D) and with Appendix F additionally a three-page list of acronyms.

The formatting of the term definition pairs is also very different. For example, while [21] uses a simple bullet point list and [27] a simple listing, [19] (in sections 3.1 - 3.3) uses a structure similar to the ECSS Glossary of Terms for the definitions of terms. [25] again uses the already mentioned requirements notation for term-definition pairs, alghough these enumerations themselves differ within the chapter.

Finally, in [21] and [24] the lists of concept-definition pairs are separated from the lists of acronyms and abbreviations.

This analysis leads to the conclusion that there is a need for a more structured and consistent solution for glossaries.

1.3 Research Questions

Instead of being subject to restrictions and problems as described in section 1.1, a glossary should allow to formally represent homonymy and synonymy to help people not think they are writing or talking about the same thing even though they are writing or talking about different things. If a homonymous term like *bug* occurs, it should be clear at a glance if a programming error or a hidden microphone or an insect is meant. In addition, it should be possible to formally define further properties of terms and definitions and relations between different term definition pairs and to use these relations for a better understanding of the relationships between terms and their definitions. Furthermore, it should be possible to explicitely represent thematic groupings more formally and flexible than, for example, with headings and angle bracket notes.

The identified problems lead to the following research questions:

RQ1 What should a glossary be like that is useful for requirements engineering?

RQ2 How can contexts be represented in such a glossary?

RQ3 How can such a glossary including contexts be technically implemented?

These research questions will be solved in the next two chapters and answered in the conclusion.

1.4 System Vision

The aim of this work is the development of a system for the *creation and usage of glossaries of terms* (short: *glossaries*) that can be used in combination with artifacts (requirements lists, diagrams, wiki entries, notes, ...) of the requirements engineering. It should enable the definition of structured and more formal glossaries. This includes in particular the definition and explicit representation of relations between glossary entries that are useful in the field of requirements engineering. The result should be glossaries of higher quality that can be better maintained and used across projects.

In a project, especially in a project where software is developed, a *glossary of terms* should be created in order to define the meanings of project-specific terms. Such project-specific terms can occur in various artifacts of a project, e.g. requirements, UML diagrams, wiki entries, or textual notes from project meetings. Members of a project can use the glossary to look up definitions of terms and to define new project-specific terms. The glossary of terms is especially used by requirements engineers to define project-specific terms used in requirements and other artifacts of the requirements engineering and to look up already defined terms. Furthermore, it is used by software engineers who implement those requirements.

1.4.1 Problems with current glossaries of terms

So far, those glossaries of terms are usually realized as simple listings of the form *term - definition* and stored in textual documents. The definitions often have different thematic references and, due to their often very specialized background, a certain complexity, e.g. the term *relative pointing error* from a requirements document of the ESA JUICE mission [25]. For this reason, it would be helpful to explicitly represent certain *relations* between the terms and definitions of a glossary.

Unfortunatelly, such relations are only represented relatively unsystematically, if at all, in textual form. For example, for the term *component* from the ECSS Glossary [6], a note is provided with the information that the term *part* is synonymous to *component* as informal textual information. However, this is still a very ordered example. In other glossaries, such information is included unseparated in the definition text or not contained at all, but is still relevant for the use of the terms.

Conventional, purely textual glossaries often use *alphanumeric ordering* of the contained terms to facilitate the search for a term and its definition. Another frequently used technique is the *thematic grouping* of terms and their definitions using *headings*. Unfortunatelly, one technique weakens the objective of the other technique: Thematic grouping of terms with the help of headings destroys a uniform, alphanumeric listing and thus makes the search more difficult if the searcher is not sure about the thematic classification of the term in the glossary, even if terms are alphanumerically sorted within a section defined by a heading. And a uniform, alphanumeric listing in turn makes it impossible to group terms thematically with the help of headings.

This problem also appears, for example, in the ECSS Glossary of Terms, where terms are listed alphanumerically in one section and according to thematic connection in another section [6]. But listing terms two times makes the glossary more difficult to read: One has to search for a term in one section and then to jump to the other section in order to become clear about the thematic context of the term.

Additionally, with sections and subsections, it is only possible to express simple, hierarchical, thematic groupings. But it is not possible, for example, to express that the terms of a subsection thematically belong to two different fields, where one of these fields cannot be expressed as a subheading of the other field, expressed as superheading. An example from the ECSS Glossary of Terms is given below. In this case, the subsection and all of its terms would have to be repeated under two different sections representing those fields, which would complicate maintainability and readability of the glossary.

These problems also make it difficult to use a glossary across projects, which could be useful, for example, for thematically related projects.

1.4.2 Solutions provided by the system to be developed

In order to improve these problems, a *schema* should be developed in the scope of this work that formally describes the structure of glossaries. This schema should provide a formal definition of relations between terms and definitions that are especially useful for artifacts of the requirements engineering. Furthermore, a system should be developed that should provide the definition and usage of glossaries based on the developed schema. The system gets the straightforward name *GlossarySystem*.

With respect to the above example, in a glossary created with the GlossarySystem, it should be possible to define systematically that *component* and *term* are synonyms. By keeping terms from the ECSS Glossary [6] as examples, it should also be possible, for example, to define that *space segment* is a *supertype* of *segment* and that *space segment* is a *compound term*. Instead of inflating the definition text, this information should even help with the clarity and findability of definitions. The system should support many more useful relations and properties of terms and definitions.

A new idea that should be modelled in the schema and realized in the GlossarySystem is to group terms and their definitions by so called *contexts*. The idea with contexts is to overcome the limitations by the already mentioned headings with which only hierarchical groupings can be defined. It should be possible to nest terms with their definitions and other contexts arbitrarily in contexts. In particular, contexts should help in the search for definitions of terms. The schema to be developed should define how contexts can be used and the GlossarySystem to be developed should support the creation and usage of contexts.

For example, the headings of section 2.2 of the ECSS Glossary of Terms [6] could be used as contexts for the terms under these headings. E.g., *segment* and *component* would be in the context *definitions for generic terms* and both *space segment* and *space segment system* in the context *definitions for space segments.* Subsequently, further contexts could be added. For example, a context *EEE (electrical, electronic and electromechanical) device* could be defined and the terms *component* and *relifing* could be set in this context. Setting *component* in the context *EEE device* would not have been possible using headings, since it does not make sense to define *EEE device* as subheading of the heading *definitions for generic terms* or vice versa. With the GlossarySystem, it should be possible to search, for example, for the term *component* using one of its contexts, e.g. *EEE device* or *definitions for generic terms*.

The GlossarySystem should be connected to an *external electronic dictionary* like the Oxford English Dictionary [17] or the WordNet [32] in order to provide non-project-specific terms and definitions. In this way, if one wants to know about a non-project-specific term that, for example, occurs in the definition of a project-specific term, the medium must not be changed for looking up this term. It may also happen that a project-specific term has not yet been defined, but its project-specific definition would be similar to a common definition of a non-project-specific term. In this case, the non-project-specific term could still help with its general definition or even help with defining a new project-specific term. For example, the definitions of the term *system* from the ECSS Glossary of Terms [6] and from the Oxford English Dictionary [17] are very similar.

The system should not be a closed system, but should make its functionalities available for external systems via an *interface*. Such external systems using the interface could be programs that are useful for the requirements engineering process, e.g. a glossary editor which displays the output of the GlossarySystem as beautiful and clearly as possible or a program that supports users in writing requirements and uses the functionalities of the GlossarySystem for this purposes.

It is planned to develop a prototypical glossary editor within the scope of this work, which will allow to search, define and edit glossaries. This editor should use at least the most central functions of the GlossarySystem's interface.

Chapter 2

Basics

In this chapter, a theoretical background to glossaries and a few technologies that are used in this work are shortly introduced.

2.1 Theoretical background to glossaries

A glossary of terms is used „to provide definitions of key concepts on which everyone should agree“ [14] and thus to define a common language between members of a project. According to van Lamsweerde [14], four central tasks can be identified for a *glossary of terms*:

- providing precise, intelligible definitions of all project specific terms, especially key concepts of a project
- avoiding or reduction of clashes in terminology, designation, and structure
- providing a list of accepted synonyms
- ensuring that the same term does not refer to different concepts and that the same concept is not referred to under different terms.

A glossary can be a good base for the architecture of the system [18]. Moreover, a good glossary is be the basis of a rough information model as a class diagram (domain model) [18].

A good glossary can give newcomers to a system a good overview of the system without further documents, as central terms are explained briefly and concisely. Other artifacts may explain these terms in too much detail for beginners. A well maintained glossary is therefore a good entry point into a system.

2.2 JGraLab

JGraLab, the *Java Graph Laboratory*, is a graph library that implements TGraphs [28] (see 2.3) and provides an API for accessing and manipulation those TGraphs [30].

In the GlossarySystem (see chapter 4), the code generator functionality of JGraLab[1] is used to generate the classes of the GlossarySchema on the persisting/database layer of the GlossarySystem (see section 4.2.1).

[1]see `https://github.com/jgralab/jgralab/wiki/Create-a-Graph`

2.3 TGraphs

As stated in [29] *TGraphs* are „a very general class of graphs“ with (among others) the following properties: All vertices and edges of a TGraph are *attributed* and *typed*. The edges of a TGraph are *directed* but can be *navigated* in both directions. Vertexes and edges can be *inherited* multiple times.

TGraphs are used as persising format by the GlossarySystem with the aid of JGraLab and the classes generated by JGraLab using the TG schema representation of the GlossarySchema (`GlossarySchema.tg` - see also section 4.2.1), generating so-called *TG files*.

2.4 WordNet

As stated in [32], the WordNet is „a large lexical database of English“ that expresses different word forms and their meanings in a formal structure that has a certain similarity to the structure of the GlossarySchema: „Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations.“ [32] The WordNet was developed at and is maintained by the Princeton University.

The WordNet can be used as a reliable source for general definitions of terms. Thus, it should be used as *external electronic dictionary* as stated in the system vision (section 1.4) and in the use case Import Concept using JGWNL (see 2.5) as stated in section 4.4.1 in future work on the GlossarySystem.

In addition, the structure and the properties of the WordNet were used in the development of the GlossarySchema (see chapter 3). Furthermore, the WordNet is frequently used as a source for term definitions in examples in this work, for example in chapter 3 and in the test case glossaries (see appendix D).

2.5 JGraLab WordNet Library (JGWNL)

The *JGraLab WordNet Library* (JGWNL) is „a library for the English language“ that consists of „a database of about 200,000 predefined words, imported from the WordNet“ and „an efficient and convenient application programming interface (API) to access and manipulate the database“ [4]. It was developed at the university of Koblenz-Landau and uses the same persisting format (TGraphs) as the GlossarySystem.

As stated in section 4.4.1, JGWNL should be used to integrate the WordNet as external electonic dictionary into the GlossarySchema in future work.

Furthermore, the JGWNL metamodel [4] was used as source of inspiration in the development process of the GlossarySchema (see chapter 3).

Chapter 3

The GlossarySchema

In this chapter, the structure and semantics of the developed GlossarySchema are explained step by step, derived from literature and illustrated with examples.

The *GlossarySchema* is a schema which can also be described as *metamodel* that allows to represent glossaries mainly for the requirements engineering and software projects as thesauri, developed in the scope of this work. According to ISO 25964-1 [12], a *thesaurus* is a controlled and structured vocabulary that represents concepts (unit of thoughts) by terms (words or phrases), explicitly represents relations between concepts and allows to represent both preferred terms and non-preferred terms for those concepts. The GlossarySchema is modelled as UML class diagram and can be seen as a whole in figure 3.1. It enables the formal representation of properties of and relations between terms and definitions of a glossary.

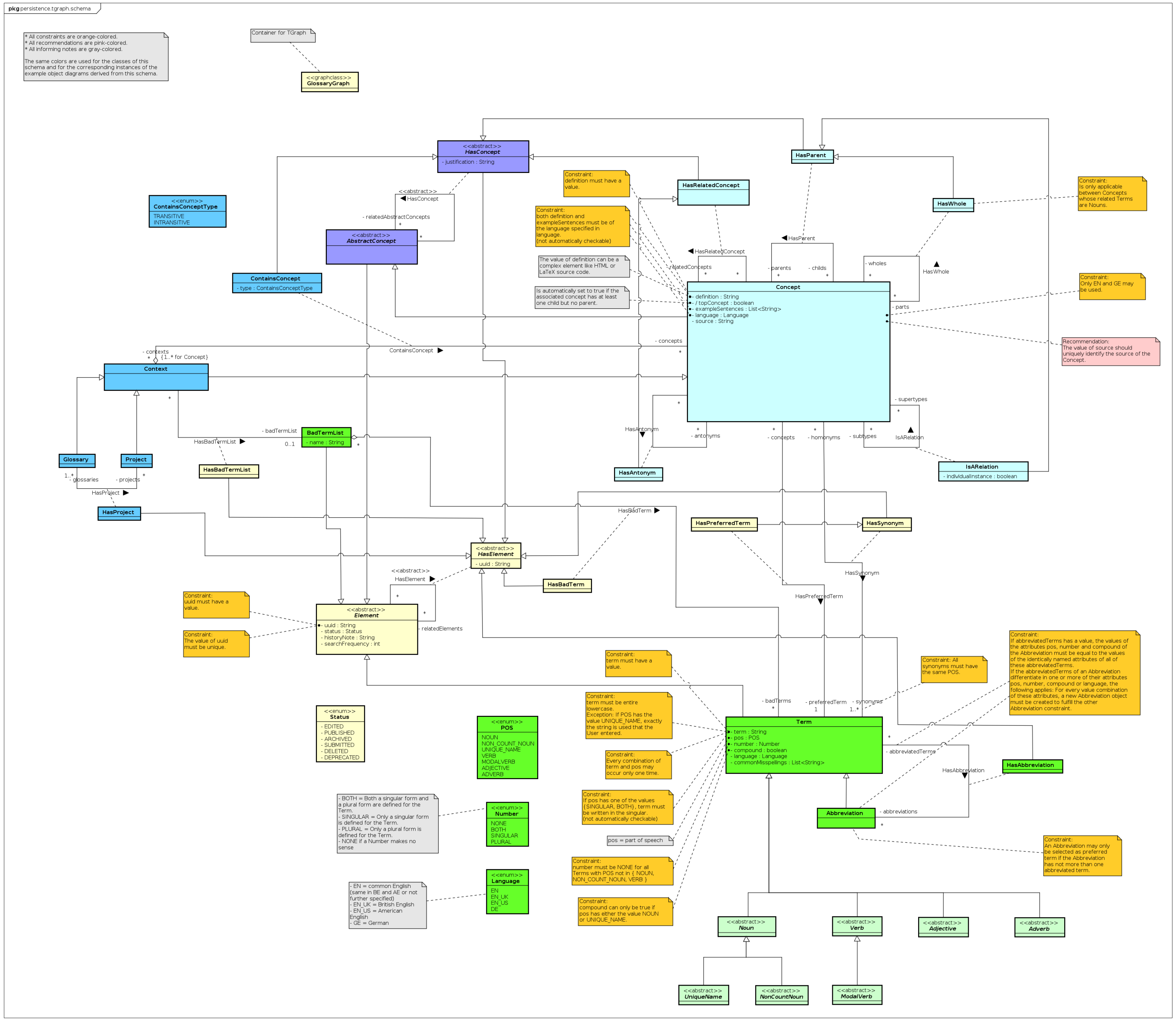
The GlossarySchema makes a fundamental distinction between terms and definitions represented within so-called *concepts*. A *concept* abstracts a specific meaning and has further properties, e.g. one or more terms and relations to other concepts. *Homonyms* (terms with more than one meaning [31]) and *synonyms* (meanings that can be described by more than one term [31]) are represented using this distinction between terms and concepts.

The focus of the GlossarySchema is on concepts by defining most of the relations supported by the metamodel between concepts (and not between terms) and by placing concepts in so-called *contexts* (also rather than terms). A *context* is a grouping of concepts and other contexts and abstracts a special scope or topic. It is realized as special concept in the GlossarySchema.

The new and innovative feature of the GlossarySchema is that it is possible to assign different concepts to different contexts, independently of the terms of these concepts. An example for this can be found in chapter 3.6, figure 3.12. Contexts are explicitely represented and concepts and contexts can be arbitrarily nested in contexts. This is used to overcome the limitations mentioned in the system vision of grouping terms and their definitions using nested headings.

The *GlossarySchema* was developed by evaluating literature from the fields of *requirements engineering*, *linguistics* and *controlled vocabulary and thesauri standards*. It was developed with the intention to make it as usable as possible in projects and artifacts in the requirements engineering and software development.

The GlossarySchema is fully compliant with *ANSI/NISO Z39.19-2005* [2], a standard for the construction, format and management of *controlled vocabularies* (lists of explicitly enumerated



pkg persistence.tgraph.schema
* All constraints are orange-colored.
* All recommendations are pink-colored.
* All informing notes are gray-colored.
The same colors are used for the classes of this schema and for the corresponding instances of the example object diagrams derived from this schema.
Container for TGraph
<<graphclass>> GlossaryGraph
<<abstract>> HasConcept
- justification : String
<<enum>> ContainsConceptType
TRANSITIVE
INTRANSITIVE
<<abstract>> HasConcept
- relatedAbstractConcepts
<<abstract>> AbstractConcept
ContainsConcept
- type : ContainsConceptType
Constraint: definition must have a value.
Constraint: both definition and exampleSentences must be of the language specified in language. (not automatically checkable)
The value of definition can be a complex element like HTML or LaTeX source code.
Is automatically set to true if the associated concept has at least one child but no parent.
HasRelatedConcept
HasParent
HasWhole
Constraint: Is only applicable between Concepts whose related Terms are Nouns.
- relatedConcepts
- parents
- childs
- wholes
- parts
Concept
- definition : String
- / topConcept : boolean
- exampleSentences : List<String>
- language : Language
- source : String
Constraint: Only EN and GE may be used.
Recommendation: The value of source should uniquely identify the source of the Concept.
- concepts
ContainsConcept
- contexts
{1..* for Concept}
Context
- supertypes
- subtypes
IsARelation
- individualInstance : boolean
- antonyms
- homonyms
HasAntonym
- badTermList
BadTermList
- name : String
0..1
HasBadTermList
Glossary
Project
1..*
- glossaries
- projects
HasProject
HasBadTerm
HasPreferredTerm
HasSynonym
<<abstract>> HasElement
- uuid : String
<<abstract>> HasElement
Constraint: uuid must have a value.
Constraint: The value of uuid must be unique.
<<abstract>> Element
- uuid : String
- status : Status
- historyNote : String
- searchFrequency : int
- relatedElements
HasPreferredTerm
HasSynonym
Constraint: term must have a value.
Constraint: term must be entire lowercase. Exception: If POS has the value UNIQUE_NAME, exactly the string is used that the User entered.
Constraint: Every combination of term and pos may occur only one time.
Constraint: If pos has one of the values {SINGULAR, BOTH}, term must be written in the singular. (not automatically checkable)
pos = part of speech
Constraint: number must be NONE for all Terms with POS not in { NOUN, NON_COUNT_NOUN, VERB }
Constraint: compound can only be true if pos has either the value NOUN or UNIQUE_NAME.
Constraint: All synonyms must have the same POS.
Constraint: If abbreviatedTerms has a value, the values of the attributes pos, number and compound of the Abbreviation must be equal to the values of the identically named attributes of all of these abbreviatedTerms. If the abbreviatedTerms of an Abbreviation differentiate in one or more of their attributes pos, number, compound or language, the following applies: For every value combination of these attributes, a new Abbreviation object must be created to fulfill the other Abbreviation constraint.
- badTerms
- preferredTerm
- synonyms
Term
- term : String
- pos : POS
- number : Number
- compound : boolean
- language : Language
- commonMisspellings : List<String>
- abbreviatedTerms
HasAbbreviation
HasAbbreviation
Abbreviation
- abbreviations
Constraint: An Abbreviation may only be selected as preferred term if the Abbreviation has not more than one abbreviated term.
<<enum>> Status
- EDITED
- PUBLISHED
- ARCHIVED
- SUBMITTED
- DELETED
- DEPRECATED
<<enum>> POS
NOUN
NON_COUNT_NOUN
UNIQUE_NAME
VERB
MODALVERB
ADJECTIVE
ADVERB
- BOTH = Both a singular form and a plural form are defined for the Term.
- SINGULAR = Only a singular form is defined for the Term.
- PLURAL = Only a plural form is defined for the Term.
- NONE if a Number makes no sense
<<enum>> Number
NONE
BOTH
SINGULAR
PLURAL
- EN = common English (same in BE and AE or not further specified)
- EN_UK = British English
- EN_US = American English
- GE = German
<<enum>> Language
EN
EN_UK
EN_US
DE
<<abstract>> Noun
<<abstract>> Verb
<<abstract>> Adjective
<<abstract>> Adverb
<<abstract>> UniqueName
<<abstract>> NonCountNoun
<<abstract>> ModalVerb

Figure 3.1: The GlossarySchema

terms, which includes more complex controlled vocabularies as thesauri), since all required and many recommended rules are fulfilled by the GlossarySchema.

ISO 25964-1 [12], a standard for the development of thesauri for information retrieval, and *ISO 25964-2* [13], a standard for the interoperability of thesauri with other vocabularies, would be further useful standards by which the GlossarySchema could be adjusted or supplemented. For reasons of time, these two standards were not worked through. The adaptation and a possible extension of the GlossarySchema to ISO 25964-1 [12] and ISO 25964-2 [13] could be future work.

In the following, the GlossarySchema is explained step-by-step.

For reasons of readability, most of the relations of the sample object diagrams are represented as simple associations rather than displaying an object for every association class. Also for reasons of readability, many attributes of the objects in the object diagrams were not filled with values and some relations required according to the constraints of the schema were not defined.

3.1 `Terms` and `Concepts`

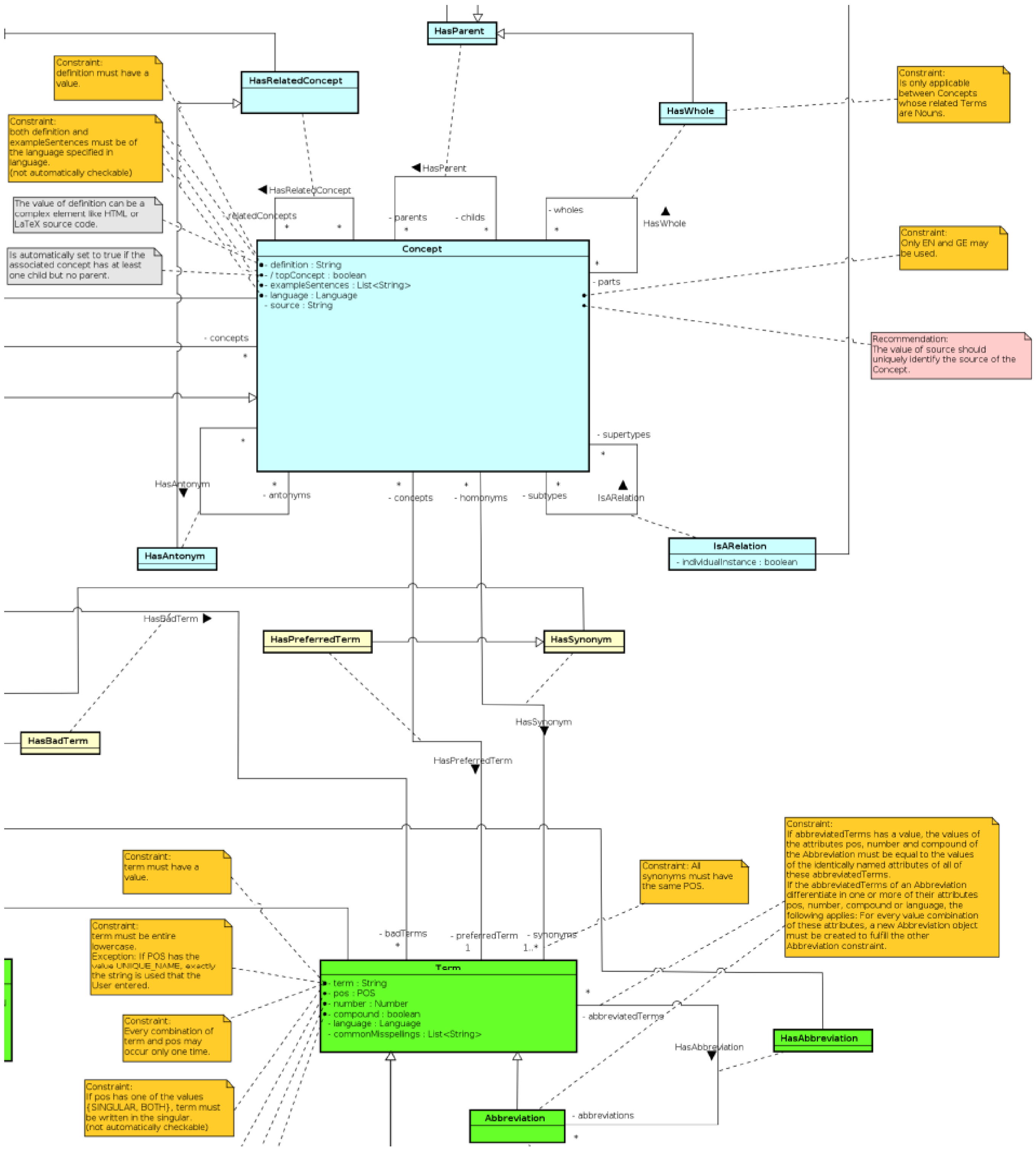
The basic idea of the GlossarySchema is to separate *terms* and *concepts* (see figure 3.2), to put the concepts into *contexts* (which will be explained later in this chapter) and add relations to them. In order to do this, terms are saved as `Term`s and their definitions as `Concept`s. For a `Term` that has n different definitions (a *homonym*), n `Concept`s are added (example: see figure 3.3). For a `Concept` which is known under n terms (these terms are *synonyms*), n `Term`s are added (example: see figure 3.4).

The support of the two relations *homonymy* and *synonymy* and of abbreviations was inspired by the *Glossary* entry from Glinz [11] where these are mentioned as frequent relations in glossaries. The GlossarySchema does not distinguish between *full homonymy* (different meaning, but same spelling and pronounciation) and *homography* (different meaning and pronounciation, but same spelling) like it is done in Ulrich [31] since this linguistic subtleties would go too far.

Polysemy (same term, but different meanings, where the meaning can be derived from the context [31]) can be expressed as normal homonymy relation and assigning different `Context`s to the different `Concept`s of the homonymous `Term`. *Homophony* (different meaning and spelling, but same pronounciation [31]) is threaded as *synonymy* in the GlossarySchema.

The separation of terms and their definitions into `Term`s and `Concept`s is inspired by the *JG-WNL Metamodel* from Bildhauer et al. [4], an extension of the *WordNet* [32] structure. In addition, in Z39.19:2005 [2], the term *concept* is defined as „unit of thought, formed by mentally combining some or all of the characteristics of a concrete or abstract, real or imaginary object. Concepts exist in the mind as abstract entities independent of terms used to express them.“ `Concept`s are thus explicitly independent of `Term`s, which is explicitly modeled in the GlossarySchema (`Term`s do not serve to *cover* `Concept`s but only to *designate* them)[1].

[1]Z39.19:2005 [2], section 11.1.3.1 states that terms *cover* concepts, which is unhappily phrased according to the research in this work.



Constraint:
definition must have a value.

Constraint:
both definition and exampleSentences must be of the language specified in language.
(not automatically checkable)

The value of definition can be a complex element like HTML or LaTeX source code.

Is automatically set to true if the associated concept has at least one child but no parent.

Constraint:
Is only applicable between Concepts whose related Terms are Nouns.

Constraint:
Only EN and GE may be used.

Recommendation:
The value of source should uniquely identify the source of the Concept.

Constraint:
term must have a value.

Constraint:
term must be entire lowercase.
Exception: If POS has the value UNIQUE_NAME, exactly the string is used that the User entered.

Constraint:
Every combination of term and pos may occur only one time.

Constraint:
If pos has one of the values {SINGULAR, BOTH}, term must be written in the singular.
(not automatically checkable)

Constraint: All synonyms must have the same POS.

Constraint:
If abbreviatedTerms has a value, the values of the attributes pos, number and compound of the Abbreviation must be equal to the values of the identically named attributes of all of these abbreviatedTerms.
If the abbreviatedTerms of an Abbreviation differentiate in one or more of their attributes pos, number, compound or language, the following applies: For every value combination of these attributes, a new Abbreviation object must be created to fulfil the other Abbreviation constraint.

Figure 3.2: GlossarySchema excerpt: `Term` and `Concept`

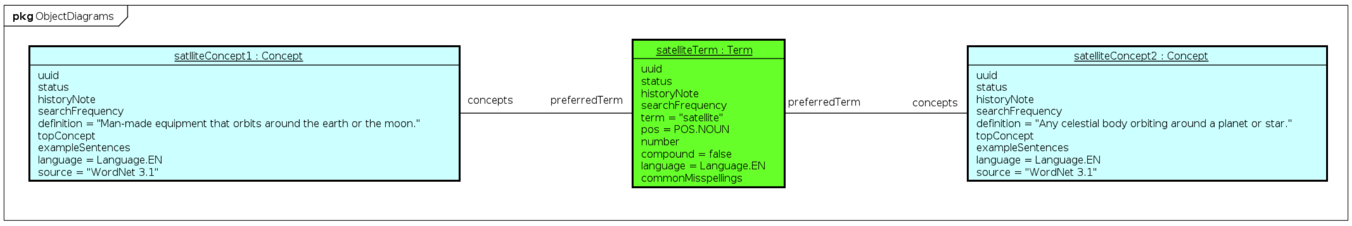


Figure 3.3: Homonymy example

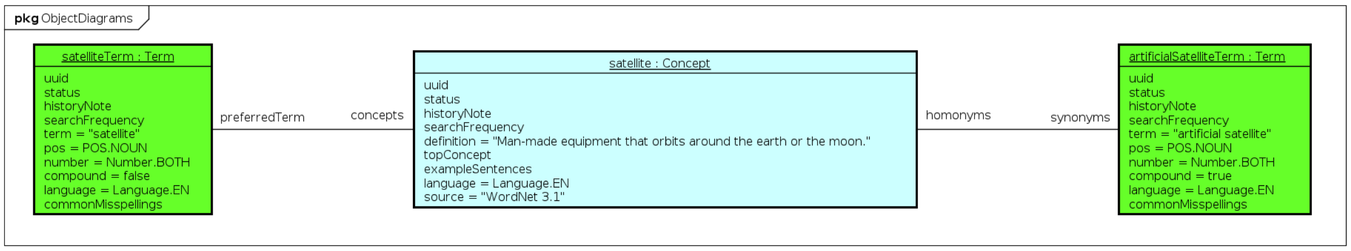


Figure 3.4: Synonymy example

According to van Lamsweerde [14], page 31, a glossary of terms should „ensure that the same term does not refer to different concepts and the same concept is not referred to under different terms". Furthermore, Z39.19:2005 [2] requires that „[a] controlled vocabulary ***must*** compensate for the problems caused by ambiguity by ensuring that each term has one and only one meaning." which would mean that no homonyms are allowed in a glossary.

Due to the previously explained structure of the GlossarySchema, *homonyms* and *synonyms* can exist in the GlossarySystem without this being a problem for clarity or traceability. Since homonyms may exist, it is also possible to insert non-project-specific terms from external, electronic dictionaries, which will be discussed later. Nevertheless, project-specific `term`s should be formulated differently from general `term`s.

In the GlossarySchema, `Term`s function only as "labels" for `Concept`s. `Concept`s are the central entities of the GlossarySchema. The main purpose of `Concept`s is to store the `definition` of at least one `Term`. The `definition` of a `Concept` must be unique.

Since `Concept`s are strictly separated from `Term`s and play the main role, relations, which in the literature are referred to as "between terms" are realized between `Concept`s in the GlossarySchema. Of course, it is implicitly clear in literature as well that these are actually relations between the concepts behind the terms. For example, in Ulrich [31], the definition of *Begriff* (German for *term*) starts with „**Begriff** [auch: Konzept]"[2] (***term*** *[also: concept]* in english) what is a clear reference to that.

Z39.19:2005 [2] requires that „each relationship indicated between **Term A** and **Term B** ***must*** have a corresponding relationship from **Term B** to **Term A**", which is called *reciprocity*. The GlossarySchema follows this rule for all of its relations. In addition, for all *symmetrical* relations (relations that inherit from `HasParent`, `HasParent` itself, and for the `ContainsConcept`), navigability is facilitated by role names on both sides. Unfortunately, this is not possible for

[2]The content in square brackets is part of the quotation and not inserted retroactively.

asymmetric relations (`HasRelatedConcept` and `HasAntonym`) since role names may not appear twice on the same association in UML.

In the GlossarySchema, *preferred terms* are `Terms` that should preferably be used to represent their corresponding `Concepts` in artifacts, e.g. in requirements. An example is given in figure 3.4. Z39.19:2005 [2] requires that „[a] controlled vocabulary ***must*** compensate for the problems caused by synonymy by ensuring that each concept is represented by a single preferred term" and recommends that non-preferred terms should be used to lead to their corresponding preferred terms. Therefore, a `Concept` can only have a single `preferredTerm`. In the GlossarySystem, the `preferredTerm` of a `Concept` is used to represent the `Concept`, which becomes clear, for example, in the prototypical GlossaryEditor.

Z39.19:2005 [2] recommends that if a concept has more than one terms, the preferred term should be the term that the user is most likely to search for. But this can vary in different contexts: E.g. for the term *component* from the ECSS Glossary of Terms [6], it is defined that „[t]he term "part" is preferred when referring to purely mechanical devices" and „[t]he term "component" is preferred for EEE [(electrical, electronic and electromechanical)] devices".

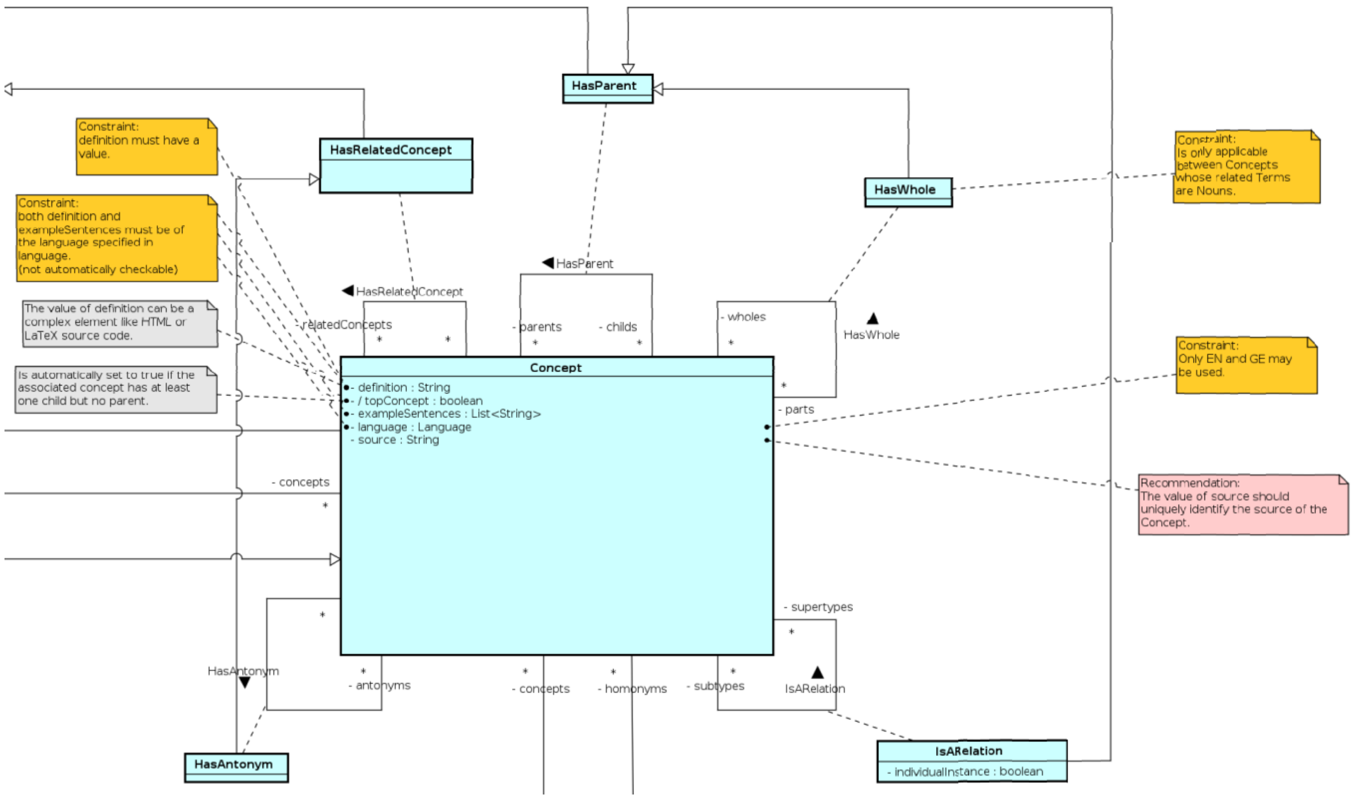
In a glossary derived from the GlossarySchema developed in this work, this can only be realized either through creating an additional `Concept` with a different `preferredTerm` or by all users agreeing on a `preferredTerm` that applies in all `Contexts`. Unfortunatelly, the GlossarySchema cannot define an association between `Contexts` and preferred terms as this could only be modelled using an association between an association class and a normal class. In graph theory, this would be edges between edges, which is not representable using *TGraphs* (the graph type on which the persistence format (TG files) of the GlossarySystem is based, which is explained later).

In addition to Z39.19:2005, source of inspiration for `preferredTerms` is the *postable/non-postable term concept* of the *NASA thesaurus* [16]. In the GlossarySchema, all Terms that are `synonyms` but no `preferredTerms` are implicitly non-preferred terms as defined in Z39.19:2005 [2] (where these are called *entry terms*). Z39.19:2005 [2] recommends that cross references between a preferred term and non-preferred terms should be made. In the GlossarySchema, this is realized transitively since other synonyms or the preferred term can be retrieved via the `Concept` to the synonym or preferred term.

3.2 `Concept` properties that are no self-relations

Every `Concept` must have a `definition`. This can be a simple sentence or a more complex format like HTML, Markdown or LaTeX source text, containing mathematical formulars and diagrams. A weak point of the GlossarySchema is that the content of a `definition` is not formally represented. For example, there is no formal provision for dealing with a `definition` that includes the `term` string of a `Term` being part of the glossary as in figure 3.6.

This problem could be delegated to external systems using (a futurely planned function of) the GlossarySystem (which is discussed later): External systems could use the functionalities of the GlossarySystem and, for example, HTML markup, to link terms to their corresponding concepts in the GlossarySystem. Alternatively or in addition, a more formal definition for the attribute `definition` could be defined in future work.



Constraint: definition must have a value.

Constraint: both definition and exampleSentences must be of the language specified in language. (not automatically checkable)

The value of definition can be a complex element like HTML or LaTeX source code.

Is automatically set to true if the associated concept has at least one child but no parent.

Constraint: Is only applicable between Concepts whose related Terms are Nouns.

Constraint: Only EN and GE may be used.

Recommendation: The value of source should uniquely identify the source of the Concept.

Figure 3.5: GlossarySchema excerpt: `Concept` and its self-relations

The same problem occurs with `exampleSentences` described later in this chapter, for which the two suggested solutions can be applied analogously.

A `Concept` can be a `topConcept` if it has `subtypes` but no `supertypes` (sub- and supertypes are explained below). The representation of `topConcept`s was inspired by the *top term concept* from Z39.19:2005 [2]. (A *top term* (also called *broadest term*) is the top hypernym of a hyponymy hierarchy, which itself does not have a hypernym, e.g. the term *entity*[3] in WordNet.)

For each `Concept` it is possible to provide any number of `exampleSentences`, which should serve to explain the definition of the corresponding `Concept`. Source of inspiration for the possibility to add `exampleSentences` to `Concept`s is the WordNet which provides example sentences for some of it's definitions. Example: The example sentence *a city and its satellite communities* for the adjective *satellite*[4] (with the definition *surrounding and dominated by a central authority or power*) [33]. In the JGWNL metamodel, the example sentences of the WordNet are stored in the the *gloss* attribute of the *Synset* class [4]. Providing `exampleSentences` could also be useful to establish compatibility with the WordNet and to map the WordNet content to the GlossarySchema.

The attribute `language` of `Concept` denotes the language of both attributes `definition` and `exampleSentences` of `Concept`. At present, the GlossarySchema supports the four lan-

[3] `http://wordnetweb.princeton.edu/perl/webwn?o2=&o0=1&o8=1&o1=1&o7=&o5=&o9=&o6=&o3=&o4=&s=entity&i=0&h=0#c`

[4] See `http://wordnetweb.princeton.edu/perl/webwn?s=satellite&sub=Search+WordNet&o2=&o0=1&o8=1&o1=1&o7=&o5=&o9=&o6=&o3=&o4=&h=0.`

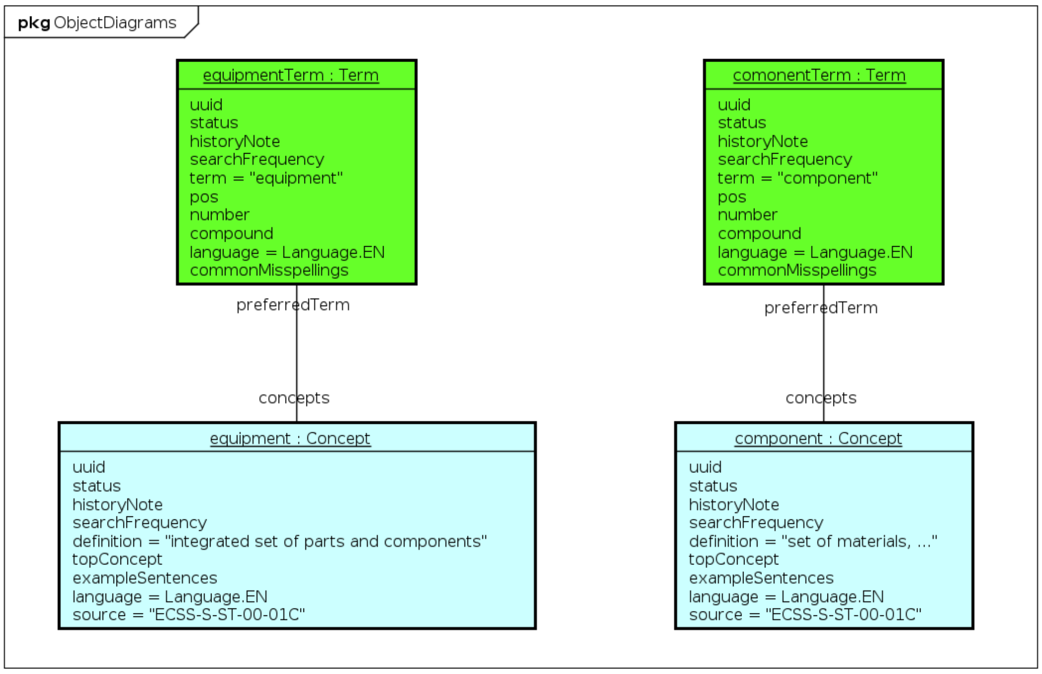


Figure 3.6: Example: The term string of the `Term` *componentTerm* is part of the definition of the `Concept` *equipment*.

guages *English*, *American English*, *British English* and *German*. A small inaccuracy of the schema is that the combination of `definition` and `exampleSentences` of the `Concept` may contain terms in British English as well as terms in American English. Therefore, the `language` indicated for a `Concept` should be either *German* or *English*, but not *British English* or *American English*.

The `source` attribute of `Concept` is used to specify the source from which the `Concept`'s definition originates. It was inspired by the *ECSS Glossary of Terms* [6], in which definition pairs taken from external sources and, if necessary, adapted, are identified by indicating the source in square brackets (e.g. *[ISO 9000:2005]*).

The `source` attribute is only made available for `Concepts`. Maybe the declaration of a source could also make sense for project specific terms and for unique names. This could be future work after application of the GlossarySystem in practice. Furthermore, it could be future work to define a formal structure for `source`.

3.3 Relations between `Concepts`

Z39.19:2005 [2], section 8.4 recommends *to make explicit the nature of the relationships between terms* linked as associative relationships. For this purpose, the GlossarySchema provides *hierarchical* and *associative* relations.

`HasParent` represents all relations that arrange `Concepts` hierarchically. It is derived from the common hierarchical relation concept in Z39.19:2005 [2].

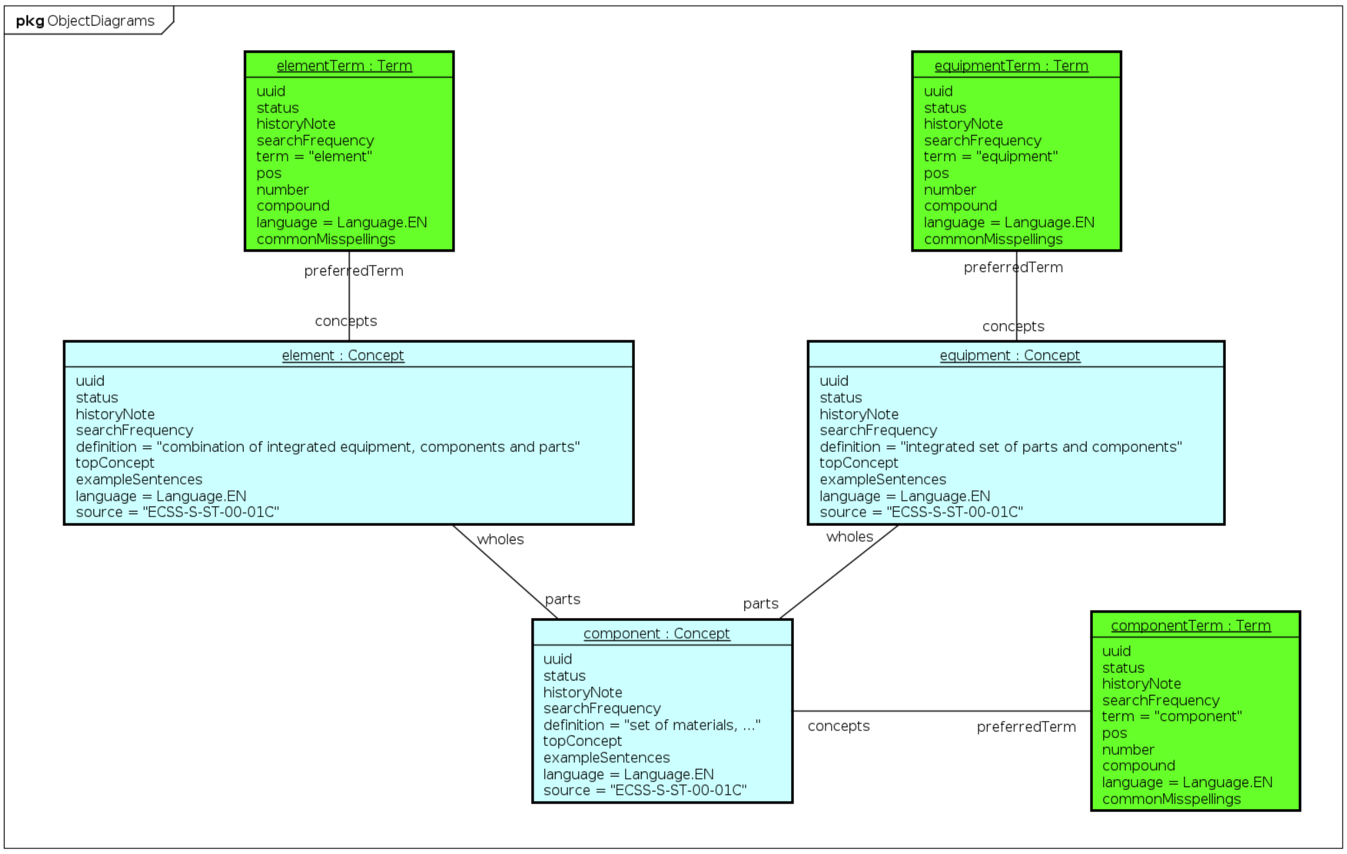


Figure 3.7: Example for a part with multiple wholes: *component* is part of both *element* and *equipment*.

The hierarchical relation `HasWhole` is derived from the *Whole-Part* relationship as described in Z39.19:2005 [2] respectively from the concept of *Holonymy/Meronymy* as described in the WordNet [32] (which is also referred there as *part-whole relation*). According to the WordNet description [32], parts of a meronymy relation are inherited transitively („if a chair has legs, then an armchair has legs as well.“ - regarding the GlossarySchema related to the corresponding `Concept`s of *chair* and *armchair*).

Z39.19:2005 [2] recommends that „the name of the whole and its part(s) ***should not*** have a hierarchical relationship“ if „a whole-part relationship is not exclusive to a pair of terms, i.e., the part can belong to multiple wholes“. For this case, Z39.19:2005 [2] recommends to link the terms (with reference to the GlossaryMetamodel the corresponding `Concepts`) associatively rather than hierarchically. However, since there seem to be many examples where a part can belong to more than one wholes, e.g. figure 3.7 and since the standard did not provide a valid argument for this limitation, this recommendation is ignored in this first version of the GlossarySchema. Maybe the multiplicity of the role whole of the `HasWhole` relation should be set to `0..1` in future work.

The hierarchical relation `IsARelation` combines the two relations *Generic Relationship* as described in Z39.19:2005 [2], section 8.3.1, and *Instance Relationship* as described in Z39.19:2005 [2], section 8.3.2. The only difference between Generic and Instance Relationships seems to be that *Generic Relationship* represents a relationship between a class (e.g. *succulent plants*) and a member or species (e.g. *cacti*), while *Instance Relationship* represents the relationship between „a general category of things [. . .] and an individual instance of category, often a *proper name*“[2]

(*proper names* are a similar concept in Z39.19:2005 [2] to *unique names* in the GlossarySchema), e.g. between *mountain regions* and *Alps*.

This distinction also seems to make sense for the context of requirements engineering. *Individual* instances (e.g. the `Concept` of the first sovjet satellite *Sputnik*, an individual instance of the `Concept` of *artificial satellite*) are distinguished from *common* instances (e.g. the `Concept` of the carrier rocket *Ariane 5*, a special type (subtype) of the `Concept` of *carrier rocket*) using the attribute `individualInstance`. Furthermore, an `IsARelation` defined in the glossary of terms in the phase of requirements elicitation can inspire to define superclasses and subclasses according to the `IsARelation`s in the later implementation phase (if an object oriented language is used for the implementation).

The multiplicity `*` for the role `supertypes` of `IsARelation` allows that a `Concept` can be of different classes. E.g. the `Concept` of *Ariane 5* could be a subtype of both the `Concept` of *space object* and *aircraft*. In object oriented programming, this would be called *multiple inheritance*.

Besides Z39.19:2005 [2], the `IsARelation` is inspired from the concepts of *broader term* and *narrower term* defined and used in the NASA thesaurus [16] and from the concept of *hyponymy* from the WordNet [32] and Ulrich [31]. According to the WordNet documentation [32], Hyponymy (and thus also `IsARelation`) is always transitive, i.e. each hyponym (`subtypes`) inherits its hypernyms (`supertypes`) from its hypernyms (`supertypes`). Example: Since the `Concept` of *bird* is supertype (hypernym) of the `Concept` of *merl* and the `Concept` of *animal* is supertype of the `Concept` of *bird*, the `Concept` of *animal* is also supertype of the `Concept` of *merl* [5].

`HasRelatedConcept` represents all relations that arrange `Concept`s associatively. It is derived from the associative relation concept as described in Z39.19:2005 [2] and from the related term relation in the NASA thesaurus [16].

The relation `HasAntonym` is derived from the concept of *antonymy* as described in Ulrich [31] and in the WordNet documentation [32]. In the GlossarySchema, an antonym is a `Concept` with an opposite meaning to another `Concept`, e.g. between the corresponding `Concepts` to the `Terms` *hot* and *cold*.[6] Unlike in the WordNet [32], antonymy is not only supported for `Concept`s whose corresponding `Term`s are adjectives but (similar to Ulrich [31]) for all parts of speech supported by the GlossarySchema, e.g. for nouns, e.g. *life* $\leftrightarrow$ *dead*.

Z39.19:2005 [2] shows two different ways to realize antonymy: Either as special *near synonym* [7] or as special associative relationship. In the GlossarySchema, the second way was choosen (through `HasAntonym` being a special `HasRelatedConcept`).

The hierarchies of association classes with their interitance relationships serve to make it easy to extend the GlossarySchema at a later point of time. For example, if this turns out to be useful, one could later simply add another relation to `Concept` or add an attribute to an

[5]`http://wordnetweb.princeton.edu/perl/webwn?o2=&o0=1&o8=1&o1=1&o7=&o5=&o9=&o6=&o3=&o4=&r=1&s=merl&i=2&h=100#c`

[6]Ulrich [31] distinguishes three different forms of antonymy, and the WordNet [32] between *full antonyms* and *pertainyms* (also called *indirect antonyms*). These different forms of antonymy are not considered in the GlossarySchema since these doesn't seem to be useful in the context of requirements engineering.

[7]„[T]erm[s] [whould be `Concept`s in the GlossarySchema] whose meaning is not exactly synonymous with that of another term, yet which may nevertheless be treated as its equivalent in a controlled vocabulary. Example: **salinity, saltiness**" [2]. Near synonyms are not modelled in the GlossarySchema.

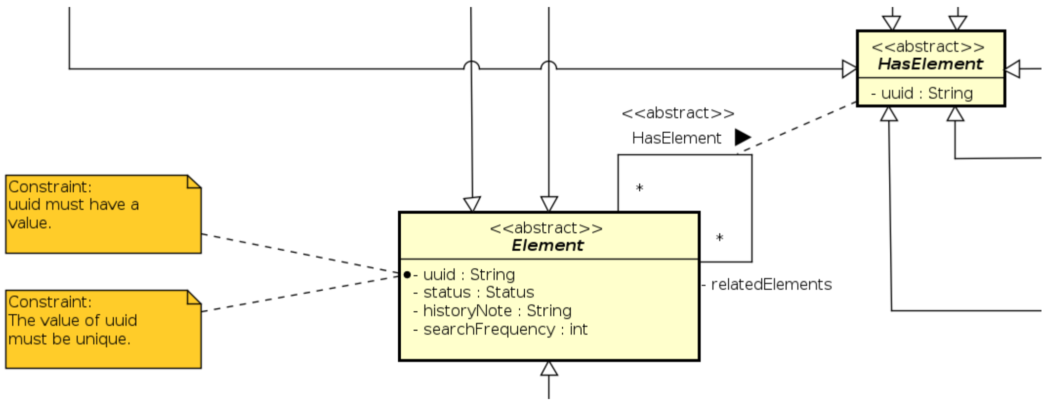


Figure 3.8: GlossarySchema excerpt: `Element` (superclass of `Concept` and `Term`)

existing relation. In addition, this serves for the simple inheritance of common characteristics, and, regarding `Concepts`, the representation of the logical relations of the provided relations between `Concepts` (e.g. the already described fact that `HasAntonym` is a special `HasRelatedConcept`).

It could be argued that the both `Concept` relations *HasParent* and *HasRelatedConcept* should be made abstract so that one has to decide for a concrete hierarchical or associative relation between `Concepts`. Since not all useful hierarchical and associative relations between `Concepts` might have been defined yet, this was not done in this first version of the GlossarySchema in order not to limit the system too much. However, this could be done in the future work after a thorough evaluation of the system and the possible addition of further relations.

3.4 Properties for both `Concepts` and `Terms`

`Concept`'s and `Term`'s attribute `historyNote` can be filled with anything regarding the editing history of the `Concept` or `Term`. It is provided to avoid having to support version control for the schema, which would go far beyond the scope of this work, but still allow time-related notes to be made.

Source of inspiration for `historyNote` is the *history note* concept from Z39.19:2005 [2], section 6.2.3, the recommendation from Z39.19:2005 [2] that „[t]he term record ***should*** note the date of each change and identify the individual responsible for it“ and the NASA thesaurus [16] in which the date of insertion is stored for all new terms inserted after April 1988 (which is called *term-added date*).

Future work might consist of substituting the relatively informal `historyNotes` with the development of a version control system for the GlossarySchema and the GlossarySystem, which could, for example, store in a formal way *when* a `Concept` was changed and *who* changed it. This could help to find out how well a glossary has been or is maintained. This would also comply the recommendation „[i]f a term is modified, the date of the change ***should*** be recorded in the history note [. . .], and a USE reference ***should*** be made from the old form to the new form.“ from Z39.19:2005 [2].

The storage of the persistence format files used by the GlossarySystem in a version control system like Git would not be a practicable solution: On the one hand, the attribute `historyNote` would not be used for the storage. On the other hand, such an external versioning would be quasi a level above the persistence format and thus much too coarsely granular changes in a single commit would be possible. Furthermore, there would be no direct connection between the GlossarySystem and the version control system.

Source of inspiration for the attribute `searchFrequency` is the option shown in Z39.19:2005 [2], section 9.3.1.1, that the number of postings for assigned terms may be displayed. One possible application for `searchFrequency` is that the `searchFrequency` of a `Term` can be used to influence the order in which the GlossarySystem returns the associated `Concepts`, which can be used by the prototypical GossaryEditor and external systems using the GlossarySystem to implement a syntax completion that preferentially suggests `Concepts` whose `Terms` have a high `searchFrequency`. Another possible application is that, as suggested by Z39.19:2005 [2], section 11.1.3.4, „Terms [or `Concepts`] with exceptionally high or low scores can be considered as candidates for modification or deletion."

To store the search frequency centrally is somewhat problematic: All programs that use this value would have to use it in the same way to make it meaningful. However, since a central search frequency would be useful for the above reasons, a better solution for this should be developed in the future.

Z39.19:2005 [2], section 11.1.6 mentiones *candidate terms* („proposed terms that have not gone through all acceptance procedures"). Due to the strict separation of `Terms` and `Concepts`, the GlossarySchema provides both *Candidate `Terms`* and *Candidate `Concepts`* with the aid of `Element`'s `status` attribute.

Figure 3.9 shows the different statuses supported by the GlossarySchema and the possible changes from one status to another. Candidate `Elements` are marked with `EDITED` or optionally with `SUBMITTED` if they should be reviewed before publishing. As required in Z39.19:2005 [2] („[a]s soon as a candidate term is approved as a term, the symbol or phrase ***must*** be deleted"), these two states are changed to `PUBLISHED` as soon as the Candidate `Element` is approved. Every state can be undone (by changing back to `EDITED`). The different states provided also serve to fulfill the requirement from Z39.19:2005 [2] that „[a] deleted or modified term ***may*** be retained in the controlled vocabulary for retrieval or historical purposes only. If it is retained, it ***must*** be marked, e.g., "for retrieval purposes only", and the date of its change in status ***must*** be recorded in the history note and displayed to users.". The last part of this requirement is fulfilled by the above defined attribute `historyNote`.

Furthermore, Z39.19:2005 [2] requires that „[w]henever a term is modified or deleted from a controlled vocabulary, the impact of the change on the ability to search previously indexed database records ***must*** be considered, unless the modified or deleted term has never been assigned". This is ensured by not deleting `Terms` or `Concepts` once they have been inserted into the glossary, but only marking them as `DELETED` or `DEPRECATED`. A `Term` or `Concept` can be marked as `DEPRECATED` if it's used in old artifacts (e.g. requirements) but should not be used in future artifacts. A `Term` or `Concept` is marked as `DELETED` if it was used once but is not used any longer and should not be used any longer.

The different statuses and their possible changes are derived from the statuses supported by the *Module Manual (MoMa) system* [15]. The main difference is that an additional status `DEPRECATED` was added for `Elements` (`Terms` and `Concepts`) that shouldn't be used any more.

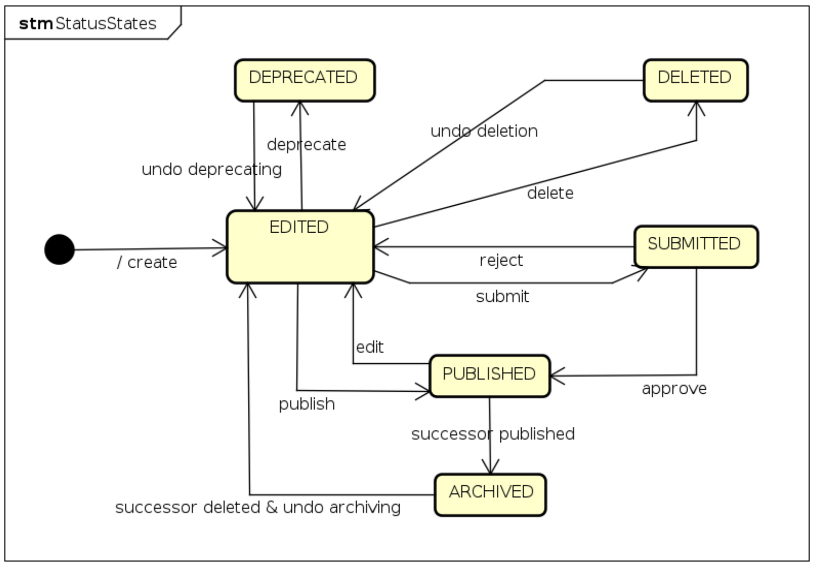


Figure 3.9: The different possible statuses of `Elements` (`Terms` and `Concepts`) and their possible changes

3.5 Properties for `Terms`

Every `Term` has a `term` attribute representing its term string. This term string must be unique.

If both singular and plural are defined for a `Term`, the value of the `Term`'s attribute `term` must be written in the singular. This is different to the recommendation from Z39.19:2005 [2], section 6.5.1, that all count nouns „***should*** normally be expressed as plurals".

The GlossarySchema provides the part of speeches *noun*, *verb*, *adjective* and *adverb*, which also occur in the WordNet [32]. Additionally, *modalverb*, *non-count noun* and *unique name* are supported.

According to Ulrich [31], a *modalverb* is a verb that expresses the conditions for the realization of a verbal event, e.g. *should* in *I should work*. The explicit identification of modal verbs is interesting in the requirements engineering and thus supported in the GlossarySchema, as modalverbs often serve to define the degree of legal obligation in requirements. This is especially true for requirements written according to formal templates like the *MASTeR templates from Sophist* [20], where the modalverbs *shall*, *should* and *will* sign the different kind of legal obligation.

Unique names are used to distinguish the names of common entities (e.g. *satellite*) from the names of unique entities (e.g. the first sovjet satellite *Sputnik*). They are inspired by the concept of *unique entities* from Z39.19:2005 [2] and the concept of *identifiers* from the NASA Thesaurus [16].

The concept of *unique names* and the `IsARelation` should help to distinguish abstract requirements from specific requirements.

Z39.19:2005 [2], section 6.5 recommends to distinguish *count nouns* („names of objects or concepts that are subject to the question "How many?" but not "How much?"", e.g. *books* or

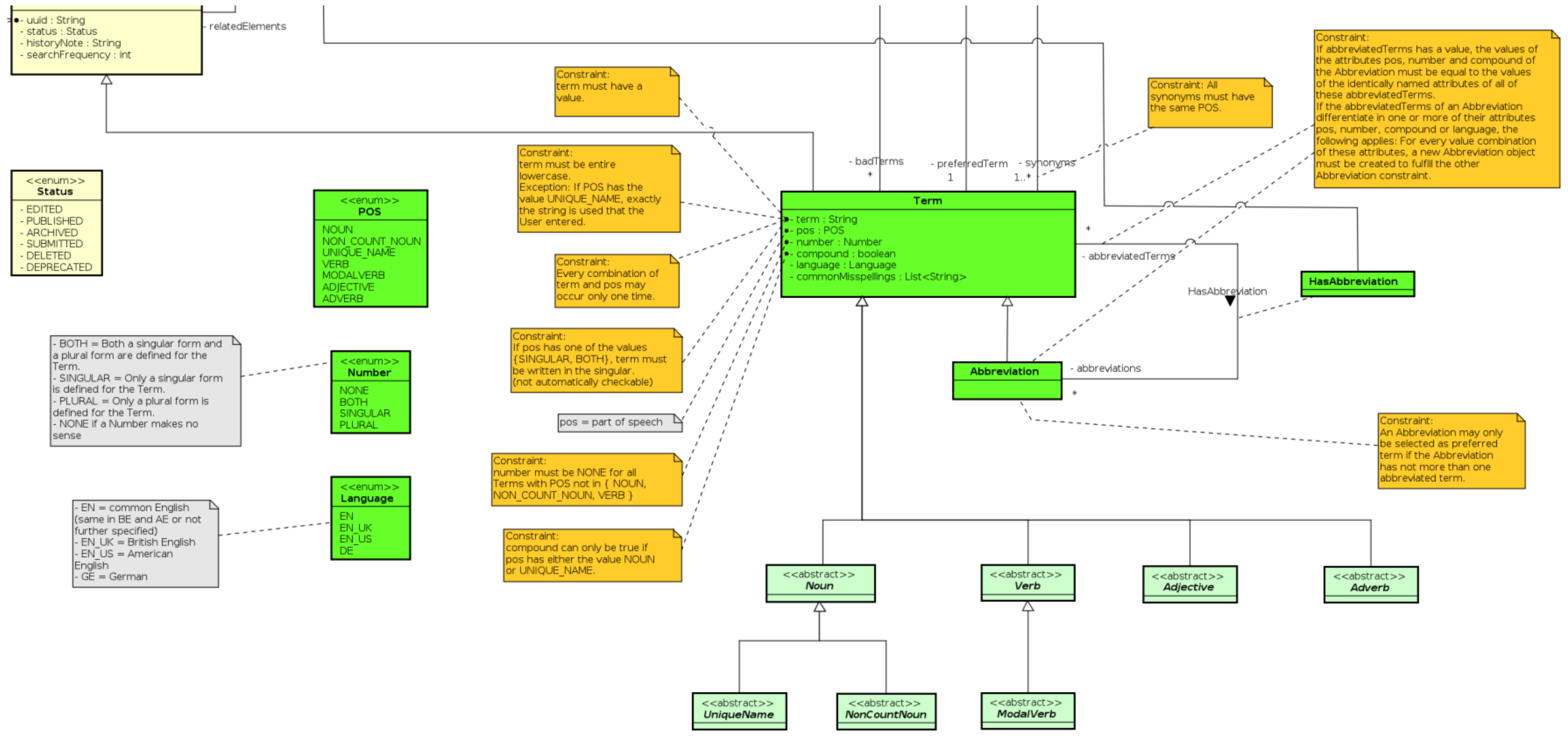

- uuid : String
- status : Status
- historyNote : String
- searchFrequency : int
- relatedElements
<<enum>> Status
- EDITED
- PUBLISHED
- ARCHIVED
- SUBMITTED
- DELETED
- DEPRECATED
<<enum>> POS
NOUN
NON_COUNT_NOUN
UNIQUE_NAME
VERB
MODALVERB
ADJECTIVE
ADVERB
- BOTH = Both a singular form and a plural form are defined for the Term.
- SINGULAR = Only a singular form is defined for the Term.
- PLURAL = Only a plural form is defined for the Term.
- NONE if a Number makes no sense
<<enum>> Number
NONE
BOTH
SINGULAR
PLURAL
- EN = common English (same in BE and AE or not further specified)
- EN_UK = British English
- EN_US = American English
- GE = German
<<enum>> Language
EN
EN_UK
EN_US
DE
Constraint: term must have a value.
Constraint: term must be entire lowercase. Exception: If POS has the value UNIQUE_NAME, exactly the string is used that the User entered.
Constraint: Every combination of term and pos may occur only one time.
Constraint: If pos has one of the values {SINGULAR, BOTH}, term must be written in the singular. (not automatically checkable)
pos = part of speech
Constraint: number must be NONE for all Terms with POS not in { NOUN, NON_COUNT_NOUN, VERB }
Constraint: compound can only be true if pos has either the value NOUN or UNIQUE_NAME.
- badTerms
*
- preferredTerm
1
- synonyms
1..*
Term
- term : String
- pos : POS
- number : Number
- compound : boolean
- language : Language
- commonMisspellings : List<String>
*
- abbreviatedTerms
HasAbbreviation
Abbreviation
- abbreviations
*
HasAbbreviation
Constraint: All synonyms must have the same POS.
Constraint: If abbreviatedTerms has a value, the values of the attributes pos, number and compound of the Abbreviation must be equal to the values of the identically named attributes of all of these abbreviatedTerms. If the abbreviatedTerms of an Abbreviation differentiate in one or more of their attributes pos, number, compound or language, the following applies: For every value combination of these attributes, a new Abbreviation object must be created to fulfill the other Abbreviation constraint.
Constraint: An Abbreviation may only be selected as preferred term if the Abbreviation has not more than one abbreviated term.
<<abstract>> Noun
<<abstract>> Verb
<<abstract>> Adjective
<<abstract>> Adverb
<<abstract>> UniqueName
<<abstract>> NonCountNoun
<<abstract>> ModalVerb

Figure 3.10: GlossarySchema excerpt: `Term`

penguins) and *mass nouns* („names of materials or substances that are subject to the question "How much?" but not "How many?"", e.g. *snow water*, also called *non-count nouns*) through writing count nouns in the plural and mass nouns in the singular. Instead, the GlossarySchema uses the enum literals `NOUN` and `NON_COUNT_NOUN` for this purpose. All `Nouns` which do not have the value `NON_COUNT_NOUN` are considered count nouns.

The part of speech of a `Term` is expressed exclusively using the attribute `pos`. The abstract subclasses of `Term` only serve to improve the readability of the GlossarySchema[8]. The fact that `UniqueName` is a subtype of `Noun` and `ModalVerb` is a subtype of `Verb` is handled by the GlossarySystem, which is explained later.

Source of inspiration for the attribute *number* of Term is the NASA thesaurus [16], where count nouns are represented in the plural and non-count nouns in the singular. Furthermore, it makes sense to know in the context of requirements engineering whether the plural is defined for a term while formulating requirements.

According to Ulrich [31], *Numerus*, a plural is defined for verbs.

`Term`'s `compound` attribute marks *multiword terms*, e.g. *stained glass*, as in Z39.19:2005 [2], section 6.3.1. Actually, a compound `Term` is a special `Term` consisting of other `Terms`. However, for reasons of clarity, this is not modelled explicitely. Instead, a compound `Term` is just marked as compound term without explicit relations to its part `Terms`. [9]. Further example of a compound: The term *masked out wildcard* would be stored in the GlossarySchema as `Term` with the part of speech *noun* that is a compound.

A term can be assigned one of four languages *English*, *British English*, *American English* and *German* using the attribute `language`.

Source of inspiration for the attribute `commonMisspellings` is the option shown in Z39.19:2005 [2], section 9.3.1.1, that common misspellings may be stored in the thesaurus. They can be used as "indirect synonyms" to find `Concept` most likely desired by the user, even if he has made a typing error when entering a corresponding term. E.g. when many users accidentally type "ACOS" instead of "AOCS", "ACOS" should be stored as common misspellings in the `commonMisspellings` attribute of the `Term` *AOCS*.

If a `Term` has at least one `Abbreviation`, this `Abbreviation` is implicitly a *synonym* for this `Term` and vice versa. The same applies if an `Abbreviation` has at least one abbreviated term. This design decision is inspired by Z39.19:2005 [2], page 42, where *UN* and *United Nations* are given as examples for synonyms.

The restriction that an `Abbreviation` can only be selected as preferred term if it has not more than one abbreviated term was made on the basis of Z39.19:2005 [2], section 6.6.3, where it is recommended that „[a]bbreviations and acronyms should be selected as terms only when they

[8]Converting the subclasses of `Term` into concrete classes would not significantly reduce the complexity of the necessary part of speech-related constraints of `Term` and `Abbreviation`.

[9]In the development phase it was considered to design the GlossarySchema in such a way that it would have been possible to explicitly represent the parts of a compound term and the compound term itself as different `Terms`. To realize that, the *Composite Pattern* [10] would have been implemented for `Terms` similar as for `Concepts` and `Contexts`. However, since this would have led to a much more complex GlossarySchema with a rather low expected use in the intended application context at the same time, this was ommited.

have become so well established that the full form of the term or proper name is rarely used. Cross-references should be made from the full forms.". Due to the explicit representation of relations between Terms and Abbreviations and the search functions of the GlossarySystem, even for frequently used Abbreviations such as *AI* (for *artificial intelligence*) it is no problem if the Abbreviation has several abbreviated terms and thus the long form (e.g. *artificial intelligence*) is used as preferred term.

3.6 Contexts

A Context is a special Concept that can be used to group and to categorize Concepts and other Contexts. Concepts and Contexts can be contained in other Contexts. In the GlossarySchema, this is implemented using the *Composite Pattern*[10] [10]. A Context can contain any number of Concepts (especially no Concept) and any number of other Contexts, but each Concept that is not a Context must be contained in at least one Context.

It is important that a Context does not need to be in at least one other Context in order not to prevent that there are root Contexts which are in no other Contexts. Especially a Glossary or a Project could be such a root Context (Glossary and Project are discussed later in this chapter).

As already mentioned in the introduction of this chapter, with Contexts, it is possible to assign Concepts to different Contexts, independently of these Concept's Terms, which is the new and innovative feature of this work. An example for this is given in figure 3.12, which demonstrates the usability of explicitely modelling Contexts and assigning Concepts to Contexts.

A Context is modelled as a special Concept since, for example, the mere name of a Context seems quite limiting to clarify it's purpose. But Concepts already have a lot of useful properties that can be used to describe relations that are useful in the context of requirements engineering. Additionally, it is easier this way to implement the conversion of a Concept into a Context if an existing Concept should be used to group other Concepts and/or Contexts, and the opposite way (the conversion of a Context into a Concept, e.g. for the case that contained Concepts and/or Contexts have been deleted).

A Concept or Context A can be either transitively or intransitively contained in another Context B. If A is *transitively* contained in B, this means that A is contained in B itself and in all other Contexts in which B is contained. If A is *intransitively* contained in B, this means that A is contained exclusively in B. Thus, complex Context nestings can be modelled in different ways.

If A is *intransitively* contained in B, this semantically means that the textual definition of B should be applicable to A and all *bad terms* (see below) of B are also bad terms of A if A is a Context. If A is *transitively* contained in B, this is also true between A and all Contexts C in which B is contained.

[10]The use of the Composite Pattern in the GlossarySchema deviates from the standard Composite Pattern in that the *composite* (Context) interits from the *leaf* (Concept) and not from the *component* (AbstractConcept). This makes the implementation of the GlossarySystem easier: Otherwise, Concepts contained in a Context would be returned as AbstractConcepts (and could not be casted to Concepts).

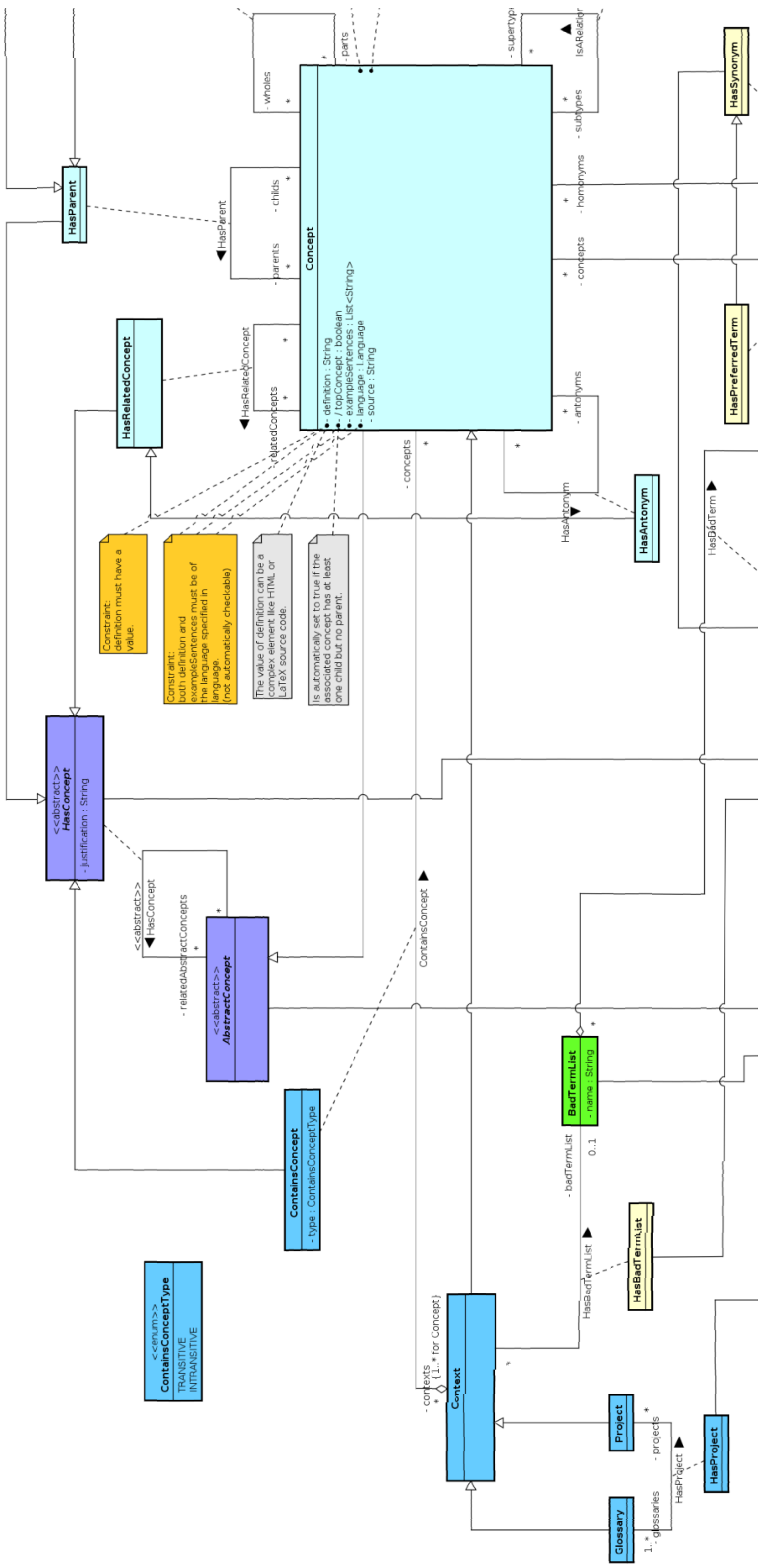


Figure 3.11: GlossarySchema excerpt: `Context` and `Concept`

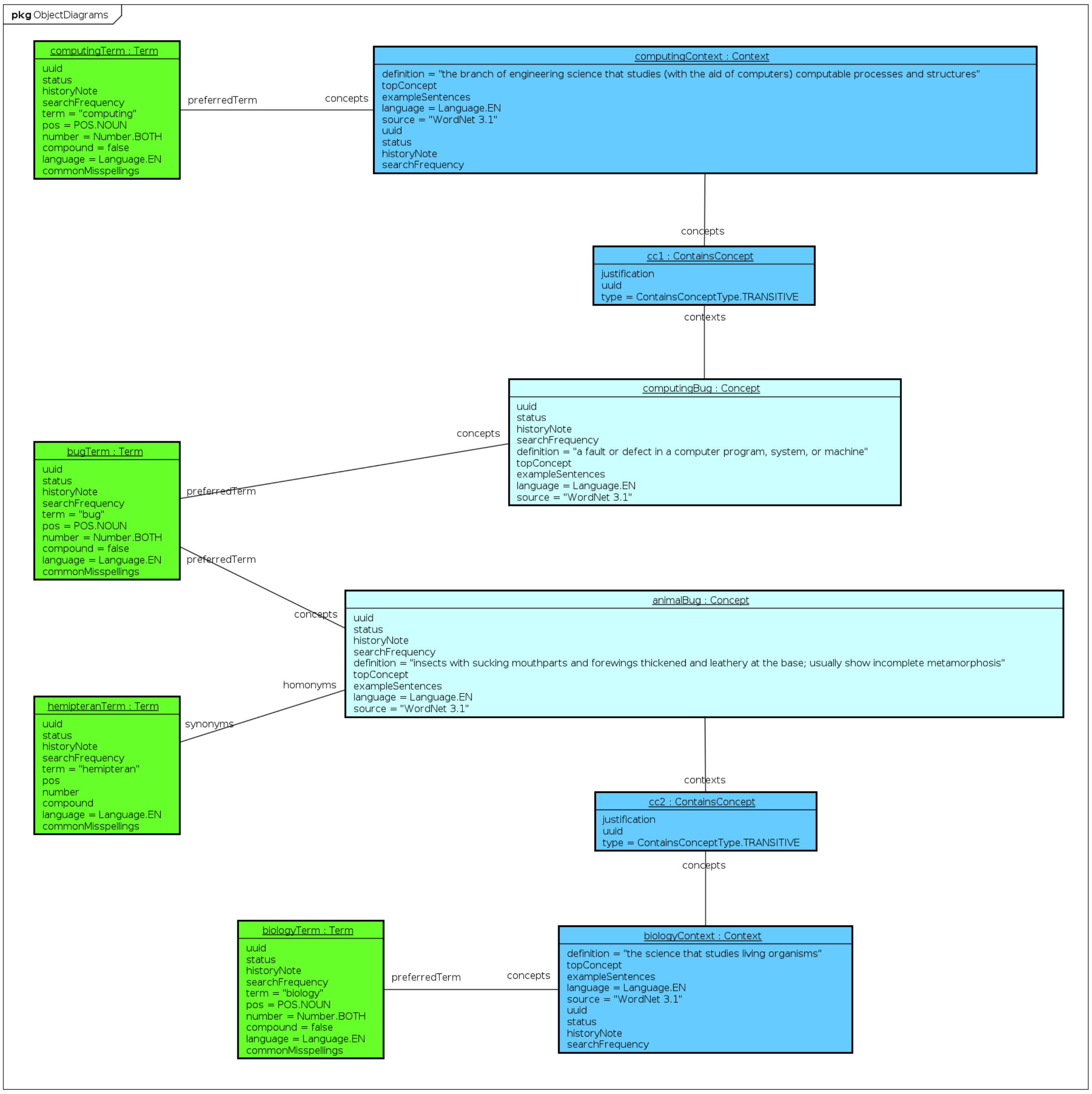


Figure 3.12: Example for the usefulness of `Contexts`: There are two different `Concepts` for the same `Term` *bug* that are in different `Contexts`.

If a `Concept` or `Context` A is contained in another `Context` B and B is not contained in another `Context` C, it does not matter whether A is transitively or intransitively contained in B. This applies, for example, to the relations in figure 3.12, since the `Concept`s and `Context`s there are not linked to any other `Concept`s or `Context`s. Furthermore, it can happen that the `ContainsConceptType` should be adapted when inserting new elements into a glossary to avoid non-elegant, complex `ContainsConcept` constructs.

The following example should demonstrate the use of an *intransitive* contains relation (the used terms and definitions were taken from the ECSS Glossary of Terms [6]): The contexts *definitions for generic terms* in which the concept of *segment* is contained and *definitions for space segment* in which the concepts of *space segment* and *space segment system* are contained could be in another context *ECSS Glossary*. ECSS Glossary again could be in a context *ECSS documents*. But it could be argued that if someone thinks about the context *ECSS documents*, he or she thinks about the different ECSS document types and concrete documents, but not about special content of those documents. Thus, it could be desirable that just *ECSS Glossary* is contained in *ECSS documents*, but not the both contexts *definitions for generic terms* and *definitions for space segment* and their concepts.

However, this is more of a constructed example. No example for the usage of an intransitive `ContainsConcept` relation could be found, which would not look constructed. But this concept does exist theoretically and is therefore provided by the GlossarySchema (but not used by the GlossarySystem, as stated in 4.6).

In the literature, many relations „between terms“ (between the concepts behind these terms) are defined. For example, it can be expressed that an entity A is a part of another entity B (a part-whole relation) or that A has an opposite meaning to B (antonymy relation). However, every of these relations is only applicable in a relatively special scope.

`Context`s are a much more general concept than the relations provided between `Concept`s. `Concept`s and `Context`s can be grouped in other `Context`s for all possible reasons, which seem meaningful to the user of a GlossarySystem.

When doing some things in a project, e.g. writing requirements, creating a diagram or writing some code, terms and their concepts used in these artifacts are implicitly always in a certain thematic area. Such thematic areas can be abstracted very well by the `Context`s of the GlossarySchema. For example, in the requirement *KR-MIS-070* of the *ESA CHEOPS Mission System Requirements Document* [21] („KR-MIS 070 The CHEOPS spacecraft shall perform the nominal operations in a circular Sun-Synchronous Orbit (SSO).“), the concept behind the term *spacecraft* is implicitely in the context *CHEOPS mission*. However, in the requirement *SRD-MIS-2* of the *Euclid System Requirements Document* [22] („SRD-MIS-2 Spacecraft consumables shall be sized for 7 years in orbit.“), the concept behind the term *spacecraft* is implicitely in the context *Euclid mission*. When searching for a certain `Concept`, e.g. for the `Concept` behind the term *spacecraft*, it would be nice to pass this `Context` as search parameter. The developed GlossarySystem, which is described in chapter 4, provides this possibility through providing search functions that take the ***current context*** as a search parameter.

In Glossaries, contexts (thematic groupings) are often annotated implicitly in the definitions of terms. Furthermore, there are some approaches regarding a more explicit annotation of contexts. For example, in the ECSS Glossary of Terms [6], contexts are annotated in angle brackets behind terms that have different meanings in different contexts, e.g. *acceptance <act>* and *acceptance <process>*. Furthermore, the subsections of section 2 of the ECSS Glossary of Terms also annotate contexts for the terms grouped below these headings.

However, explicitely modelling those contexts and allowing arbitrary nestability of concepts and contexts in other contexts (excluding self-containment) as it is done in the GlossarySchema gives a much higher flexibility which is comparable with tagging things (adding keywords to different content where this content can be found using this keywords), but with a much higher formality and much more power, especially through `Context`s being special `Concept`s.

In the ECSS Glossary of Terms [6], a part of the term definition pairs from one section (Section 2.3 of [6], which is alphanumerically sorted according to the terms due to an easy findability) are repeated in another section (Section 2.2 of [6]) according to a defined hierarchy (shown in Figure 2.1 of [6]) with the help of headings and writing down the terms in a specific order. With the aid of the GlossarySchema, it is possible to *combine* the two objectives *representation of structure* and *easy searchability of terms*. Furthermore, it is possible to represent complex, non-transitive nestings of `Concept`s and `Context`s in other `Context`s which cannot be represented with headings and writing down terms and definitions in a specific order.

This is demonstrated by the program `CreateECSSGlossaryGraph.java`, which was developed in the scope of this work and can be found in the package `de.uni_koblenz .glossary_system.glossary_creator` in the `src` directory of the GlossarySystem's project directory. The program reads out the ECSS Glossary of Terms [6] PDF file and maps it to the GlossarySchema. Furthermore, it generates contexts based on the headings in chapter 2 of the ECSS Glossary of Terms and using the abovely mentioned contexts annotated in angle brackets and assigns all these contexts accordingly. The output file has the name `ecss-glossary.tg`, is stored in the `createdGlossaries` directory in the GlossarySystem's project directory and can be queried using the Prototypical GlossaryEditor, which is described in chapter 4.

In the GlossarySchema, there is no need for a direct linking between Context and Term. As mentioned above, the GlossarySchema instead places `Concept`s in `Context`s, while `Term`s are only labels for concepts. This is best illustrated by Figure 3.12. Regarding the abovely mentioned example from the ECSS Glossary of terms, the two different meanings of *acceptance* from the ECSS Glossary of Terms would be represened by a single `Term` *acceptance* and two different `Concepts`, where one of these `Concept`s could be set in context *act* and the other in context `process`. This is also done by `CreateECSSGlossaryGraph.java`.

The ECSS Glossary of Terms [6], section 2.2 and `CreateECSSGlossaryGraph.java` also help to make clear the usefulness of `Context`s being special `Concept`s: On page 10, there is the heading "Definitions for space segment" and directly below the term-definition pair "space segment". Among other things, `CreateECSSGlossaryGraph.java` creates `Context`s using the headings from section 2 of the ECSS Glossary of Terms and `Concept`s using the term-definition pairs from section 2 of the ECSS Glossary of Terms. For example, `CreateECSSGlossaryGraph.java` takes both the heading "Definitions for space segment" and the term-definition pair "space segment" and creates a single `Context` with the preferred term *space segment*. For all term-definition pairs below the heading "Definitions for space segment", `Concept`s are created that are placed in this `Context`.

`Glossary` and `Project` are realized as special `Contexts`. `Project` is a `Context` that is expected to be used so frequently that it is modeled as an explicit subclass of `Context` in addition to `Glossary`. It should serve to aggregate all `Concept`s used in a specific project.

A `Glossary` can be related with any number of `Project`s. There may be `Glossaries` that do not depend to a specific `Project`. A `Project` must be related with at least one `Glossary`. Under certain circumstances, there may be `Project`s in which more than one `Glossary` is used despite the fact that the GlossarySchema and the GlossarySystem are designed to use `Glossaries` across projects.

The GlossarySchema supports the concept of *bad terms*, which are terms that should not be used in a specific context. `badTerms` depend on `Contexts` and thus on `Glossaries` and `Projects` so that a `Term` can be modelled as bad term in one or more specific `Contexts` (while not being a bad term in other `Contexts`).

The concept of bad terms is inspired by Berry et al. [3], section 3.3.5, where „linguistic vagueness" is described as a risk in software requirements, since „it is not clear how to measure whether the requirement is fulfilled or not". According to Berry et al. [3], „[a] statement is considered vague if it admits borderline cases" and describes *fast response time* as example of a vague statement, because *fast* cannot be measured since it is a relative term. In *Space engineering - Technical requirements specification* [7], section 8.3.3, such terms are listet, e.g. *relevant*, *necessary*, and *etc.*. The `badTermsList` of a `Context` serves to denote all such `Terms` that should not be used in this `Context` for the reasons stated above.

In Berry et al. [3] a non-functional, vague reqirement is defined as requirement that has „no precise way of describing and measuring it" or as requirement with „arbitrary quantification". The use of the previously named terms may lead to such ambiguous requirements. Therefore, it should not be used and included in the `badTermList` of a corresponding `Context`.

3.7 Further aspects of the GlossarySchema

The inheritance hierarchies of the GlossarySchema serve to improve its extensibility. For example, the class `BadTermList` was added relatively late in the modelling phase of this work. To give `BadTermList` a UUID, it was simply derived from `Element`. A further example would be the later addition of another associative relation, which can be realized relatively easily by deriving this new relation from `HasRelatedConcept`.

With `HasConcept`'s attribute `justification`, the GlossarySchema provides the possibility to justify every defined relation between `Concepts`. For example, in complex technical domains, it might be useful to explain why a certain `Concept` has been placed in a `Context` to make the glossary easier to understand.

The attribute `justification` is inspired by the concept of *node labels* from Z39.19:2005 [2], which is applicable there to broader and narrower terms (which is covered by the `IsARelation` in the GlossarySchema, as already mentioned). In Z39.19:2005 [2], the concept is illustrated with the following example: *diesel cars* and *electric cars* are narrower terms of *cars by motive power*, while *racing cars* and *sports cars* are also narrower terms of *cars by purpose*. *by motive power* and *by purpose* are the particular node labels. This cannot be expressed by assigning the `Concepts` of *diesel cars* and *electric cars* into a `Context` *motive power* and the `Concepts` of *racing cars* and *sports cars* into a `Context` *purpose* since in the GlossarySchema, `Contexts` refer to whole `Concepts` and `Contexts` and not to specific relations between `Concepts` like `IsARelation`.

Furthermore, `justification` helps to fulfill the recommendation from Z39.19:2005 [2], section 8.4 „to avoid subjective judgments" about associative relationships „as much as possible" to avoid inconsistencies.

It was considered to provide a relation "translates" for concepts, which could have been defined between two concepts, where one concept would have been the exact translation of the other

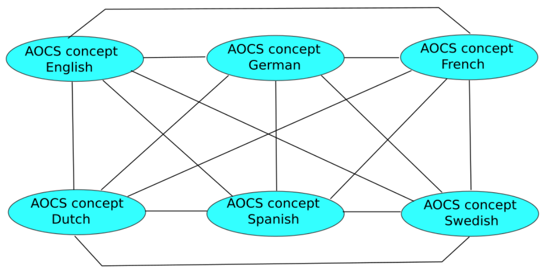


Figure 3.13: Translates relation links between translations of the same concept without a language-independent concept.

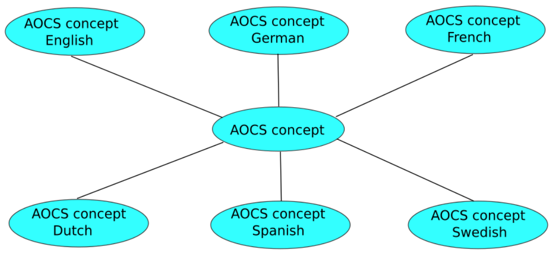


Figure 3.14: Translates relation links between translations of the same concept if a language-independent concept is used.

concept. Due to the fact that only English is supported in the GlossarySystem and due to the problems described below, this relation has been omitted.

Another point of extension of the GlossarySchema could be the `Language` enum, whereby additional things have to be taken into account: To add additional languages in future work, a possibility should be found to relate `Concepts` that are translations of each other. If, for example, only the two languages *English* and *German* are supported, a simple translates relation between two concepts would be a sufficient solution. However, in a multinational organisation such as ESA with 22 member states[1], the support of several languages in a glossary used for different projects could be useful or even necessary. However, if such translation would be realized as a simple `Concept`-self-relation as, for example, the `HasWhole` relation, increasing the number of languages would require increasing the multiplicities of the translates relation. Thus, there would be a translates relation between each `Concept` in a specific language and each other `Concept` that represents a translation for that `Concept` in another language as, for example, in Figure 3.13 for 6 languages. For n languages, this would be $n * \frac{n-1}{2}$ translates relations, which does not seem to be a good solution.

A solution to overcome this problem could be to create some sort of language-independent concept for each concept and, in addition, a further concept for each translation of the concept, including a translates relation to the language-independent concept. That would reduce the number of translates relation links to a maximum of the number of languages (assuming that there is a single concept representing the translation into a single language), as shown in figure 3.14.

To define the constraints of the GlossarySchema, the *Object Constraint Language (OCL)* [5] could have been used. OCL could have been useful, since it is more formal and precise than text in natural language. Further, it can be used for code generation [5] and free tools exist for that, e.g. *Eclipse OCL* [9]. However, the constraints of the GlossarySchema were not defined in OCL, because OCL is much less intuitive than text written in natural language, which would make the GlossarySchema much more difficult to read. In addition, the GlossarySchema was translated into the TGraph syntax by hand, and JGraLab does not support OCL. Since the use of OCL could therefore not be used for code generation, its usage would not have been an advantage. But possibly OCL could be useful in further development of this work.

Some further potentially useful elements and relations could have been added to the GlossarySchema. However, some elements and relations have not been added so as not to make the GlossarySchema too detailled and thus incomprehensible.

3.8 Related work for GlossarySchema elements

This chapter briefly describes some approaches from the literature, which are similar to approaches of the GlossarySchema, but were not used as inspiration, but solved differently.

Z39.19:2005 [2] names *qualifier* (defining terms noted in parentheses behind terms) as a way to distinguish *homographs* (as already described a special homonym with different meaning and pronounciation, but same spelling; in the GlossarySchema homographs are not distinguished from other forms of homonymy). Example: *Mercury (metal)* and *Mercury (planet)*. With the GlossarySchema, this is realized by generating and assigning different `Concept`s and `Context`s for the same `Term` in the GlossarySchema. With regard to the example given in Z39.19:2005 [2]: The `Term` *Mercury* would be assigned two corresponding `Concept`s. In addition, one of these `Concept`s could be assigned to a `Context` *metal*, the other to a `Context` *plantes*.

Furthermore, Z39.19:2005 [2] names *scope notes* („A note following a term explaining its coverage, specialized usage, or rules for assigning it.“). The purposes for scope notes described in Z39.19:2005 [2], section 6.2.2, however, are implemented in the GlossarySchema as follows:

- If a `Term` has different applications, a new `Concept` is added.
- The distinction between `Term`s that have overlapping meanings in natural language is handled by the separation of `Term`s and `Concept`s.
- An advice for `Term` usage is given through providing preferred terms.

For the term *illumination*, which is exemplarily listed in Z39.19:2005 [2], section 6.2.2, 3 different `Concept`s with corresponding `Context`s would be created in a glossary derived from the GlossarySchema.

Z39.19:2005 [2], section 6.3.2 states that concepts can be grouped into general types, e.g. *painting* and *sweing* are *activities or processes* and *birthdays* and *civil wars* are *occurrences*. In the GlossarySchema, such general types can be expressed by `Context`s.

Chapter 4

The GlossarySystem

The *GlossarySystem* is a program developed in the scope of this work that implements the GlossarySchema described in chapter 3 and makes its features accessible through providing useful functionalities. It provides the possibility to create, edit and search glossaries based on the GlossarySchema.

The structure of this chapter is as follows: First, the use cases for the system are presented and explained. The detailled requirements to the GlossarySystem can be found in appendix B. Then, the architecture and implementation details of the GlossarySystem and the single methods of the *GlossaryAPI* (the central interface of the GlossarySystem) are introduced. The fulfillment of the constraints of the GlossarySchema are listed and detailly explained in appendix C. Then, the *Prototypical GlossarySearcher* (a user interface that uses the GlossarySystem) is presendet. After that, some remaining work to the GlossarySystem and the Prototypical GlossarySearcher is enumerated. Finally, some further remarks on the GlossarySystem are mentioned.

4.1 Use Cases

4.1.1 Use Cases for the role *User*

Search Concept

If a user wants to search for a concept in a glossary, he should first set the current context (see Set Current Context) using its unique identifier. Then he passes a search term that is (probably) contained in the concept's *preferred term* or in an abbreviation of the preferred term or in one of the concept's *synonyms* (or in *the* synonym if the concept just has one) or in an abbreviation of one of its synonyms or in the concepts *definition* or in one of the concept's *example sentences* or in the concept's *source*. In response, he receives a list from the GlossarySystem containing all concepts that contain the passed search term in one of these elements and are simultaneously in the current context.

The concepts in the list are ordered according to their particular *nesting distance* to the current context. If this nesting distance is the same for two or more concepts, these concepts are sorted according to the position of the passed search term in their components. The priority is as follows: preferred term or an abbreviation of a preferred term, definition, synonyms or an abbreviation of a synonym, example sentences, source. If this position is also the same for

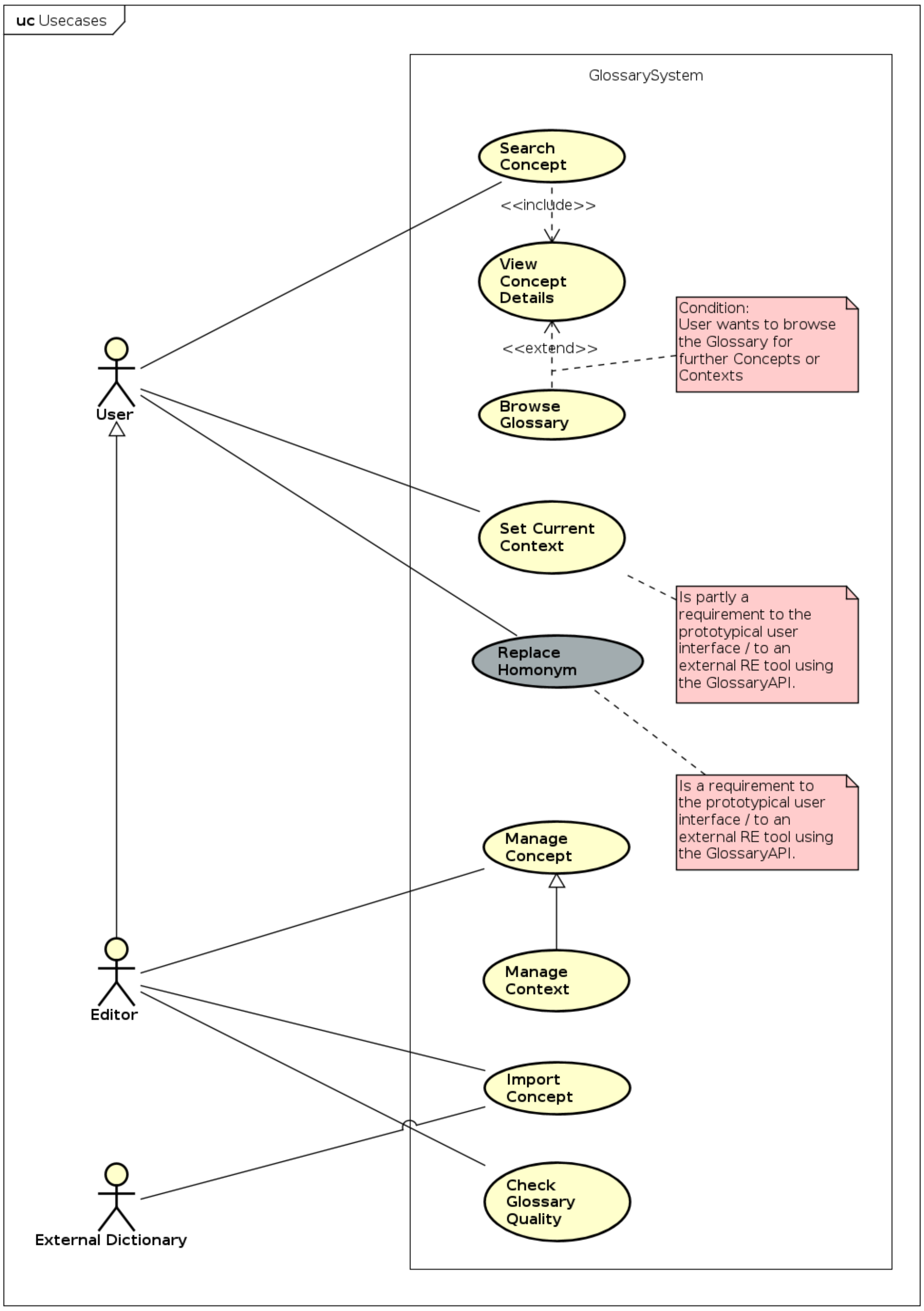

uc Usecases
GlossarySystem
Search Concept
<<include>>
View Concept Details
<<extend>>
Browse Glossary
Condition: User wants to browse the Glossary for further Concepts or Contexts
Set Current Context
Is partly a requirement to the prototypical user interface / to an external RE tool using the GlossaryAPI.
Replace Homonym
Is a requirement to the prototypical user interface / to an external RE tool using the GlossaryAPI.
Manage Concept
Manage Context
Import Concept
Check Glossary Quality
User
Editor
External Dictionary

Figure 4.1: Use cases for the GlossarySystem.

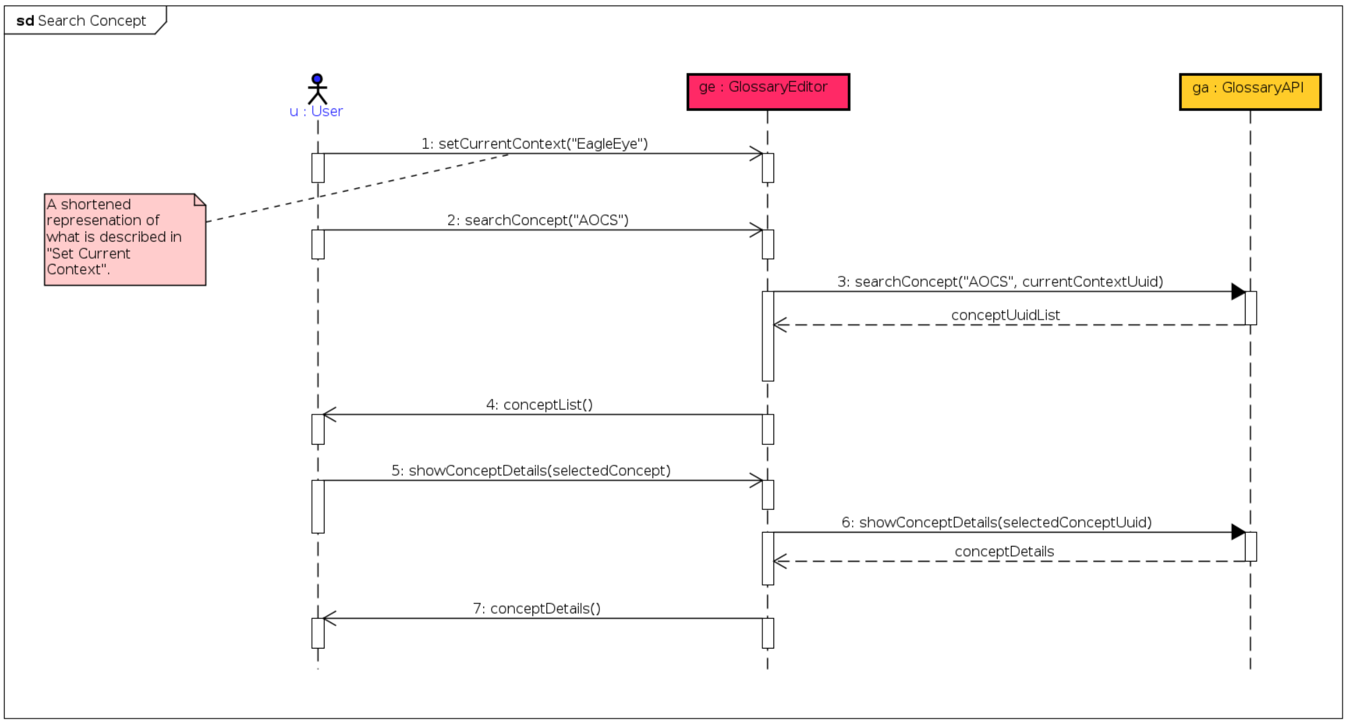

sd Search Concept
u : User
ge : GlossaryEditor
ga : GlossaryAPI
A shortened represenation of what is described in "Set Current Context".
1: setCurrentContext("EagleEye")
2: searchConcept("AOCS")
3: searchConcept("AOCS", currentContextUuid)
conceptUuidList
4: conceptList()
5: showConceptDetails(selectedConcept)
6: showConceptDetails(selectedConceptUuid)
conceptDetails
7: conceptDetails()

Figure 4.2: An exemplary search for a concept with the term “AOCS”.

two or more concepts, these concepts are sorted alphanumerically according to their preferred terms.

The user then selects a concept from the list from which he wants to get details displayed.

When passing a search term, upper and lower case and leading and trailing whitespaces of this search term do not play a role. For example, “AOCS”, “Ao cs” and “aOcs” all return the corresponding concept with the synonym “AOCS”.

In addition, it is possible to filter the returned concepts by the *part of speech* of the concept’s preferred term and the concept’s synonyms and/or to search only specific elements of the concept for the passed search term. For example, it is possible to return only such concepts that have at least one preferred term or synonym of type *noun*[1] or to search only preferred terms and their abbreviations and definitions of the concepts for the passed search term.

The GlossarySystem also allows searching for a concept without specifying the current context. In this case, the GlossarySystem returns all concepts associated within any contexts, as long as the other passed parameters match these concepts as stated above. Furthermore, the GlossarySystem allows to search only for concepts that are contexts.

An additional, useful function of the GlossarySystem is that all concepts, contexts, terms, abbreviations, homonyms and synonyms contained in a glossary can be output and displayed. In the prototypical implementation in the scope of this work, this is limited to a certain number of concepts respectively contexts respectively terms per call.

[1](all preferred terms and synonyms of a concept should be of the same part of speech)

View Concept Details

View Concept Details is the next step following *Search Concept*: The user is shown all details of the concept that was searched for. These include the definition of the concept, it's preferred term and it's synonyms, example sentences, the language of the definition and the example sentences, it's source, and contexts associated with the concept and relations to other concepts. Possible relations to other concepts are the hierarchical relations *part-whole relation* and *instance relations* and the associative relations *common associative relation* and *antonymy relation*. Thus, another concept may be a *child*, *parent*, *part*, *whole*, *subtype*, *supertype*, *related concept* or *antonym* to the concept.

contexts are special concepts that serve as containers for concepts and other contexts. For contexts, all associated contexts and bad terms are shown in addition to the other concept elements mentioned above.

Browse Glossary

Browse Glossary is a possible extension of *View Concept Details* facilitating navigation through the glossary using concepts and contexts: If, for example, a concept is selected which has an antonymic relationship to the originally searched concept, again its details can be shown. The same applies to contexts in which the searched concept is contained. In addition, the concepts and contexts and their details of the associated contexts and related concepts and again their associated contexts and related concepts and so on can be displayed.

Set Current Context

When creating an artifact in the RE, e.g. writing requirements or modelling a sequence diagram, one is always in a certain context (which is called the *current context*). This current context is usually implicit clear to the person creating the artifact (abstracted in the GlossarySystem by the role *user*). For example, in the requirement *KR-MIS-070* of the *ESA CHEOPS Mission System Requirements Document* [21] („KR-MIS 070 The CHEOPS spacecraft shall perform the nominal operations in a circular Sun-Synchronous Orbit (SSO).“), the concept behind the term *spacecraft* is implicitely in the context *CHEOPS mission*. However, in the requirement *SRD-MIS-2* of the *Euclid System Requirements Document* [22] („SRD-MIS-2 Spacecraft consumables shall be sized for 7 years in orbit.“), the concept behind the term *spacecraft* is implicitely in the context *Euclid mission*. To get the the intended concept from the glossary the user must first tell the GlossarySystem about the current context.

For setting the current context, this context must first be searched in the glossary to be sure that this context is already part of the glossary. This works either as described in Search Concept (with the difference that only concepts are returned that are contexts) or through passing the unique identifier of the context to the GlossarySystem, if it is known. If the first way is choosen to search for the context and if the GlossarySystem returns more than one context, the user must select the intended context.

Then, this context can be used to pass it as current context when searching for a concept. External systems could also query the current context and cache it and use it every time a search is performed, until the user changes this current context.

Set Current Context is an important part of *Search Concept* to get meaningful results.

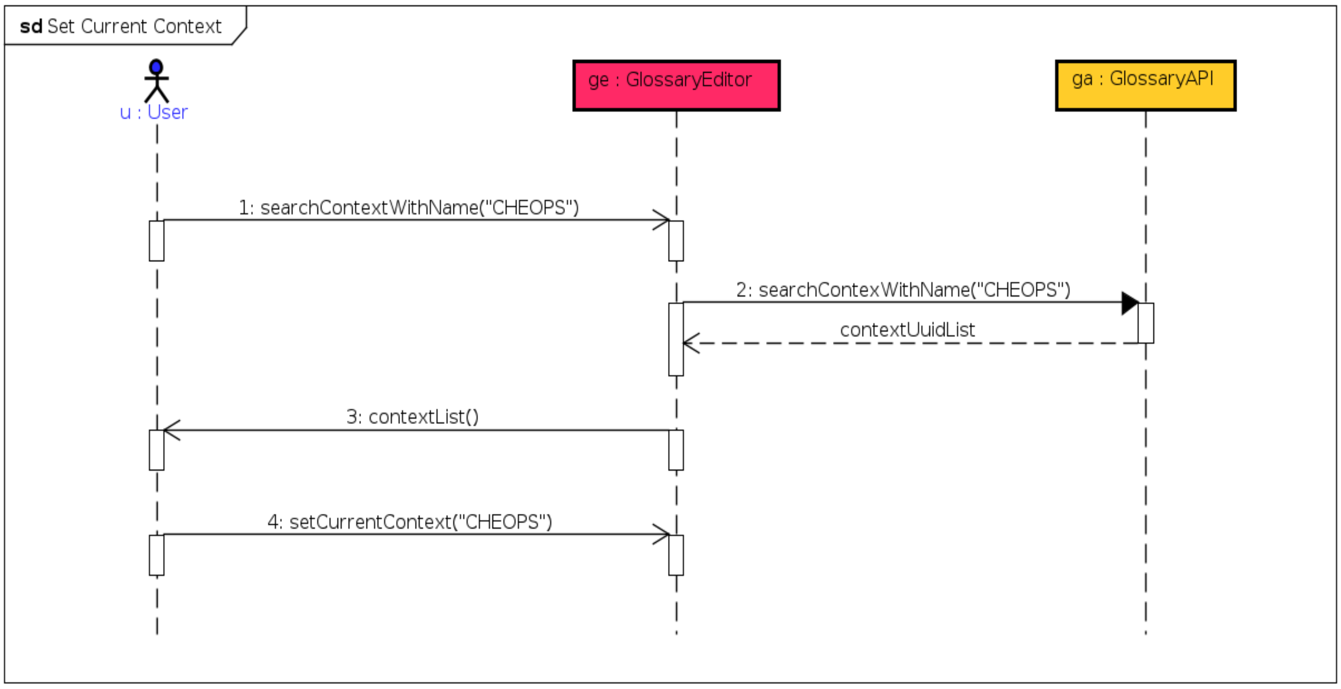

sd Set Current Context
u : User
ge : GlossaryEditor
ga : GlossaryAPI
1: searchContextWithName("CHEOPS")
2: searchContexWithName("CHEOPS")
contextUuidList
3: contextList()
4: setCurrentContext("CHEOPS")

Figure 4.3: Setting the context *CHEOPS* as the current context.

Replace Homonym

For the reasons explained in Chapter 3.1, homonyms should be avoided in the glossary, despite the fact that, due to the GlossarySchema, they can be presented in such a way that the risk of ambiguity is significantly reduced. For this reason, using the GlossaryAPI, the Prototypical GlossaryEditor (and hopefully other external systems) provide a function to detect and gradually remove all homonyms. At the push of a button, all homonyms in the glossary are displayed. These can then be removed step by step by renaming the individual terms.

4.1.2 Use Cases for the role *Editor*

Manage Concept

Manage Concept includes *creating*, *editing*, *deleting*, *converting*, and *linking* a concept.

To *create* a concept, the user must at least specify a definition and a preferred term for the new concept. Additionally, he can specify a status, an arbitrary number of synonyms, a language, a source, an arbitrary number of example sentences.

Then the new concept has to be assigned to at least one context and can be assigned to an arbitrary number of contexts. Before doing this, this context must be either searched in the glossary or redefined. The first version of the GlossarySystem only supports *transitive* context assignments.

It is also possible to relate the concept to an arbitrary number of other concepts. Possible relations can be found in the description of the use case View Concept Details. Before doing this, this concept must also be either searched in the glossary or redefined.

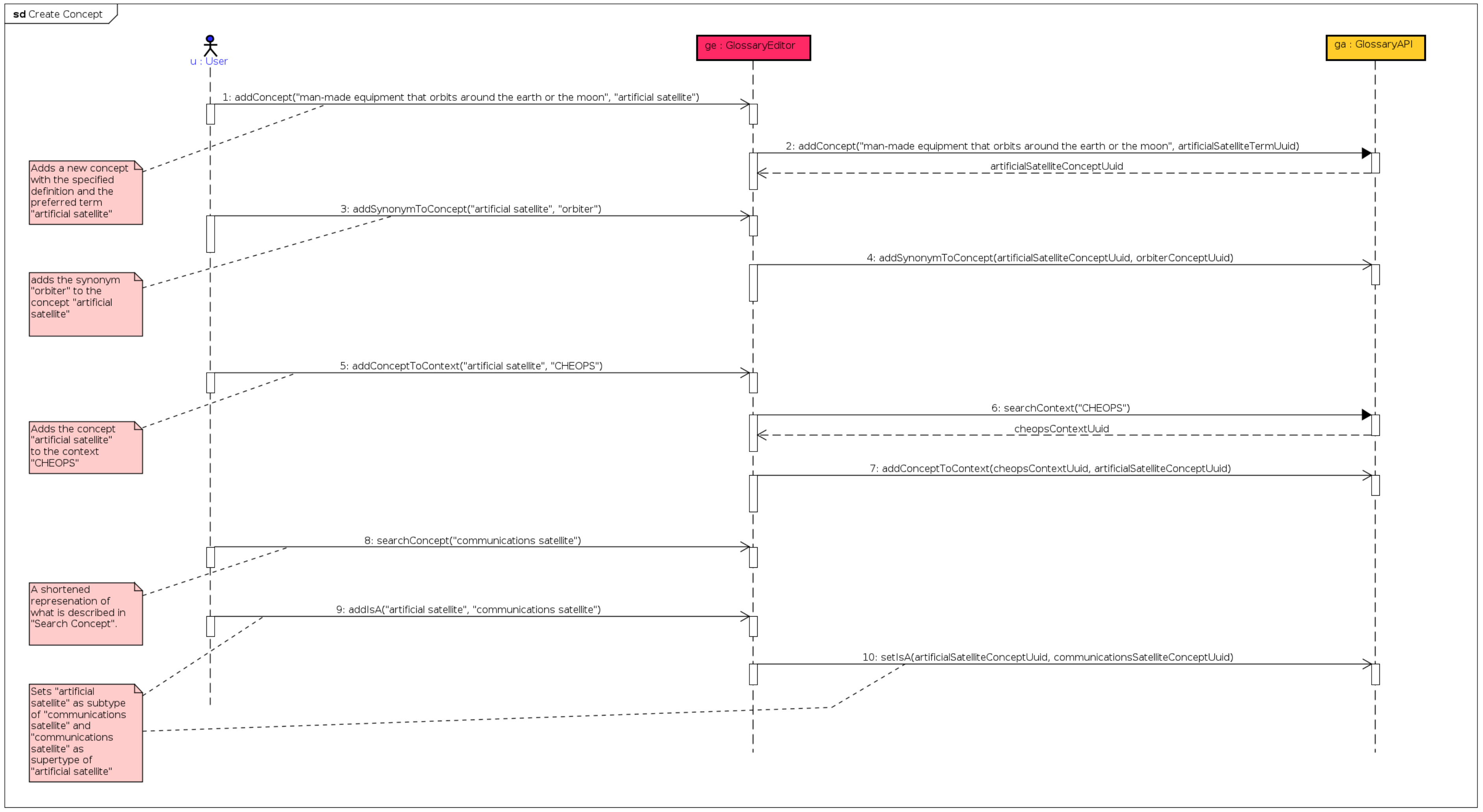

sd Create Concept
u : User
ge : GlossaryEditor
ga : GlossaryAPI
1: addConcept("man-made equipment that orbits around the earth or the moon", "artificial satellite")
2: addConcept("man-made equipment that orbits around the earth or the moon", artificialSatelliteTermUuid)
artificialSatelliteConceptUuid
3: addSynonymToConcept("artificial satellite", "orbiter")
4: addSynonymToConcept(artificialSatelliteConceptUuid, orbiterConceptUuid)
5: addConceptToContext("artificial satellite", "CHEOPS")
6: searchContext("CHEOPS")
cheopsContextUuid
7: addConceptToContext(cheopsContextUuid, artificialSatelliteConceptUuid)
8: searchConcept("communications satellite")
9: addIsA("artificial satellite", "communications satellite")
10: setIsA(artificialSatelliteConceptUuid, communicationsSatelliteConceptUuid)
Adds a new concept with the specified definition and the preferred term "artificial satellite"
adds the synonym "orbiter" to the concept "artificial satellite"
Adds the concept "artificial satellite" to the context "CHEOPS"
A shortened represenation of what is described in "Search Concept".
Sets "artificial satellite" as subtype of "communications satellite" and "communications satellite" as supertype of "artificial satellite"

Figure 4.4: An exemplary creation of a new concept *artificial satellite*.

To *edit* a concept, Search Concept and View Concept Details must be executed first. Then, every detail of the returned concept can be edited, including the concept's definition, preferred term, synonyms, status, language, source, example sentences, contexts and relations to other concepts (that are listed in View Concept Details).

To *delete* a concept, the concept must first be searched in the glossary. When deleting the concept, the GlossarySystem also deletes all context assignments of this concept and all relations of this concept to other concepts.

Since contexts are special concepts, it is possible to convert a concept into a context if this concept should additionally be used to group concepts and/or other contexts.

In addition, it is possible to add a direct linking from a term in a textual artifact and the corresponding concept stored in the GlossarySystem. For example, the links in this section to the textual glossary (only visible in the PDF version of this document) could lead to the associated concept in the glossary, where the preferred term or one of the synonyms of this concept being the link text in this document.

Manage Context

As already mentioned, contexts serve as containers for concepts and other contexts. *Manage Context* is very similar to *Manage Concept* since contexts are special concepts. *Manage Context* includes creating, editing, and deleting a context.

There are only few differences between creating, editing and deleting contexts and concepts: When *creating* a context, concepts and other context can be added to this context, additionally. The same is true for *editing* a context. Additionally, when editing a context, the associations to other contexts and concepts that are contained in this context can be deleted. When *deleting* a context, the GlossarySystem deletes all associations to all other concepts and contexts before the context itself is deleted.

A context can have a list with so called *bad terms* that should not be used in this context. Such bad terms can be added when creating or editing a context and the relations to the particular terms are deleted when deleting the context.

Since contexts are special concepts, it is possible to convert a context into a concept if it's meaning should further be used but if it no longer contains any concepts or other contexts.

Import Concept

As soon as a user searches for a concept using a search term, the *WordNet* [32] is also searched. Results from the WordNet are fallback results, which means that the search functions of the GlossarySystem always output content from the WordNet after all matching glossary content whose source is not the WordNet. Search results from the WordNet are especially interesting if there is no concept in the glossary matching the search term.

The GlossarySystem does include search results from the WordNet into the glossary as soon as the matching concept matches passed search terms a predefined number. Previously, matching concepts from the WordNet are only displayed without adding them to the glossary. The complete WordNet is not read in and mapped to the GlossarySchema, as this would be too much effort due to the size of the WordNet.

Searching the WordNet when performing a search can be turned on and off by the user of the Prototypical GlossaryEditor.

Check Glossary Quality

The GlossaryAPI provides functionalities that can be used to check the quality of a glossary. One of those quality checks consists in the check for homonymous terms, which is part of the use case *replace homonym*. Another quality check is the search for isolated contexts (contexts that do not contain any concept) and for isolated terms (terms that are neither used as preferred terms or synonyms of a concept, nor as bad terms of a context). A further quality check could consist in building some metrics considering the last update date of the glossary and/or the `searchFrequency` attribute.

4.2 Implementation of the GlossarySystem

In this section, the structure and functionalities of the implementation of the GlossarySystem are described.

The naming of `Element` and its subclasses used in the GlossarySystem slightly deviates from the names in the GlossarySchema (see figure 3.1) in that `Element` itself and all its subclasses are named with a trailing "Element". This was done because of the layered architecture used in the GlossarySystem (which is explained below) and omitted in the graphical representation of the GlossarySchema for reasons of legibility.

Besides, when it is spoken of "the graph" in this section, the glossary stored in the GlossarySystem in the TG file persisting format is meant.

4.2.1 Description of the architecture of the GlossarySystem

The GlossarySystem was implemented using the Java programming language. The implementation is not a closed program, but supports its functionalities to external programs via the central application programming interface (API) `GlossaryAPI`.

The GlossarySystem is implemented on four different layers: A *database/persistence layer*, a *business logic layer*, and an *application layer*. The fourth layer in this layer architecture, the *presentation layer*, is realized by external systems using the GlossaryAPI, and by the Prototypical GlossaryEditor, which is described later. Figure 4.5 provides an overview over this four layers and its packages and classes in the GlossarySystem source code.

Instead of directly converting the modelled classes of the GlossarySchema to Java code and providing objects of these classes to users of the GlossaryAPI, the following was done:

The classes and associations of the GlossarySchema are used on the database/persisting layer of the GlossarySystem. The GlossarySystem uses the code generator functionality of *JGraLab*,

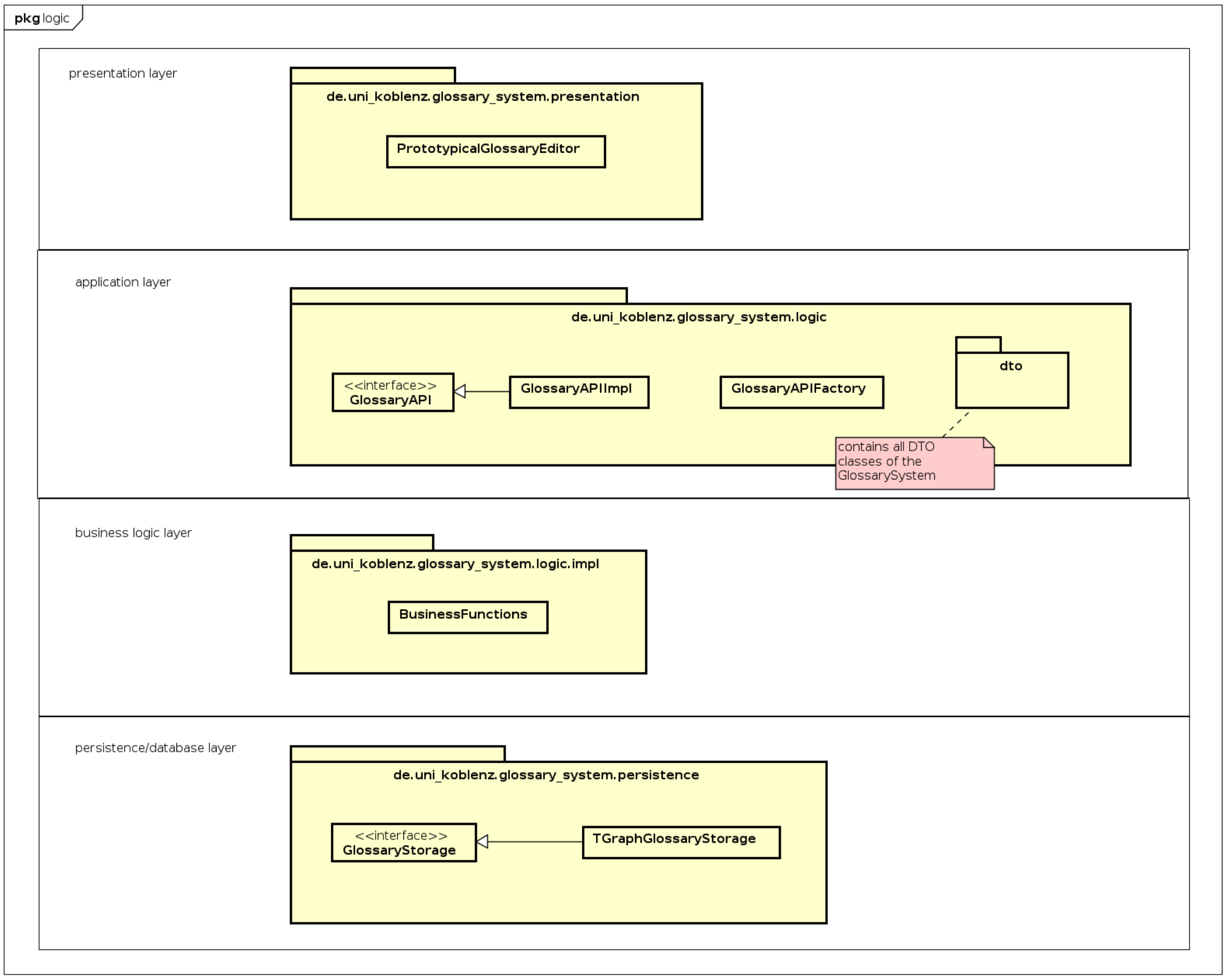

pkg logic
presentation layer
de.uni_koblenz.glossary_system.presentation
PrototypicalGlossaryEditor
application layer
de.uni_koblenz.glossary_system.logic
<<interface>>
GlossaryAPI
GlossaryAPIImpl
GlossaryAPIFactory
dto
contains all DTO classes of the GlossarySystem
business logic layer
de.uni_koblenz.glossary_system.logic.impl
BusinessFunctions
persistence/database layer
de.uni_koblenz.glossary_system.persistence
<<interface>>
GlossaryStorage
TGraphGlossaryStorage

Figure 4.5: An overview over the different layers of the GlossarySystem and its packages and classes.

the *Java Graph Laboratory*, to create all the classes and associations that are modelled in the GlossarySchema. For that, JGraLab uses the hand-written file `GlossarySchema.tg` which is the translation of the UML representation of the GlossarySchema (see figure 3.1) to the TGraph syntax representation of the GlossarySchema. This allows an explicit representation and type-safety usage of all Elements and relations of the GlossarySchema in the GlossarySystem. The GlossarySchema classes are generated to the package `de.uni_koblenz.glossary_system .persistence.tgraph.schema` through executing the file `build.xml` which uses the build management tool *Apache Ant* [8] for the build process. Both `GlossarySchema.tg` and the `build.xml` can be found in the root directory of the GlossarySystem's project directory.

The class `GlossaryAPIImpl` is on the application layer and implements the `GlossaryAPI`. Its constructor is inaccessible to the outside. An object to access the methods of the `GlossaryAPI` can only be created via the method `createGlossaryAPI` of the class `GlossaryAPIFactory`. This method takes two parameters: The filename from where the persisting format file (TG file) to operate on should either be loaded or saved (depending on the next parameter), and the information whether either an existing glossary should be loaded (value `false` for the parameter `newGlossary`) or a new glossary should be created (value `true` for the parameter `newGlossary`). `createGlossaryAPI` calls both the `GlossaryAPIImpl` and the `TGraphGlossaryStorage` constructor. Both classes `GlossaryAPI` and `GlossaryAPIFactory` can be found in the package `de.uni_koblenz.glossary_system.logic` in the `src` directory of the GlossarySystem's project directory.

The main task of `GlossaryAPIImpl` is to check input parameters of the methods provided by the `GlossaryAPI` and to convert Entity objects (of the classes generated by JGraLab on the basis of the GlossarySchema) to data transfer objects (DTOs - objects that are returned by the GlossaryAPI) and vice versa. The complex search methods of the GlossaryAPI are implemented in the class `BusinessFunctions`, which realizes (a big part of) the business logic layer (the calculation of the *nesting distance* (which is explained later in this section) and of homonymous and synonymous terms is realized in `TGraphGlossaryStorage` because in this class, these things can be calculated closer to the graph and therefore more efficiently). `GlossaryAPIImpl` accesses the methods of `BusinessFunctions`. Methods that calculate some complex stuff that are added in future work to the GlossarySystem should also be implemented in `BusinessFunctions`.

With `GlossaryAPI`, the *facade pattern* [10] is used in the GlossarySystem, hiding the internal complexity and functionality of the GlossarySystem. The aim of this is that the methods provided by the `GlossaryAPI` are as useful as possible for the intended application context and do not depend on the internal persistence structure. For example, it is possible to abstract a synonym merely as a list entry of a `Term` data transfer object (DTO) in a `ConceptDetails` DTO instead of, as in the GlossarySchema, as a relation class between `ConceptEntity` and `TermEntity`. In addition, the GlossarySchema and the `GlossaryAPI` can be changed independently of each other in this way, which considerably increases the flexibility and maintainability of the GlossarySystem.

Several facades could access the class `BusinessFunctions`. Because different facades may use different `GlossaryStorage`s, a `GlossaryStorage` object is given to each method of `BusinessFunctions` instead of `BusinessFunctions` storing a `GlossaryStorage` object centrally (which would then have to be from a specific `GlossaryStorage`). Both `BusinessFunctions` and `GlossaryAPIImpl` access `GlossaryStorage`.

Entity objects returned by the code generated by JGraLab on the basis of the GlossarySchema are quite powerful: As soon as a method of an Entity is called, the content of the corresponding element in the persistence format is also changed. In contrast to that, the GlossaryAPI provides small functions and returns so-called *data transfer objects* (DTOs) that can be used and manipulated by external systems without immediately changing the graph. These DTO classes are tailored to the specific methods of the GlossaryAPI. All DTO classes of the current implementation of the GlossaryAPI are shown in figure 4.6. A change to a DTO only affects the persistence format if the DTO is passed as a parameter to an edit method of the GlossaryAPI.

The code generated by JGraLab on the basis of the GlossarySchema also contains helpful methods to query and manipulate the generated graph that are internally used in the GlossarySystem. The GlossarySystem performs necessary checks of input parameters of the methods of the GlossaryAPI and enables an increase in efficiency due to the intermediate storage of calculated search results.

The only persistence format that the system currently uses is *TGraph files*. These are accessed and manipulated via a further internal interface named `GlossaryStorage` (which is on the database/persistence layer and also implements the abovely mentioned facade pattern [10]). The methods described in `GlossaryStorage` are implemented by the data access object (DAO) class `TGraphGlossaryStorage`. This class operates on the TGraph files using the functionalities of the JGraLab library. It is currently the only DAO class of the GlossarySystem.

The reason why the internal persistence/database layer of the GlossarySystem was also realized with the help of the facade pattern is to make it possible to access further persisting formats in future work, e.g. a SQL database. If this were done, another DAO class would have to be created, which converts database queries (e.g. SQL queries) into Entity objects and vice versa.

In addition, an additional `GlossaryAPIFactory` class would have to be added for each additional persistence format since the factory must call the constructor of a specific DAO class.

4.2.2 Description of the methods provided by the GlossaryAPI

In the following, the single methods of the GlossaryAPI are described in detail.

Many methods of the GlossaryAPI return lists of DTOs. In the prototypical implementation, the number of DTOs in these lists is limited to a fixed number, realized by the `MAX_INT` constant of the `GlossaryAPI` interface: For example, a frequently occurring search term passed to one of the search functions of the GlossaryAPI or calling the method `getAllConcepts()`, applied to a large glossary could return a very large number of elements. For example, it is not reasonable in terms of resources to output 100.000 matching concepts in a single list. In future work, a bit by bit output of elements for the methods of the GlossaryAPI returning lists should be realized.

Search methods

The GlossaryAPI provides four ***search methods***, two for concepts and two for contexts. With the aid of these search methods, it is possible to search for concepts respectively contexts passing a *searchTerm*, the information which *elements of the concepts* of the glossary should be searched for this search term, the *current context* and the *part of speech* that should apply to the part of speeches of the preferred terms and synonyms of the returned concepts. Possible

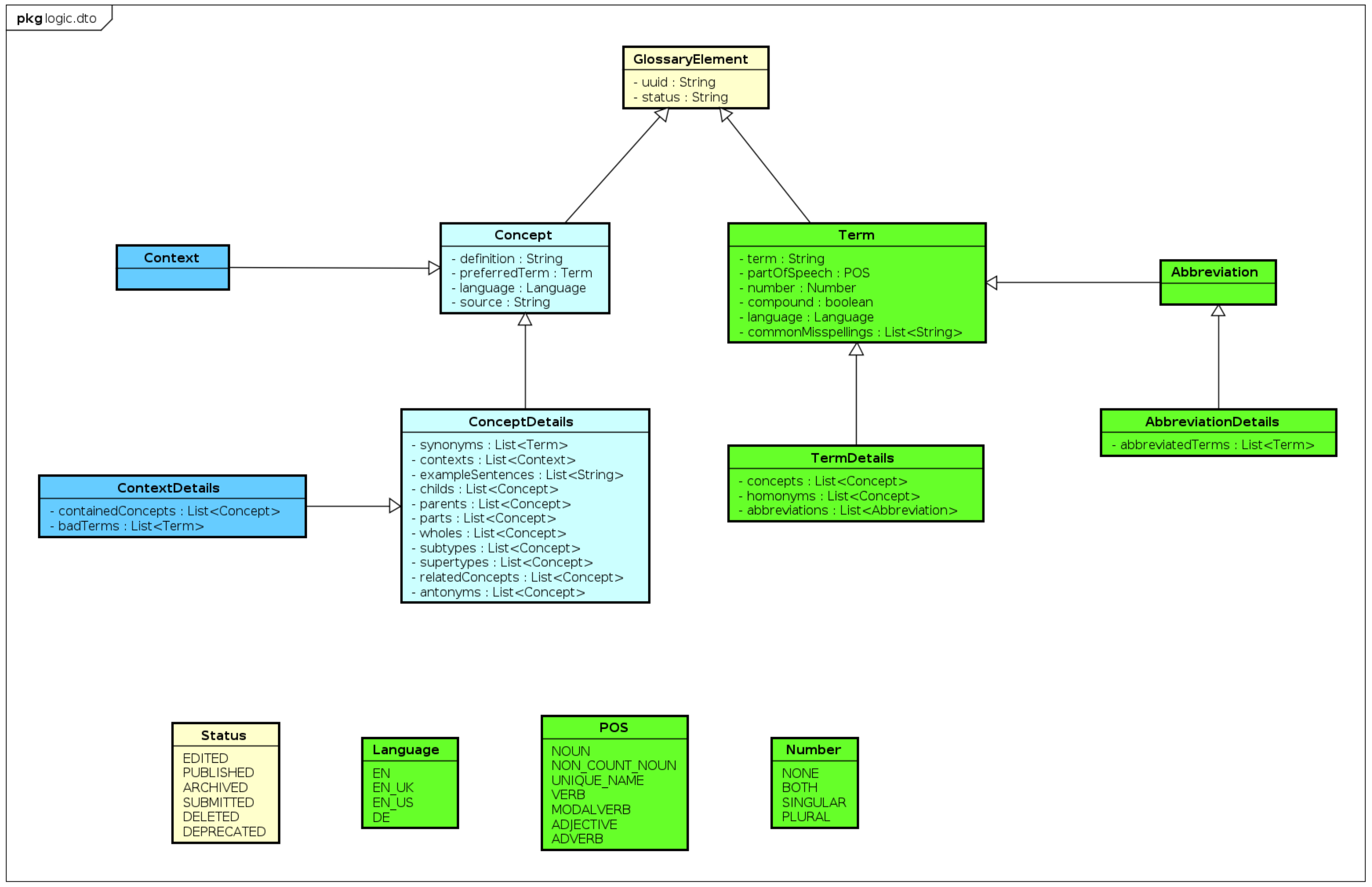


Figure 4.6: Classes for creating data transfer objects (DTOs) for the GlossaryAPI.

elements that can be searched for the search term are: the concept's *preferred term* and *abbreviations* of this preferred term, its *definition*, it's *synonyms* and *abbreviations* of these synonyms, its *example sentences* and its *source*.

The list of the concepts respectively contexts returned by these search methods is sorted in ascending order according to the *nesting distance*[2] between the past current context and the concepts of the list. This nesting distance between a current context Ct and a concept C is calculated as follows:

- If C itself is the current context, the nesting distance is 0.
- If C is more *specific* than Ct (meaning that C is contained in Ct or in one of the direct or transitive subcontexts of Ct), the nesting distance is a *negative* number corresponding to the number of edges (single `ContainsConcept` relations) between Ct and C.
- If C is more *common* than Ct (meaning that C itself is a context and that Ct is contained directly or transitively in C), the nesting distance is a *positive* number corresponding to the number of edges (single `ContainsConcept` relations) between Ct and C.
- The nesting distance is infinity[3] if C and Ct are neither directly nor transitively connected through a `ContainsConcept` relation.

The nesting distance of a concept to the current context represents the potential *relevance* of this concept to a user in this particular context. Thus, the GlossaryAPI returns concepts with a smaller nesting distance to the current context first, since they are most likely more relevant for the searcher. *Being in a context* or *being in the current context* means that the more direct a concept is contained in this context, the more likely it is to be returned. This is especially relevant since the output of the number of elements of functions in the prototypical implementation of the GlossarySystem is not done bit by bit but is limited by a fixed value.

If the nesting distance is the same for two or more concepts, these concepts are sorted according to the point of occurrence of the search term in one of the abovely described elements of the concepts. The priority is as follows: *preferred term* or an abbreviation of a preferred term, *definition*, *synonym* or an abbreviation of a synonym, *example sentences*, *source*.

If this point of accourrence is also the same for two or more concepts, these concepts are sorted alphanumerically according to their preferred terms.

Sorting according to these criteria is very important to get possibly relevant concepts output first. The sorted output becomes even more important due to the fact that all functions of the GlossaryAPI only return a fixed number of concepts in the prototypical implementation. Without proper sorting, highly relevant concepts might be cut off at the bottom of the list, while less relevant concepts might be at the top.

Get DTO methods

For each data transfer object class shown in figure 4.6, the GlossaryAPI provides a method that returns a DTO of the type of the respective DTO class using the unique identifier of the corresponing `ElementEntity` in the glossary. The returned DTOs are assigned the corresponding values stored in the graph.

[2]The nesting distance is calculated internally in `TGraphGlossaryStorage` using the breadth-first search (BFS) algorithm.

[3]In the implementation, the nesting distance is set to 1.000.000 in this case.

Get all methods

One method each is provided to return all *concepts*, *contexts*, *terms*, *abbreviations*, *homonyms* (terms contained in the glossary that are associated with more than one concept) and *synonyms* (concepts contained in the glossary that are associated with more than one term) contained in the glossary. The returned DTOs are assigned the corresponding values stored in the graph.

Is methods

The GlossaryAPI provides four methods which check whether the transferred unique identifier belongs to a concept, context, term or abbreviation in the glossary.

Get methods for contains relations between contexts and concepts

The GlossaryAPI provides one method that returns all concepts that are directly contained in the passed context and another method that returns all contexts in which the passed concept is directly contained.

Get methods for all hierarchical relations between concepts

The GlossaryAPI provides two methods for each hierarchical relationship type, which return the child or parent concept to the passed concept.

Get methods for all associative relations between concepts

The GlossaryAPI provides one method that returns all related concept to the passed concept and another method that returns all antonyms to the passed concept.

Add glossary element methods

The GlossaryAPI provides several methods that allow to add a new concept or context or term or abbreviation to the glossary. The different methods take different numbers of parameters so that it is not necessary to pass `null` as parameters several times when, for example, a concept with a minimal amount of initial values should be created.

Add relation methods

The API provides one method each, which adds:

- a concept to a context
- a synonym to a concept
- a preferred term to a concept

- a hierarchical relation between two concepts, where the relation type is specified by a parameter
- an associative relation between two concepts, where the relation type is specified by a parameter
- an abbreviation to a term

In future work, it is planned to add another method that adds a term as bad term to a context.

Edit methods

The GlossaryAPI provides four edit methods, one for concepts, one for contexts, one for terms and one for abbreviations. Each of this methods takes the unique identifier of a concept or context or term or abbreviation in the glossary and a `Concept` or `Context` or `Term` or `Abbreviation` DTO. When calling the particular method with these two values, the values of the concept or context or term or abbreviation in the glossary corresponding to the passed unique identifier are overwritten with the values of the passed DTO.

Delete/remove methods

The GlossaryAPI provides one method to remove a `GlossaryElement` corresponding to the passed unique identifier from the glossary, and one method each, which removes:

- a concept from a context
- a synonym from a concept
- the preferred term of a concept
- a hierarchical relation between two concepts, where the relation type is specified by a parameter
- an associative relation between two concepts, where the relation type is specified by a parameter
- an abbreviation from a term

In future work, it is planned to add another method that removes a term as bad term from a context.

The method `saveGlossary()`

The GlossaryAPI provides a method `saveGlossary()` that saves all changes (additions, edits and deletions) to the TG persisting file that was specified when creating the method `GlossaryAPIFactory.createGlossaryAPI` (see section 4.2.1).

4.3 The Prototypical GlossarySearcher

Unfortunately, for time reasons, it was not possible to implement a Prototypical GlossaryEditor with all functionalities described in the use cases and the requirements within the scope of this work. Instead, the class `PrototypicalGlossarySearcher.java` was developed, which can be found in the package `de.uni_koblenz.glossary_system.presentation`.

The Prototypical GlossarySearcher allows to load existing glossaries (TG files), to set the current context, to search the loaded glossary for concepts and to show details of the loaded concepts. The set current context is kept as current context until it is set again, which implements the use case Set Current Context. The Prototypical GlossarySearcher allows to search all appropriate components of a concept for the search term (preferred term or an abbreviation of a preferred term, definition, synonyms or an abbreviation of a synonym, example sentences, source) or only the preferred term and the synonyms or only the definition. Furthermore, it is possible to define that only such concepts can be returned whose preferred terms or synonyms are of a certain part of speech. Figure 4.7 shows a screenshot of the prototypical GlossarySearcher in action.

4.4 Future work for the GlossarySystem

This chapter lists functionalities which should be realized in future work on the GlossarySystem.

4.4.1 The integration of the WordNet

Unfortunately, for time reasons, it was not possible to integrate the WordNet into the GlossarySystem as described in the use cases and the requirements within the scope of this work. This task remains as one of the most important tasks for future work.

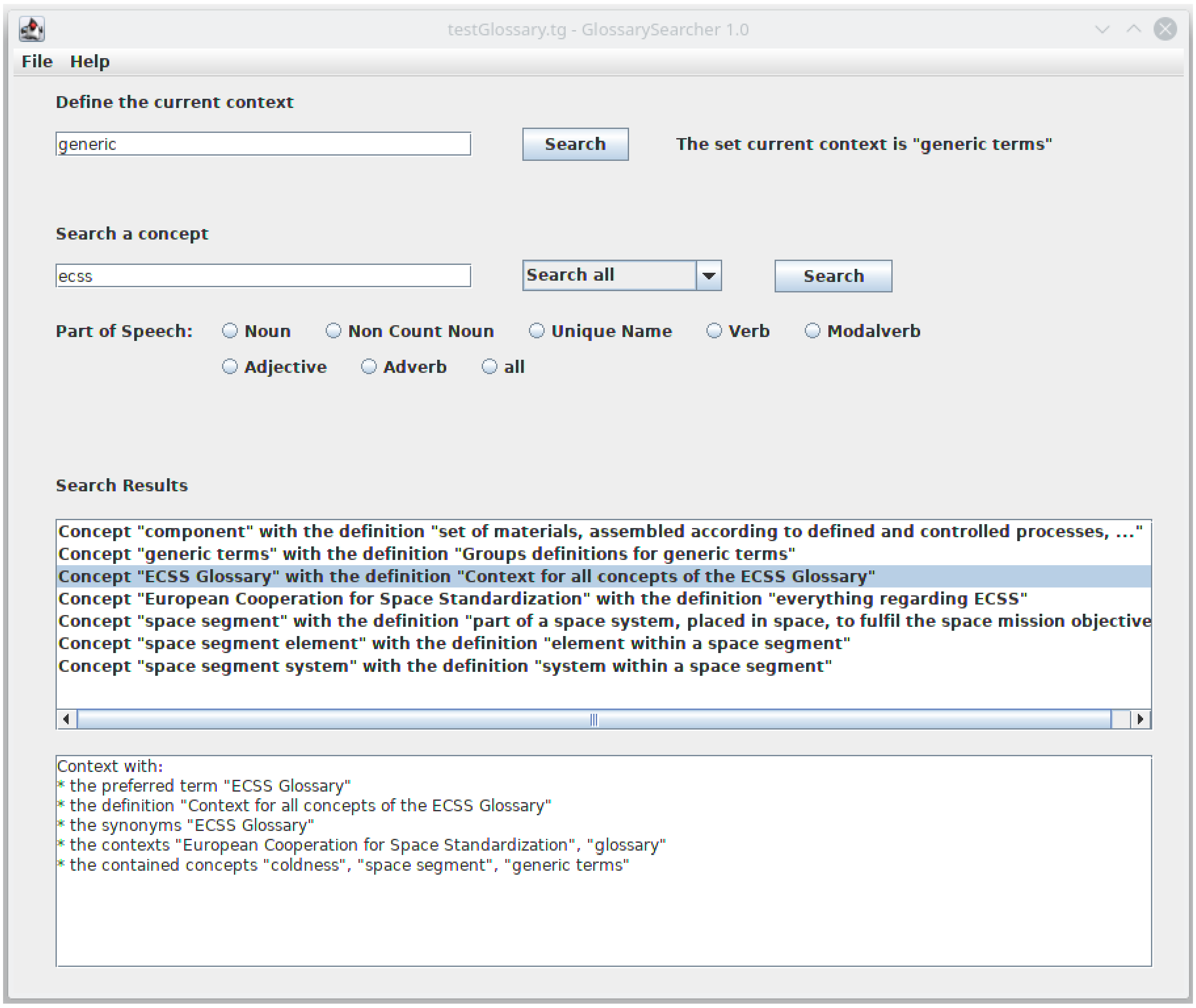
The WordNet should be integrated into the GlossarySystem as follows: As API to access the WordNet, JGWNL (the JGraLab WordNet Library) 2.5 should be used. The GlossarySystem should connect to JGWNL as a client.

As soon as a user searches for a concept using a search term, the WordNet should also be searched. Results from the WordNet are fallback results. The search functions of the GlossarySystem always output content from the WordNet after all matching glossary content whose source is not the WordNet.

4.4.2 Future work regarding the GlossaryAPI

One of the most important remaining tasks that should be done in future work is the implementation of the *not yet implemented constraints* of the GlossarySchema. 9 constraints annotated in the GlossarySchema were implemented in the GlossarySystem, 3 constraints are not automatically checkable (what is noted in the GlossarySchema for the concerned constraints), and the implementation of the following 3 constraints remains future work:

- „Every combination of term and pos may occur only one time.“



generic

The set current context is "generic terms"

ecss

```
Concept "component" with the definition "set of materials, assembled according to defined and controlled processes, ..."
Concept "generic terms" with the definition "Groups definitions for generic terms"
Concept "ECSS Glossary" with the definition "Context for all concepts of the ECSS Glossary"
Concept "European Cooperation for Space Standardization" with the definition "everything regarding ECSS"
Concept "space segment" with the definition "part of a space system, placed in space, to fulfil the space mission objective
Concept "space segment element" with the definition "element within a space segment"
Concept "space segment system" with the definition "system within a space segment"
```

```
Context with:
* the preferred term "ECSS Glossary"
* the definition "Context for all concepts of the ECSS Glossary"
* the synonyms "ECSS Glossary"
* the contexts "European Cooperation for Space Standardization", "glossary"
* the contained concepts "coldness", "space segment", "generic terms"
```

Figure 4.7: Example of the usage of the Prototypical GlossarySearcher using the search term "ecss" and the current context "generic terms". (In the figure, the TestCaseGlossary (see figure D.1) is loaded.)

- „All synonyms must have the same POS.“
- „If abbreviatedTerms has a value, the values of the attributes pos, number and compound of the Abbreviation must be equal to the values of the identically named attributes of all of these abbreviatedTerms. If the abbreviatedTerms of an Abbreviation differentiate in one or more of their attributes pos, number, compound or language, the following applies: For every value combination of these attributes, a new Abbreviation object must be created to fulfill the other Abbreviation constraint.“

Furthermore, regarding constraints, in the prototypical implementation of the GlossarySystem, when inserting concepts or contexts into other contexts, it is only checked for direct containment whether the inserted concept or context is inserted into itself. In future work, this should also be checked for *transitive containment* (e.g. if a context $C1$ is inserted into a context $C2$ that is again contained in $C1$, or more complex nestings).

Another quality check that should be implemented in future work is that every concept of a glossary is contained in at least one context.

According to section 3.4, it is not possible to delete a term, concept or context once it has been inserted into a glossary derived from the GlossarySchema. However, in the first version of the implementation of the GlossarySystem, it seems to be useful to also allow the deletion of terms, concepts and contexts since not allowing to delete a once inserted GlossaryElement seems to be quite limiting. The strict requirement of only marking terms, concepts and contexts as deleted could be implemented in further versions. In this context, it might be useful to apply this strict rule only to GlossaryElements that have or once had the status `PUBLISHED`.

In this context, could also be future work to restrict the possible changes of status states to figure 3.9.

Another point that should be part of future work and could become important is the *simultaneous usability* of the GlossarySystem: In the future, the GlossarySystem should be able to handle multiple users (who may use different external systems) writing to the same glossary at the same time. This problem was not considered in the prototypical implementation of the GlossarySystem.

The methods *addBadTermToContext* and *removeBadTermFromContext* that are already specified (commented out) in the GlossaryAPI source code should be implemented.

Two more methods that return the contains relations between contexts and concepts could be added: One method that returns all concepts that are *transitively* contained (via intermediate contexts) in the passed context and another method that returns all contexts in which the passed concept is *transitively* contained. The returned concepts or contexts should be sorted according to the nesting distance to the passed concept or context, as described in section Search Methods.

In the current implementation of the GlossarySystem, the search methods supports exact matches of passed search terms (through passing `searchTerm`, partly matches through passing `*searchTerm` or `searchTerm*` and contains checks through passing `*searchTerm*`. In addition, case sensitiviy and leading and trailing whitespaces are ignored. In future work, *further stemming* of passed search terms should be supported: diacritical characters, hyphens and inflections of passed searchTerms should also be ignored.

Regarding the search methods of the GlossaryAPI, the *search by common misspellings* should also be implemented in future work (compare section 3.5). Common misspellings should not be output, but only serve to ensure that the correct concept is also output when a user passes a common misspelling (see also section 3.5).

For the GlossaryAPI, a method *isTopConcept* could be added that returns *true* if the associated concept has at least one child but no parent and *false* otherwise.

To fulfill Z39.19-2005 [2], section 9.5, the GlossarySchema could provide methods that return the *number of concepts, contexts and preferred terms* of the glossary and *the date and time of the last update* or the glossary. The number of concepts and contexts in a glossary can already be determined by calling `getAllConcepts().size()` and `getAllContexts.size()`. Explicit functions for determining the respective number could be made more efficient internally, for example by using GReQL requests in class `TGraphGlossaryStorage`. However, the last update is a difficult metric: If it is a long time ago, this can mean either that the glossary is not sufficiently maintained, but also that it is already quite good and therefore no maintenance was necessary.

In addition, a method `getAllProjectSpecificHomonyms(String projectUuid)` that returns all homonyms that belong to the project with the past projectUuid could be added.

Another useful feature that could be supported by the GlossarySystem in future work is the *conversion of concepts into contexts* as described in the use case Manage Concept and the *conversion of contexts into concepts* as described in the use case Manage Context.

Another feature that the GlossarySystem could support is *multilingualism* for the reasons explained in section 3.7. Although all enum constants of the enum `Language` of the GlossarySchema (see figure 3.1) can be used when defining new `GlossaryElements`, the system is only designed for the English language. If multilingualism is supported, the GlossarySchema must also be adapted, as explained in section 3.7. In addition, the ISO standard ISO 25964-2 [13] already mentioned at the beginning of chapter 3 should be consulted.

As already mentioned in chapter 3 in the description about the `definition` attribute, the processing of html markup for definitions and example sentences of concepts should be supported in future work. For the prototypical implementation of the GlossarySystem in the scope of this work, it is assumed that the values of these attributes are plaintext only.

Finally, as already mentioned in the introduction of section 4.2.2, a bit by bit output of elements for the methods of the GlossaryAPI returning lists should be realized for the reasons stated in the introduction of section 4.2.2.

4.4.3 Future work regarding other system components

The GlossaryAPI already generates meaningful exceptions if parameters are passed to its methods that make no sense or that violate one of the constraints of the GlossarySchema. Besides this, further specific exceptions should be defined that are caused by internal errors caused by wrong input parameters. For example:

- If `TGraphGlossaryStorage.loadGlossary()` throws a `GraphIOException` which is caused by a `FileNotFoundException`, the GlossaryAPI should extract the `FileNotFoundException` and pass it to external systems instead of passing the `GraphIOException` to the outside.
- If `GlossaryAPIFactory.createGlossary` is passed a non-existent file name, `TGraphGlossaryStorage` throws a `NullPointerException`. Passing something like a GraphFileNotFoundException to the outside would be nicer.

Future work could also consist of using GReQL in the methods of TGraphGlossaryStorage. This could help to avoid having to use the functions generated by JGraLab to calculate a large number of elements from the graph in order to filter only a few elements.

4.4.4 Future work regarding the Prototypical GlossarySearcher

The Prototypical GlossarySearcher is actually not part of the GlossarySystem but an external system, but due to parallel development it is nevertheless listed as a subsection of this section.

In future work, the Prototypical GlossarySearcher should be developed into the planned Prototypical GlossaryEditor and expanded by functionalities that allow to define, edit and delete `GlossaryElement`s and to create and save new glossaries.

A further feature that could be integrated is the possibility to replace homonyms, as stated in the use case Replace Homonym.

Furthermore, the Prototypical GlossaryEditor should be supplemented by a function that prints the whole glossary to a text and/or PDF file. This output could contain all Concepts of the glossary alphanumerically sorted according to their preferred terms and further information like synonyms, Contexts in which the particular Concept is contained and for Concepts that are Contexts Concepts which are contained in the particular Context.

Another task that could be solved in future work is the direct linking from terms occuring in textual artifacts to the associated concept in a glossary of the GlossarySystem, mentioned in the use case Manage Concept. This feature would need some more thought, but would most likely be part of the presentation layer, so it could be part of the Prototypical GlossaryEditor.

4.5 Recommendations regarding the usage of the GlossarySystem

This section provides some recommendations regarding the usage of the GlossarySystem, mainly derived from the requirements and recommendations from Z39.19:2005 [2].

Both van Lamsweerde [14] („to ensure that the same term does not refer to different concepts and the same concept is not referred to under different terms“) and Z39.19:2005 [2] („a compound term ***should*** express a single concept or unit of thought“; „As a general principle of vocabulary control, terms ***should*** represent single concepts [. . .]“) state that terms should refer to a single concept and concepts should be represented by a single term and represent this rules as very important, central concepts. Due to the strict separation of `Terms` and `Concepts` in the GlossarySchema, it is not necessary to stick to this rules when using the GlossarySystem.

However, this decision is primarily the responsibility of Editors of the GlossarySystem. But it isn't a bad idea to do so as the usability of the glossary of terms otherwise depends strongly on the GlossarySystem.

Due to the strong separation of `Terms` and `Concepts` and the possibilities of the `Context` system in the GlossarySchema, many recommendations from Z39.19:2005 [2], chapter 6 (*Term Choice, Scope, and Form*) do not necessary need to be applied as these recommendations assume that terms and concepts are more interrelated than they actually are in the GlossarySchema.

According to Z39.19:2005 [2], section 6.4 recommends that the grammatical form of a term should be a noun or noun phrase, while verbs should not be used alone as terms. Furthermore, Z39.19:2005 [2] recommends that adjectives and adverbs should be used only under special circumstances (e.g. for adjectives and adverbs with special meanings, such as *legato* and *staccato* in music). This recommendations also seem to make sense for user of the GlossarySystem who add project-specific terms to the glossary: Activities (e.g. *to distill* or *to freeze*) that occur in a textual artifact (e.g. a requirement) and whose meaning needs to be defined in the glossary can be represented by nouns or gerunds (e.g. *distillation* or *freezing*).

Z39.19:2005 [2] enumerates multiple criteria for establishing (Z39.19:2005 [2], chapter 7.5) and splitting (Z39.19:2005 [2], chapter 7.6) compound terms. Like many other recommendations of the standard, their compliance is the responsibility of users of the GlossarySystem.

According to Z39.19:2005 [2], „controlled vocabulary maintenance personnel" („users [that] [...] are likely to be experts in the subject domain of the controlled vocabulary") „***must*** have access to all views of a controlled vocabulary and complete information about each term, with the ability to edit and manipulate term records, cross-references, classification notation, and hierarchies". With regard to the corresponding elements and relations of the GlossarySchema, the GlossaryAPI fulfills this requirement by means of its corresponding functionalities.

Z39.19:2005 [2] recommends that *preferred terms* should be assigned consistently according to defined rules. For example, glossary-wide either scientific or other names should be selected as `preferredTerms`.

4.6 Further remarks on the GlossarySystem

In the implementation of the GlossarySystem, only transitive contains relations are supported, which means that every concept C that is contained in a context $Ct1$ is semantically also contained in all contexts $Ct2$ in which $Ct1$ is contained (role name `contexts` of the `ContainsConcept` of the GlossarySchema) and in all contexts $Ct3$ that $Ct1$ contains (role name `concepts` of the `ContainsConcept` of the GlossarySchema). No non-constructed examples for the existence of intransitive contains relationships could be found. However, this possibility should be available for future implementations.

One of the big advantages of the GlossarySystem is that, due to the layer architecture of the GlossarySystem, the GlossarSchema can contain details that are not directly implemented in the GlossaryAPI.

For the problematic reasons described in chapter 3, the GlossarySystem does not support storing a central search frequency, as modelled in the GlossarySystem. Instead, it is recommended that developers of external systems using the GlossaryAPI store the search frequency locally to use it for the puropses named in chapter 3. The local search frequency might even be more interesting, since search terms used individually by a single user could be used. In this way, the value is not distorted the other search habits of other users.

Z39.19:2005 [2] requires that „[u]nique entities [...] are usually expressed as proper nouns“ and „[a]postrophes that are part of proper names ***must*** be retained“. The GlossarySystem ensures that unique names are stored exactly as they are entered by the user.

Chapter 5

Conclusion

With the developed GlossarySystem and the underlying GlossarySchema, the objective will be reached to develop glossaries of higher quality that are easier to use and maintain across projects. The GlossarySchema allows the formal representation of the properties and relations, the developed GlossarySystem allows an easy, practical and efficient handling of the glossaries, and the planned and partially implemented Prototypical GlossaryEditor allows a good representability and usability of the glossaries.

The relations modeled in the GlossarySchema and implemented in the GlossarySystem are no longer, as in conventional term definition list glossaries, rather hindering the search for and understanding of correlations by enlarging the textual definition. Instead, they serve the understanding of the correlations of the respective concept due to the formal representation outside the term definition.

Through the development of the GlossarySchema and the GlossarySystem, the research questions RQ1 - RQ3 listed in section 1.3 could be answered:

The developed GlossarySchema answeres RQ1 (*What should a glossary be like that is useful for requirements engineering?*) through explicitely representing homonymy and synonymy, providing numerous useful properties of terms and concepts and the provided relations between concepts. Furthermore, is answeres RQ2 (*How can contexts be represented in such a glossary?*) through the implemented realization of contexts and the possibility to arbitrarily nest concepts and contexts in other contexts.

Finally, the development of the GlossarySystem answers RQ3 (*How can such a glossary including contexts be technically implemented?*).

Finally, future work should consist of *applying* the GlossarySystem based on the current GlossarySchema and, if necessary, modifying the functionalities of the GlossarySystem and/or the underlying GlossarySchema. New methods could be added to the GlossaryAPI or existing methods that prove to be less useful could be removed. The same applies to properties of elements and relations between elements of the GlossarySchema.

Appendix A

Textual Glossary

antonym - the role of a ↑concept (first definition) that has an ↑antonymic relationship to another ↑concept (first definition). Is realized in the ↑GlossarySchema as role name `antonyms` of the association class `HasAntonym`.

antonymy relation - a special ↑associative relation representing that one ↑concept (first definition) is an ↑*antonym* (an opposite) of another one, e.g. the corresponding ↑concepts (first definition) to the ↑terms (second definition) *hot* and *cold*. Is realized in the ↑GlossarySchema as association class `HasAntonym`.

any character wildcard - The ↑wildcard that is used for "any number of any characters". The functions of the ↑GlossarySystem that support wildcards use the character `*` for this purpose.

associated , *associate* - If a ↑concept (first definition) or a ↑context A is *contained* in another ↑context B, B is *associated* with A and a is *associated* with B.

associative relation - a relation between ↑concepts (first definition) that defines some kind of non-↑hierarchical relations between two or more ↑concepts (first definition). Is realized in the ↑GlossarySchema as association class `HasRelatedConcept`.

bad term - a ↑term (second definition) that should not be used in a specific ↑context.

child - the role of a ↑concept (first definition) that is subordinate to another ↑concept (first definition) in a ↑hierarchical relation. Is realized in the ↑GlossarySchema as role name `childs` of the association class `HasParent`.

concept

1. The *meaning* of at least one ↑term (second definition), represented by a *textual definition* and the following useful properties: The concept's corresponding ↑*synonyms* (second definition) and it's ↑*preferred term*, illustrating *example sentences*, the *language* of the textual definition and the example sentences, the concept's ↑*hierarchical relations* (↑*part-whole relations*, ↑*common instance relations* and ↑*individual instance relations*) and ↑*associative relations* including ↑*antonymy relations*, the ↑*contexts* in which the concept is contained and an information about the *source* of the Concept. Concepts are the central entities of the ↑GlossarySchema and the ↑GlossarySystem and realized as class `Concept` in the ↑GlossarySchema.
2. An abstract or general idea inferred or derived from specific instances. [33]

common instance relation - a special ↑instance relation between two ↑concepts (first definition) representing a *common* ↑supertype/↑subtype relation, e.g. between the corresponding ↑concepts (first definition) to the terms *succulent plant* and *cacti*. Is realized in the ↑GlossarySchema through setting the attribute `individualInstance` of `IsARelation` to *false*.

common subtype - a possible role of a ↑concept (first definition) in an ↑instance relation.

compound term - a ↑term (second definition) whose ↑term (first definition) is composed of more than one ↑terms (first definition). Consists linguistically of more than one lexemes and/or more than two morphemes. Is realized in the ↑GlossarySchema through the `Term`'s `compound` attribute.

concept relation - a relation between two ↑concepts. Can be either a ↑hierarchical relation, or an ↑associative relation or a ↑contains relation. Is realized in the ↑GlossarySchema as association class `HasConcept`.

contains relation - a possible relation between a ↑concept (first definition) or ↑context A and another ↑context B: A can be either ↑transitively or ↑intransitively contained in B. Is realized in the ↑GlossarySchema as association class `ContainsConcept`.

context - a thematic grouping of ↑concepts (first definition) and/or other contexts. A context can group any number of ↑concepts (first definition) and other contexts. Contexts can be arbitrarily nested using ↑contains relations and have all the properties of ↑concepts (first definition). Additional to the contained ↑concepts (first definition) and contexts, a context can have a list of ↑*bad terms*. Contexts are special ↑concepts (first definition), thus having all properties of ↑concepts (first definition). Concepts are realized in the ↑GlossarySchema as class `Concept`.

See also ↑contains relation

context association - see ↑associated

current context - The ↑context in which a ↑user is currently mentally located if he/she searches for a ↑concept (first definition). If, for example, the ↑user formulates requirements for a specific project or system, the current context could be the project/system or a specific part of the project or system. For example, when writing requirements for the ↑GlossarySystem, either ↑*UserAPI* or ↑*Prototypical GlossaryEditor* could be the current context.

Since the current context is a normal ↑context, all ↑contexts in which the current context is contained are also implicit current contexts. If the current context is transitively contained in another context, this even goes further down the context hierarchy (see ↑context).

editor - an extension of the role ↑user. An editor has all the skills of a user and, in addition, the rights to edit the content of a ↑glossary of the ↑GlossarySystem. See also ↑EditorAPI.

EditorAPI - the part of the ↑GlossaryAPI providing all functionalities which should be exclusively available for the role ↑editor.

external system - a system that uses the ↑GlossaryAPI to provide functionalities based on the ↑GlossarySystem.

glossary , *glossary of terms*

1. A collection of terms and correspondig definitions which represent key ↑concepts (second definition) of a project or system. It should serve to avoid clashes in terminology, designation and structure and provide a list of accepted synonyms. Furthermore, it should „ensure that the same term does not refer to different concepts [(homonymy)] and the same concept is not referred to under different terms [(synonymy)]" [14].

2. A glossary as defined in the first definition created with the ↑GlossarySystem according to the ↑GlossarySchema. When created with the GlossarySystem, this is technically a TG file, but theoretically, a glossary can also be derived as an object diagram directly from the GlossarySchema. Due to its structure, a glossary created and used with the aid of the GlossarySystem makes it even possible to have homonyms and synonyms without that creating the danger of clashes in terminology, designation and structure as mentioned in the first definition of glossary.
   Is realized as class `Glossary` in the ↑GlossarySchema.

GlossaryAPI - the Application Programming Interface (API) of the ↑GlossarySystem providing all functionalities that should be usable by ↑external systems. Consists of the ↑UserAPI and the ↑EditorAPI. The implementation of the GlossaryAPI in the ↑GlossarySystem is not divided into ↑UserAPI and ↑EditorAPI, but implemented as a single interface "GlossaryAPI".

GlossarySchema - a schema which can also be described as *metamodel* that allows to represent glossaries mainly for the requirements engineering and software projects as thesauri. Is modelled as UML class diagram and enables the formal representation of properties of and relations between terms and definitions of a glossary. Is the underlying schema of the ↑GlossarySystem.

GlossarySystem - a program that implements the ↑GlossarySchema and makes its features accessible through providing useful functionalities, including the creation, editing and searching of glossaries based on the GlossarySchema. Provides its functionalities to ↑external systems via the ↑GlossaryAPI.

hierarchical relation - a relation that defines some kind of hierarchy between two or more ↑concepts (first definition). Is divided into ↑part-whole relation and ↑instance relation. Is realized in the ↑GlossarySchema as association class `HasParent`.

individual instance relation - a special ↑instance relation between two ↑concepts (first definition) representing an individual ↑supertype/↑subtype relation, e.g. between the corresponding ↑concepts (first definition) to the terms *artificial satellite* and *Sputnik 1*. Is realized in the ↑GlossarySchema through setting the attribute `individualInstance` of `IsARelation` to *true*.

individual subtype - a possible role of a ↑concept (first definition) in an ↑instance relation.

instance relation - a special ↑hierarchical relation between two ↑concepts (first definition) representing a ↑supertype/↑subtype relation. Is either a ↑common instance relation or an ↑individual instance relation. Is realized in the ↑GlossarySchema as association class `IsARelation`.

intransitively contained - A special ↑contains relation: If a ↑concept (first definition) or ↑context A is *intransitively* contained in B, this means that A is contained exclusively in B. See also ↑transitively contained. In addition, the textual definition of B should be applicable to A and all bad terms of B are also ↑bad terms of A if A is a ↑context.

Is expressed in the ↑GlossarySchema using the enum literal `INTRANSITIVE`. See also ↑transitively contained.

IsA relation - see instance relation

number - the information if a singular and/or a plural is defined for a ↑term (second definition). Is realized in the ↑GlossarySchema attribute `number` of the class `Term`.

one character wildcard - The ↑wildcard that is used for "a single, arbitrary character". The functions of the ↑GlossarySystem that support wildcards use the character `?` for this purpose.

parent - the role of a ↑concept (first definition) that is superordinate to another ↑concept (first definition) in a ↑hierarchical relation. Is realized in the ↑GlossarySchema as role name `parents` of the association class `HasParent`.

part - a possible role of a ↑concept (first definition) in a ↑part-whole relation. Is realized in the ↑GlossarySchema as role name `parts` of the association class `HasWhole`.

part-whole relation - a special ↑hierarchical relation representing that one ↑concept (first definition) is ↑part of another one (the ↑whole), e.g. the corresponding ↑concepts (first definition) to the ↑terms (second definition) *chair* and *leg*. Is realized in the ↑GlossarySchema as association class `HasWhole`.

preferred term - a ↑synonym (second definition) that is preferentially used to name a concept. Is realized in the ↑GlossarySchema as role name `preferredTerm`.

Prototypical GlossaryEditor - a program that uses the ↑GlossaryAPI demonstrating the basic features provided by the GlossaryAPI. The main purpose of the Prototypical GlossaryEditor is to demonstrate and evaluate the functionality and the usage of the ↑GlossarySystem.

Prototypical GlossarySearcher - a program that realizes a subset of the planned features of the ↑Prototypical GlossaryEditor and can be used to load glossaries, set the current context and search the loaded glossaries for concepts and contexts.

related concept - the role of a ↑concept (first definition) that has an ↑associative relation to another ↑concept (first definition). Is realized in the ↑GlossarySchema as role name `relatedConcepts` of the association class `HasRelatedConcept`.

SHALL - a keyword used in the detailled requirements of the ↑GlossarySystem to identify the legal obligation of a requirement. A requirement marked with *shall* is *mandatory* in the realization. See also ↑SHOULD.

SHOULD - a keyword used in the detailled requirements of the ↑GlossarySystem to identify the legal obligation of a requirement. A requirement marked with *shall* is *optional* in the realization. See also ↑SHALL.

simple type - the Java types *int*, *boolean* and *String*.

status - the status of a ↑concept (first definition), ↑context or ↑term (second definition). Possible statuses are *edited*, *published*, *archived*, *submitted*, *deleted* and *deprecated*. Is realized in the ↑GlossarySchema as attribute `status` of the enum type `Status` of the class `Element`.

subtype - a possible generic role of a ↑concept (first definition) in an ↑instance relation. See common subtype and individual subtype. Is realized in the ↑GlossarySchema as role name `subtypes` of the association class `IsARelation`.

supertype - the role of a ↑concept (first definition) that is superordinate to another ↑concept (first definition) in an ↑instance relation. Is realized in the ↑GlossarySchema as role name `supertypes` of the association class `IsARelation`.

synonym

1. A ↑term (first definition) which has the same definition as another ↑term (first definition).
2. A ↑term (second definition) that is used as possible designation of a ↑concept (first definition). Is realized in the ↑GlossarySchema as role name `synonyms`.

term

1. A string representing a word. Consists linguistically of at least one lexeme and/or two or more morphemes. Is realized in the ↑GlossarySchema as `Term`'s attribute `term`. See also ↑compound term.
2. A combination of a string storing a ↑term (first definition) and the further properties *part of speech*, the ↑*number*, the fact if the term is either a ↑*compound term* or not, the *language*, possible *abbreviations*, and *common misspellings*. Functions only as a label for one or more ↑concepts (first definition). Is realized in the ↑GlossarySchema as class `Term`. See also ↑compound term.

top concept - a ↑concept (first definition) that is a supertype but itself has no supertypes (see ↑instance relation). In the ↑GlossarySchema, the `Concept`'s attribute `topConcept` is used to express that a `Concept` is a top concept.

transitively contained - A special ↑contains relation: If a ↑concept (first definition) or ↑context A is *transitively* contained in another ↑context B, this means that A is contained in B itself and in all other ↑contexts C in which B is contained. In addition, the textual definitions of B and C should be applicable to A and all ↑bad terms of B and C are also ↑bad terms of A if A is a ↑context. Is expressed in the ↑GlossarySchema using the enum literal `TRANSITIVE`. See also ↑intransitively contained.

user - a role that represents an average user of the ↑GlossarySystem without further rights to edit the ↑GlossarySystem. See also ↑UserAPI.

UserAPI - the part of the ↑GlossaryAPI providing all functionalities which should be available for the role ↑user.

whole - a possible role of a ↑concept (first definition) in a ↑part-whole relation. Is realized in the ↑GlossarySchema as role name `wholes` of the association class `HasWhole`.

wildcard - A character that is a placeholder for a number of other characters. See also ↑*any character wildcard* and ↑*one character wildcard*.

Appendix B

List of requirements to the GlossarySystem

Formatting Notes

All terms preceded by an up arrow (↑) are defined in the TextualGlossary.

UA = UserAPI
EA = EditorAPI

B.1 Requirements to the UserAPI

B.1.1 Requirements regarding the use case *Search Concept*

UA-10 PRIO 1 The UserAPI SHALL provide external systems with the ability to search for concepts by specifying a search criterion.

UA-20 PRIO 1 One search criterion SHALL be a comparison between the passed string and

- **a)** the concept's *definition*.
- **b)** the particular *term* of the concept's term.
- **c)** one of the concept's synonyms
- **d)** the concept's example sentences
- **e)** the concept's source

UA-20-1 PRIO 1 If the passed string is compared with one of these elements of a concept, the passed string SHALL either be *contained in* or *match* this element.

UA-50 - UA-80 refer to UA-10 - UA-40.

UA-50 PRIO 1 The UserAPI SHOULD process passed strings ignoring

a) case sensitivity of the passed strings.
Example: It should be possible that "aocs", "AOCS" and "aOcs" are threated equal when searching for a concept.

b) leading and trailig spaces of passed strings.

UA-60 PRIO 1 The UserAPI SHALL process *any character wildcards* in passed strings.

UA-70 PRIO 2 The UserAPI SHOULD process *one character wildcards* in passed strings.

UA-80 PRIO 4 The UserAPI SHOULD remove the HTML markup of

a) term strings of terms

b) definitions of concepts

c) passed strings

before every comparison operation with the corresponding term string, definition or passed string.

UA-90 a) - c) respectively UA-90-1 - UA-90-3 are further search criterions as mentioned in UA-10.

UA-90 The UserAPI SHALL provide external systems with the ability to search for concepts passing:

a) PRIO 1 the current context.

b) PRIO 2 a predefined part of speech.

UA-90-1 PRIO 1 If a context was passed as current context, the concepts in the search result SHALL be ordered in ascending order with reference to the nesting distance to the current context.

UA-90-2 PRIO 2 If a part of speech was passed, the search result SHALL contain only such concepts that have at least one preferred term or synonym that is of the passed part of speech and that contains the passed search term.

UA-100 PRIO 3 The UserAPI SHOULD provide external systems with the ability to get all concepts stored in a glossary.

UA-110 PRIO 3 The UserAPI SHOULD provide external systems with the ability to get all terms stored in a glossary.

UA-120 PRIO 2 The UserAPI SHALL limit

a) the returned concepts
(including contexts)

b) the returned terms

to a predefined limit per call.

This also restricts the concepts and terms returned from UA-100 and UA-110 to this predefined limit per call.

UA-130 PRIO 1 The type of all parameters and results of the UserAPI functions SHALL be either a simple type, or a data transfer object, or a standard Java collection of those types.

Example for such a collection: `List<Concept>`*.*

UA-140 PRIO 1 The UserAPI SHALL ensure that the same concept is returned only once in a single search request.

B.1.2 Requirements regarding the use case *Set Current Context*

UA-200 PRIO 1 All functions of the UserAPI that provide a search for concepts SHALL provide the user with the ability to pass the current context.

B.1.3 Further requirements to the UserAPI

UA-210 PRIO 3 The UserAPI SHOULD provide external systems with the ability to link terms in textual artifacts to concepts in a glossary.

B.2 Requirements to the EditorAPI

B.2.1 Requirements regarding the use cases *Manage Concept* and *Manage Context*

EA-10 PRIO 1 The EditorAPI SHALL provide external systems with the ability to create

a) new concepts
b) new contexts
c) new terms

in a glossary.

EA-10-1 PRIO 1 The creation of a concept SHALL encompass

a) the addition of at least one synonym
b) the definition of exactly one preferred term
c) the creation of a definition
d) the specification of a source
e) the addition of a contains relation to at least one context
f) the definition of a status

for the newly created concept.

*These are the **mandatory** elements every concept must contain. This includes the creation of contexts since contexts are special concepts.*

EA-10-1-1 PRIO 1 The EditorAPI SHALL provide external systems with the ability to search for the term in the glossary that should be used as preferred term for the newly created concept.

EA-10-1-2 PRIO 1 The EditorAPI SHALL provide external systems with the ability to search for the terms in the glossary that should be used as synonyms for the newly created concept.

EA-10-1-3 PRIO 1 The EditorAPI SHALL provide external systems with the ability to search for the contexts in the glossary that should be associated with the newly created concept.

*The concept elements specified in EA-10-2 and EA-10-3 are **optional** elements of a concept. These elements are also optional elements for contexts since contexts are special concepts.*

EA-10-2 PRIO 1 If a new *concept* is created, the EditorAPI SHALL provide external systems with the ability to define an arbitrary number of other concepts as

- **a)** childs
- **b)** parents
- **c)** parts
- **d)** wholes
- **e)** common subtypes
- **f)** individual subtypes
- **g)** supertypes
- **h)** related concepts
- **i)** antonyms

to the newly created concept.

EA-10-3 PRIO 1 If a new *concept* is created, the EditorAPI SHALL provide external systems with the ability to define an arbitrary number of

- **a)** synonyms
- **b)** example sentences
- **c)** transitively contains relations to other contexts
- **d)** intransitively contains relations to other contexts

for the newly created concept.

EA-10-4 PRIO 1 If a new *concept* is created, the EditorAPI SHALL provide external systems with the ability to define the status of the newly created concept.

This includes the status of concepts that are contexts.

EA-10-5 PRIO 1 If a new *context* is created, the EditorAPI SHALL provide external systems with the ability to define an arbitrary number of

- **a)** transitively contained concepts
- **b)** intransitively contained concepts
- **c)** bad terms

for the newly created context.

These are ***optional*** *elements for a context. Further optional elements for a context are specified in EA-10-2.*

EA-20 The EditorAPI SHALL create a unique identifier for every concept, context and term.

EA-30 PRIO 1 The EditorAPI SHALL provide external systems with the ability

- **a)** to edit
- **b)** to delete

the properties and relations of

- **a)** concepts
- **b)** contexts

c) terms

of a glossary.

EA-40 PRIO 1 The EditorAPI SHALL ensure that no concept relation is defined between one and the same concept.

For example, a context may not be contained in itself or a concept may not be a part (see part-whole relation) of itself.

EA-50 PRIO 2 The EditorAPI SHOULD provide external systems with the ability to convert a concept into a context.

EA-60 PRIO 2 If a context has no contains relation to another concept, the EditorAPI SHOULD provide external systems with the ability to convert this context into a concept.

B.2.2 Futher requirements to the EditorAPI

EA-70 PRIO 3 The EditorAPI SHOULD provide external systems with the ability to merge two glossaries (see glossary) into one glossary.

Appendix C

Fulfillment of the constraints of the GlossarySchema in the GlossarySystem

In this appendix, the fulfillment of the constraints of the GlossarySchema in the implementation of the GlossarySystem are listed and detailly explained.

- The constraints *`uuid` must have a value* and *the value of `uuid` must be unique* are fulfilled by the `GraphChangeListener` of class `TGraphGlossaryStorage` of the GlossarySystem generating unique identifiers for every new `ElementEntity` and `HasElement` relation using `UUID.randomUuid.toString()`.

- The constraint *`definition` must have a value* is fulfilled by the `addConcept`, `addContext`, `editConcept` and `editContext` methods throwing an `IllegalArgumentException` if the passed term String (regarding `addConcept` and `addContext`) or the `term` attribute in the passed `Concept` DTO (regarding `editConcept`) or `Context` DTO (regarding `editContext`) has the value `null` or empty String.

- The constraint *`term` must have a value* is fulfilled by the `addTerm`, `addAbbreviation`, `editTerm` and `editAbbreviation` methods of the GlossaryAPI throwing an `IllegalArgumentException` if the passed term String (regarding `addTerm` and `addAbbreviation`) or the `term` attribute of the passed `Term` DTO (regarding `editTerm`) or the passed `Abbreviation` DTO (regarding `editAbbreviation`) is either `null` or the empty String.

- The constraint *`term` must be entire lowercase. Exception: If `POS` has the value `UNIQUE_NAME`, exactly the string is used that the user entered.* is fulfilled by the `addTerm` methods of the GlossaryAPI converting passed term strings to lowercase if the passed part of speech (POS) has another value than `POS.UNIQUE_NAME` and the method `editTerm` of the GlossaryAPI converting term strings of passed `Term` DTOs to lowercase if the POS of the `Term` DTO has another value than `POS.UNIQUE_NAME`.

- The constraint *`HasWhole` is only applicable between `Concepts` whose related `Terms` are nouns* is fulfilled by the `addHierarchicalRelation` method of the GlossaryAPI throwing a `GlossarySystemException` if a relation of type `HasWhole` should be created and one of the preferred terms or synonyms of one of the passed concepts between which the relation should be created has neither the type `POS.NOUN`, nor one of its subtypes `POS.NON_COUNT_NOUN` or `POS.UNIQUE_NAME`.

- The constraint *an `Abbreviation` may only be selected as preferred term if the `Abbreviation` has not more than one abbreviated term* is fulfilled by the method `addPreferredTermToConcept` of the GlossaryAPI throwing a `GlossarySystemException` if the passed `termUuid` belongs to the unique identifier (UUID) of an abbreviation in the glossary and this abbreviation abbreviates more than one term.

- The constraint *`number` must be `NONE` for all `Terms` with `POS` not in { `NOUN`, `NON_COUNT_NOUN`, `VERB` }* is fulfilled by the `addTerm` and `addAbbreviation` methods and the methods `editTerm` and `editAbbreviation` of the GlossaryAPI automatically converting the `Number` of the added or edited term or abbreviation to `Number.NONE` if the part of speech of this term or abbreviation is something different than `null`, `POS.NOUN`, `POS.NON_COUNT_NOUN` or `POS.VERB`.

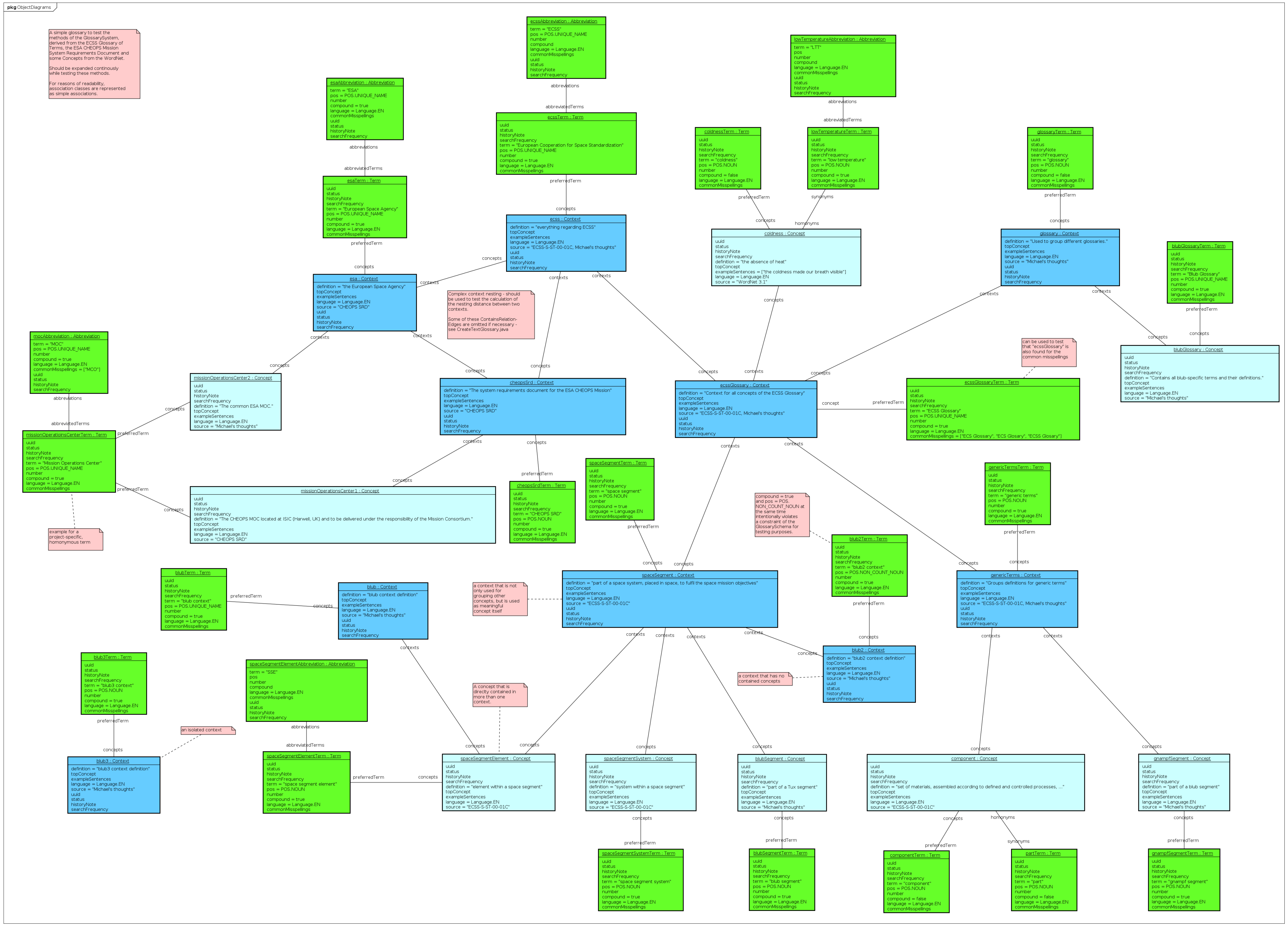
- The constraint *`compound` can only be true if `pos` has either the value `NOUN` or `UNIQUE_NAME`* is fulfilled by the `addTerm` and `addAbbreviation` methods and the methods `editTerm` and `editAbbreviation` of the GlossaryAPI automatically setting the compound of the added or edited term or abbreviation to false if the part of speech of this term or abbreviation is something different than `null`, `POS.NOUN` or `POS.UNIQUE_NAME`.

Appendix D

Test Case Glossaries

To test the GlossarySystem, three main test case glossaries have been used in the JUnit tests. These are listed below as object diagrams. For reasons of readability, the relations of the sample object diagrams are represented as simple associations rather than displaying an object for every association class.

In the GlossarySystem, each TG file corresponding to one of the diagrams below and used for the JUnit tests is generated by a corresponding class in the package `de.uni_koblenz.glossary_system.glossary_creator`.



pkg ObjectDiagrams
A simple glossary to test the methods of the GlossarySystem, derived from the ECSS Glossary of Terms, the ESA CHEOPS Mission System Requirements Document and some Concepts from the WordNet.
Should be expanded continously while testing these methods.
For reasons of readability, association classes are represented as simple associations.
esaAbbreviation : Abbreviation
ecssAbbreviation : Abbreviation
lowTemperatureAbbreviation : Abbreviation
esaTerm : Term
ecssTerm : Term
coldnessTerm : Term
lowTemperatureTerm : Term
glossaryTerm : Term
blubGlossaryTerm : Term
esa : Context
ecss : Context
coldness : Concept
glossary : Context
blubGlossary : Concept
mocAbbreviation : Abbreviation
missionOperationsCenter2 : Concept
missionOperationsCenterTerm : Term
missionOperationsCenter1 : Concept
cheopsSrd : Context
cheopsSrdTerm : Term
ecssGlossary : Context
ecssGlossaryTerm : Term
Complex context nesting - should be used to test the calculation of the nesting distance between two contexts.
Some of these ContainsRelation-Edges are omitted if necessary - see CreateTextGlossary.java
can be used to test that "ecssGlossary" is also found for the common misspellings
example for a project-specific, homonymous term
spaceSegmentTerm : Term
genericTermsTerm : Term
compound = true and pos = POS.NON_COUNT_NOUN at the same time intentionally violates a constraint of the GlossarySchema for testing purposes.
blub2Term : Term
blubTerm : Term
blub : Context
a context that is not only used for grouping other concepts, but is used as meaningful concept itself
spaceSegment : Context
genericTerms : Context
blub2 : Context
a context that has no contained concepts
blub3Term : Term
spaceSegmentElementAbbreviation : Abbreviation
A concept that is directly contained in more than one context.
an isolated context
blub3 : Context
spaceSegmentElementTerm : Term
spaceSegmentElement : Concept
spaceSegmentSystem : Concept
blubSegment : Concept
component : Concept
gnampfSegment : Concept
spaceSegmentSystemTerm : Term
blubSegmentTerm : Term
componentTerm : Term
partTerm : Term
gnampfSegmentTerm : Term
abbreviations
abbreviatedTerms
preferredTerm
concepts
contexts
concept
synonyms
homonyms

Figure D.1: TestCaseGlossary: The glossary used for most of the test cases.

]

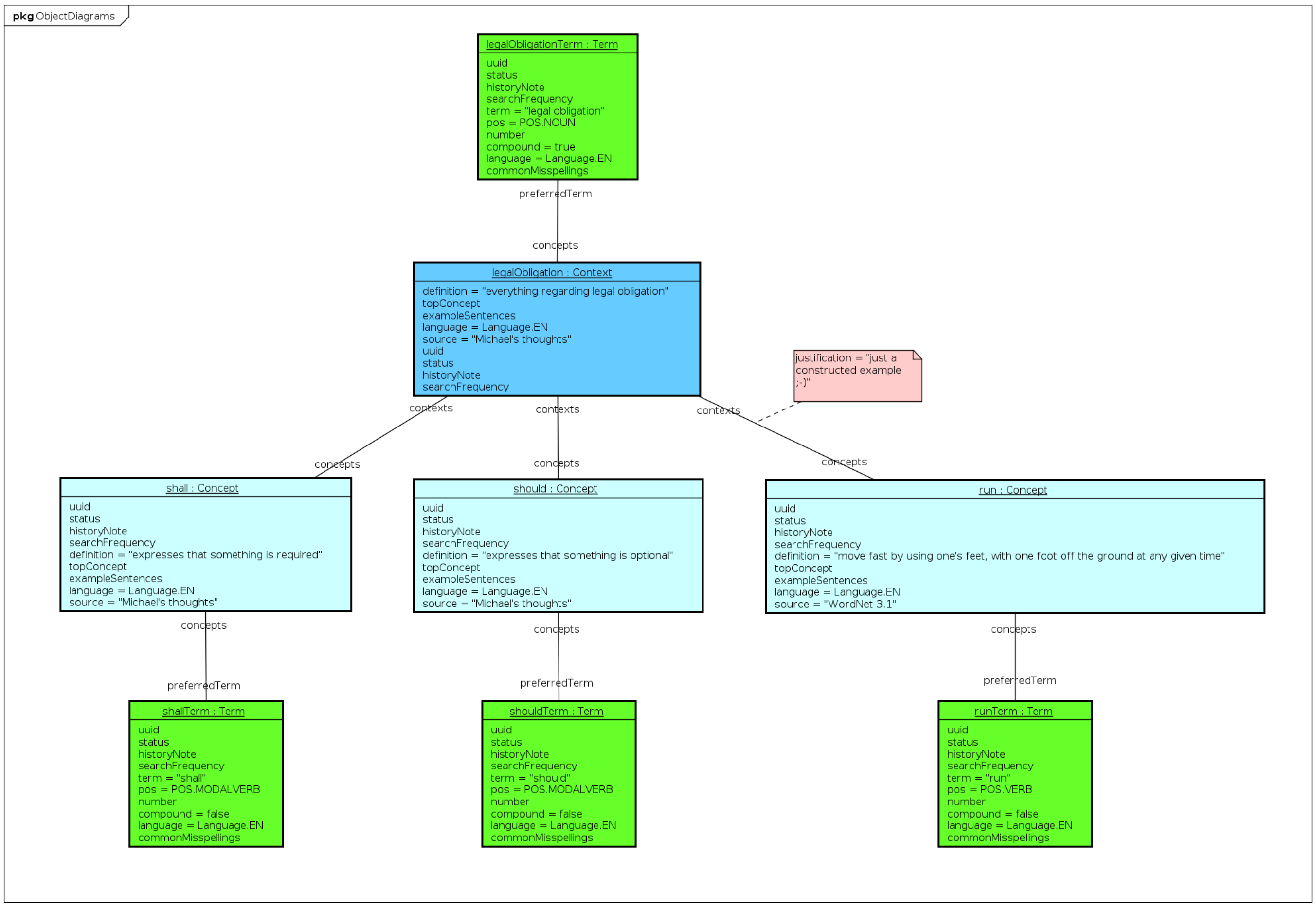


Figure D.2: TestCaseGlossary2: Used in the tests of the search concept methods of the GlossaryAPI.

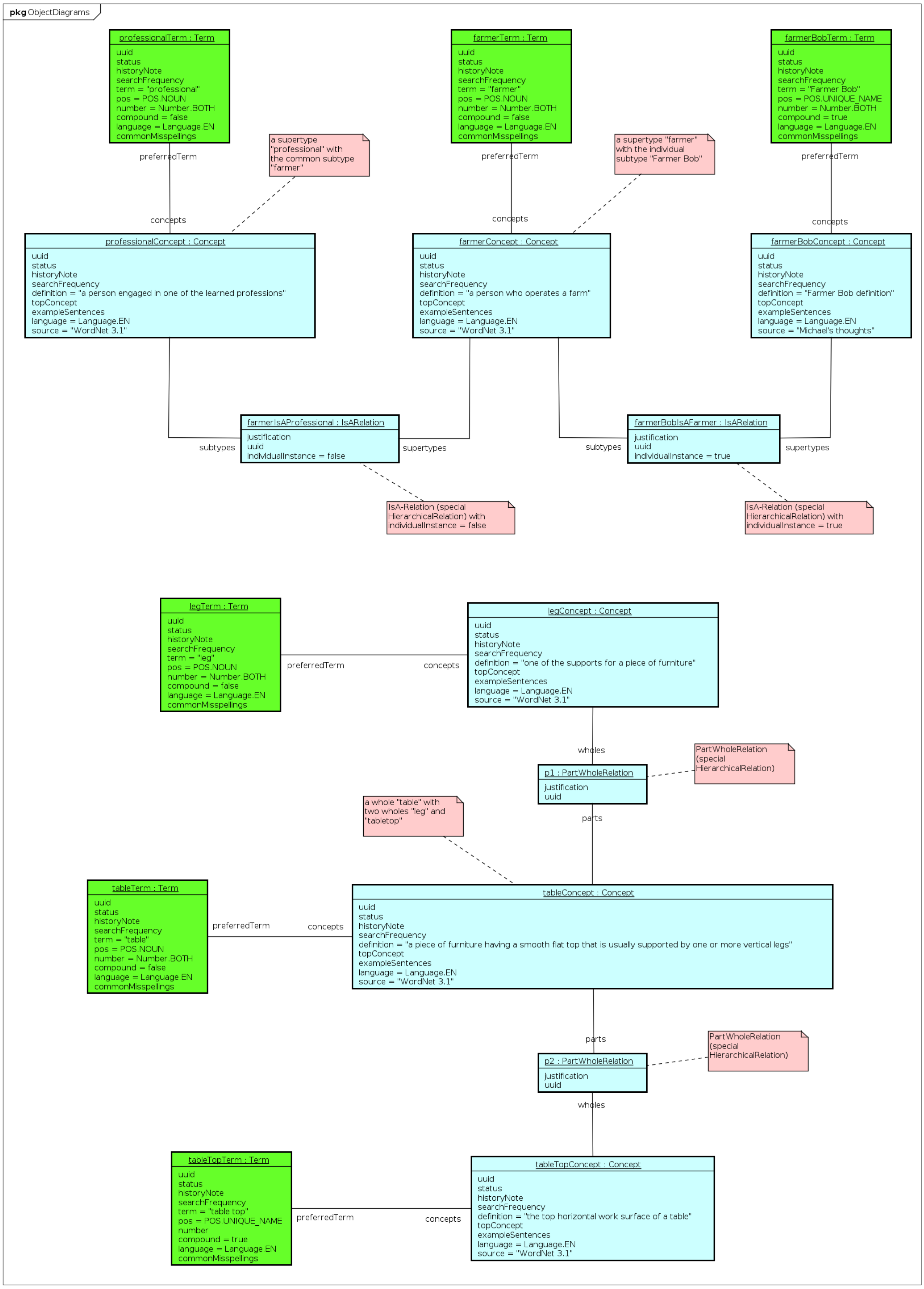


Figure D.3: TestHierarchicalRelationsGlossary

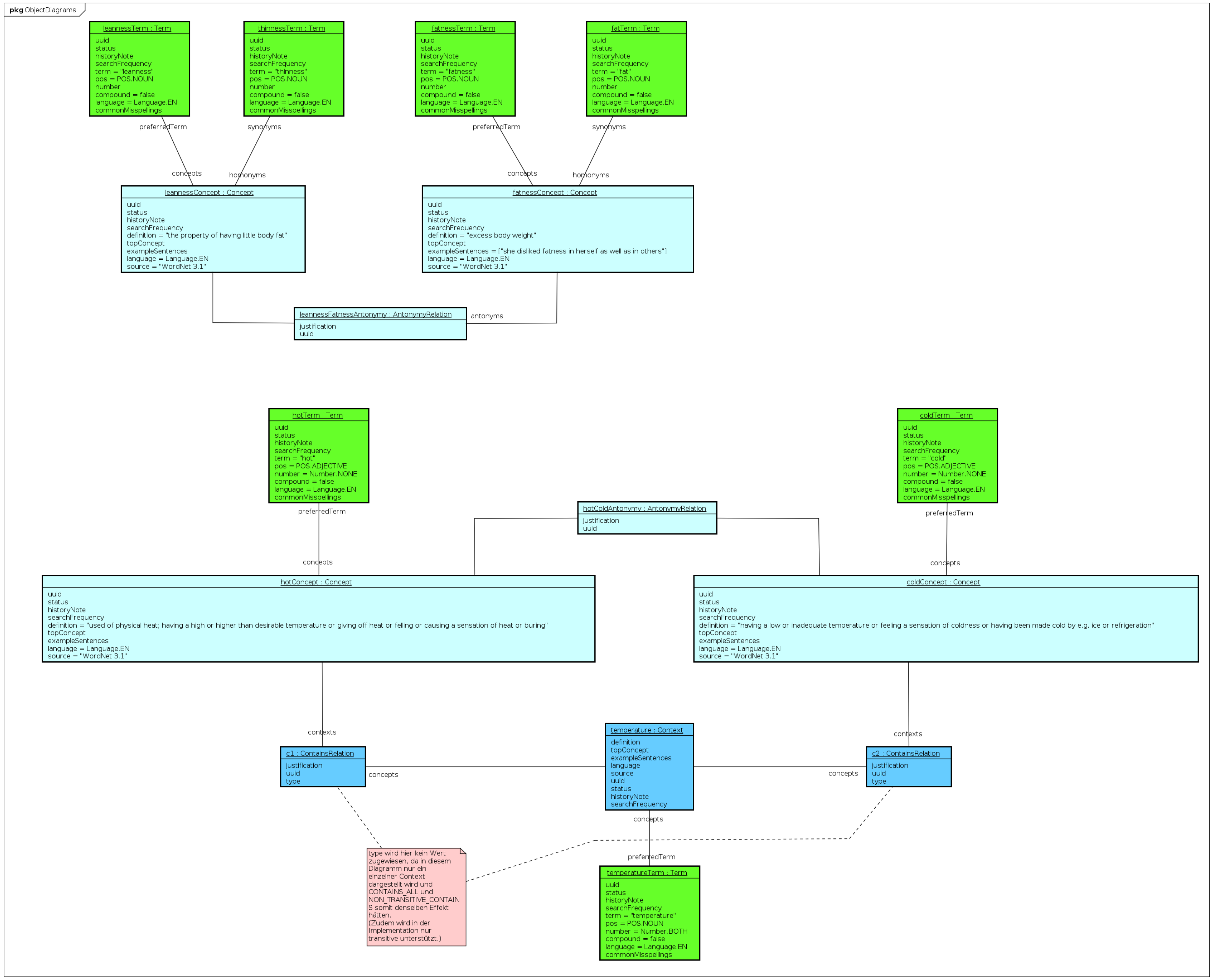


Figure D.4: TestAssociativeRelationsGlossary

Bibliography

[1] European Space Agency (ESA). *Member States & Cooperating States*. https://www.esa.int/About_Us/Corporate_news/Member_States_Cooperating_States. last retrieved on November 10., 2019.

[2] *ANSI/NISO Z39.19-2005 (R2010) - Guidelines for the Construction, Format, and Management of Monolingual Controlled Vocabularies* (NISO - the National Information Standards Organization). 2005. URL: https://www.niso.org/publications/ansiniso-z3919-2005-r2010 (visited on 10/18/2019).

[3] Daniel M. Berry, Erik Kamsties, and Michael M. Krieger. *From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity*. 2003.

[4] Daniel Bildhauer, Tassilo Horn, and Volker Riediger. JGWNL - JGraLab WordNet Library API documentation and manual. Tech. rep. University of Koblenz-Landau, Sept. 2008.

[5] Tony Clark and Jos Warmer. *Object Modeling with the OCL*. Springer-Verlag, 2002.

[6] ECSS Secretariat ESA-ESTEC Requirements & Standards Division. *ECSS Glossary of Terms (ECSS-S-ST-00-01C)*. 2012.

[7] ECSS Secretariat ESA-ESTEC Requirements & Standards Division. *Space engineering - Technical requirements specification*. 2009.

[8] Apache Software Foundation. *Apache Ant Website*. https://ant.apache.org/. last retrieved on January 26., 2020.

[9] The Eclipse Foundation. *Eclipse OCL (Object Constraint Language) website*. https://projects.eclipse.org/projects/modeling.mdt.ocl. last retrieved on November 11., 2019.

[10] Erich Gamma et al. *Design Patterns CD - Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., 1998.

[11] Martin Glinz. *A Glossary of Requirements Engineering Terminology*. 2014.

[12] *ISO 25964-1:2011(E) The International Standard for Thesauri and Interoperability with other Vocabularies - Part 1: Thesauri for Information Retrieval* (ISO - The International Standards Organization). 2011. URL: https://www.niso.org/schemas/iso25964 (visited on 09/02/2019).

[13] *ISO 25964-2:2013(E) The International Standard for Thesauri and Interoperability with other Vocabularies - Part 2: Interoperability with other Vocabularies* (ISO - The International Standards Organization). 2013. URL: https://www.niso.org/schemas/iso25964 (visited on 09/02/2019).

[14] Axel van Lamsweerde. *Requirements Engineering*. A John Wiley and Sons, Ltd., Publication, 2009.

[15] *Module Manual help page*. https://ist.uni-koblenz.de/MoMa/help;MOMA2SID=bf5ea5f15134823dbe002868896b. last retrieved on October 08., 2019.

[16] National Aeronautics and Space Agency Scientific and Technical Information Program (NASA STI). *NASA thesaurus*. `https://www.sti.nasa.gov/thesvol1.pdf`. 2012.

[17] *Oxford English Dictionary webinterface*. `https://www.lexico.com/en`. last retrieved on November 15., 2019.

[18] Dr. Volker Riediger. *lecture* Informatik für IM III - Softwaretechnik. University of Koblenz-Landau, 2017.

[19] ECSS Secretariat. *Space engineering Thermal control general requirements*. European Cooperation for Space Standardization (ECSS), 2008.

[20] Die SOPHISTen. *MASTeR - Schablonen für alle Fälle*. SOPHIST GmbH. 2013.

[21] CHEOPS Project Team. *CHEOPS System Requirements Document*. European Space Research and Technology Centre, 2013.

[22] Euclid Project Team. *Euclid System Requirements Document*. European Space Research and Technology Centre, 2012.

[23] FLEX Team. *FLEX Space Segment Requirements Document*. European Space Research and Technology Centre, 2017.

[24] GMES Sentinel-1 Team. *GMES Sentinel-1 System Requirements Document*. European Space Agency, 2006.

[25] JUICE Project Team. *JUICE - Jupiter Icy Moons Explorer Space Segment Requirements Document*. European Space Research and Technology Centre, 2014.

[26] PLATO Project Team. *PLATO System Requirements Document*. European Space Agency, 2017.

[27] PROBA-V Team. *PROBA Vegetation System Requirements Document*. European Space Agency, 2008.

[28] The JGraLab Team. *JGraLab GitHub page*. `https://github.com/jgralab/jgralab`. last retrieved on January 26., 2020.

[29] The JGraLab Team. *JGraLab TGraphs Wiki Page*. `https://github.com/jgralab/jgralab/wiki/TGraphs`. last retrieved on January 26., 2020.

[30] The JGraLab Team. *JGraLab Wiki*. `https://github.com/jgralab/jgralab/wiki`. last retrieved on January 26., 2020.

[31] Winfried Ulrich. *Wörterbuch Linguistische Grundbegriffe*. Berlin: Gebrüder Borntraeger, 2002. ISBN: 3-443-03111-0.

[32] *WordNet home page*. `https://wordnet.princeton.edu/`. last retrieved on October 03., 2019.

[33] *WordNet webinterface*. `http://wordnetweb.princeton.edu/perl/webwn`. last retrieved on November 13., 2019.