Software Case Marking Language Definition

Deliverable D4.3, version 1.00, 20.11.2007

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

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

Software Case Marking Language Definition

Workpackage	WP4
Task	T4.3
Document number	D4.3
Document type	Deliverable
Title	Software Case Marking Language Definition
Subtitle	
Author(s)	Daniel Bildhauer, Jürgen Ebert, Volker Riediger, Katharina Wolter, Markus Nick, Andreas Jedlitschka, Sebastian Weber, Hannes Schwarz, Albert Ambroziewicz, Jacek Bojarski, Tomasz Straszak, Sevan Kavaldjian, Roman Popp, Alexander Szep
Internal Reviewer(s)	Daniel Bildhauer, Hannes Schwarz, Katharina Wolter
Internal Acceptance	Project Board
Location	https://svn.redseeds.eu/svn/redseeds/1_DeliverablesSpace/WP4_-Technologies_for_reusable_cases/D4.3/ReDSeeDS_D4.3.00_Software_Case_Marking_Language_Definition.tex
Version	1.00
Status	Final
Distribution	Public

20.11.2007

History of changes

Date	Ver.	Author(s)	Change description
04.10.2007	0.01	Daniel Bildhauer, Jürgen Ebert, Hannes Schwarz	Proposition of ToC
22.10.2007	0.02	Daniel Bildhauer	First content for Chapter 4
23.10.2007	0.03	Katharina Wolter (UH)	First content for Chapter 3
24.10.2007	0.04	Hannes Schwarz (UKo)	Preface for chapter 5
26.10.2007	0.05	Hannes Schwarz (UKo)	Contents for sections 1.1, 1.2, 1.4, 1.5
28.10.2007	0.06	Katharina Wolter (UH)	Further content for Section 3.1, 3.2
31.10.2007	0.07	Hannes Schwarz (UKo)	Preface for chapter 6
01.11.2007	0.08	Katharina Wolter (UH)	Restructuring of Chapter 3 and further content for 3.1, 3.2
02.11.2007	0.09	Katharina Wolter (UH)	Content for Sections 3.3 and 6.1
06.11.2007	0.10	Albert Ambroziewicz (WUT)	Content for Section 6.2
06.11.2007	0.11	Katharina Wolter (UH)	Further content for Section 6.1
07.11.2007	0.12	Jacek Bojarski, Tomasz Straszak (WUT)	Initial content for Section 6.3
07.11.2007	0.13	Daniel Bildhauer	Further content for Chapter 6
08.11.2007	0.14	Katharina Wolter (UH)	Additions and corrections for Sections 3.1, 3.3, 6.1 and citations
09.11.2007	0.15	Daniel Bildhauer	Added initial content for conclusion and summary
09.11.2007	0.16	Katharina Wolter (UH)	Added Section 6.2.1
09.11.2007	0.17	Hannes Schwarz	Added section 5.2
13.11.2007	0.18	Albert Ambroziewicz (WUT)	Additions to section 6.2

Date	Ver.	Author(s)	Change description
13.11.2007	0.19	Sevan Kavaldjian, Roman Popp, Alexander Szep (TUW)	Added Section 6.2.3
14.11.2007	0.20	Hannes Schwarz	Modified chapter 2 and summary
14.11.2007	0.21	Daniel Bildhauer	Added missing content for section 3.3
15.11.2007	0.22	Sebastian Weber, Markus Nick, Andreas Jedlitschka	Added section 5.1
16.11.2007	0.23	Hannes Schwarz	Additions to section 5.3
20.11.2007	1.00	Hannes Schwarz	Finalisation

Summary

This document deals with the definition of a mechanism for identifying and marking the results of a query. A query in ReDSeeDS includes requirements stemming from a current project (a *current software case*). These requirements are compared to the requirements of *past software cases* stored in a repository. In order to enable such a comparison, a mapping between the requirements of the two software cases is defined. On the basis of this mapping, approaches for visualising the differences between the requirements are described, enabling the ReDSeeDS user to select a set of requirements for reuse.

The selected requirements serve as starting point for the computation of a so-called *partial software case* containing those elements of a past software case's architecture, design and code which are related to the requirements and thus eligible for reuse, too. In this document, approaches for the computation of past software cases as well as for their visualisation are presented.

Note that the title of this document, *Software Case Marking Language Definition*, stems from the ReDSeeDS project contract's annex "description of work". In fact, the title can be misleading, for this document does not define a *language*, but rather a coherent concept for identifying and marking the results of a query serving as foundation for the development of the ReDSeeDS engine which is to be developed in workpackage 5 of the ReDSeeDS project.

Table of contents

List of figures

List of tables

Chapter 1

Scope, conventions and guidelines

1.1 Document scope

This document provides an overview of the solution marking mechanism used to identify and visualise those parts of a past software case which are reusable in the current case. To this end, it is described how to employ the similarity measures introduced in ReDSeeDS deliverable D4.2 [WKB+07] to create a mapping between requirements of the current and a past software case. Based on the similar, thus reusable requirements of the past software case, it is then possible to retrieve related parts of the case's architecture, design and code. This activity is pursued on the basis of traceability information manifesting the relations between the mentioned parts.

Once the potentially reusable parts of a software case have been retrieved, they have to be displayed to the user, including a visualisation of the similarity mapping between the requirement specifications. This document presents different techniques in order to accomplish this task.

1.2 Conventions

The following notation conventions are used throughout this document:

- *Italics font* is used for emphasised text, e.g. *Web Ontology Language*.
- Sans-serif font is used for names of model elements such as classes, attributes and associations, e.g. Requirement.

- Package names are separated from class names by two colons ("::"). If the package a class belongs to can be gathered from the context, the package name may be omitted for brevity.
- Keywords or other elements of textual language such as the Software Case Query Language or SQL are written using `typewriter font`.

1.3 Related work and relations to other documents

The first issue discussed in this document, the mapping between the requirements of the current and a past software case, relies upon similarity measures which are based on case-based reasoning, information retrieval, graph theory, and description logics. These measures are introduced in deliverable D4.2 [WKB$^+$07], where other related work can be found, too.

The notion of a *software case* is described by Jedlitschka and Nick in [JN06]. The terminology used in the approach for computing partial software cases as well as its elementary ideas are roughly based on the concept of *program slicing* introduced in [OO84, Wei84].

Related work leading to the approaches for visualising differences between requirements and partial software cases includes the *diff*-algorithm [HM76] and the visualisation of ontologies and ontology alignments, e.g. described in [FSH02, LS06].

1.4 Structure of this document

Following this chapter on scope, conventions and guidelines, chapter 2 gives a general introduction on work already done which is relevant for the contents of this document. Further on, it makes central assumptions affecting the scope of following chapters.

Based on a brief summary of the results of deliverable D4.2 [WKB$^+$07], chapter 3 describes the mapping of requirements originating from two different software cases.

While chapter 4 illustrates how requirements from a past software case can be selected in order to reuse them in a current case, chapter 5 introduces the notion of *partial* software cases which are computed based on this requirements selection.

Chapter 6 deals with the visualisation of the concepts described in previous chapters: differences between requirements and partial software cases. Selected requirements in a software case are considered to be contained within the partial software case they originate.

Finally, Chapter 7 summarises the whole document.

1.5 Usage guidelines

The description of the solution marking mechanism should be used as a book that guides the reader through the basic technologies, ideas and concepts of the solution marking mechanism to mark similar cases in the software case repository. Therefore, this document is mainly intended for members of workpackage 5 of the ReDSeeDS project who are implementing the ReDSeeDS system prototype.

The document could also be useful for advanced ReDSeeDS users who wish to understand the background and main concepts of the marking mechanism.

Chapter 2

Introduction

The fourth work package of the ReDSeeDS project addresses the development of concepts and technologies facilitating the retrieval of software cases or parts thereof. The choice of retrieved artefacts is based on a complete or partial requirement specification given as argument of some query to the repository holding the software cases (see deliverable D4.1.1 [BEK+07]). A query engine has to compare the requirements given as argument of the query to requirements of software cases stored in the repository.

This document deals with the definition of a mechanism for marking the results of a query. These include a mapping between the requirements given in the query, i.e. the requirements of a *current software case*, and the requirements of a *past software case* in the software case fact repository. On the basis of this mapping, the differences and commonalities between the requirements specification of the two software cases could be displayed to the user. Thus he is enabled to review and potentially modify the requirements the ReDSeeDS engine has selected for reuse. It is important to understand that such a mapping is binary in nature, i.e. a past software case is mapped to the current software case *only*. The engine does not establish some kind of relation between past software cases whatsoever.

Following the selection of a set of requirements for reuse, a *partial software case* has to be computed. A partial software case contains those elements of the architectural, design and code parts of a (complete) software case which are related to the selected requirements and which consequently are candidates for reuse themselves. Once a partial software case has been computed, it also has to be visualised to the user so that he can manually modify it if required.

Chapter 3

Mapping Information for the Solution Marking

In this chapter, the results of deliverable D4.2 "Software Case Similarity Measure Definition" [WKB+07] are related to the solution marking mechanism defined in this deliverable.

The first section 3.1 defines the information the solution marking requires. Section 3.2 gives a short overview of the similarity calculation algorithms described in detail in deliverable D4.2 and then determines whether the algorithms provide the information needed for the solution marking.

Finally, in Section 3.3, the data structure for the mapping is described.

3.1 Weighted Mapping of Requirement Model Elements

In order to support the reuse of past software cases it is necessary to mark similarities and differences between a retrieved past software case (PC) and the current software case (CC) or a query. The information needed for this marking is a mapping between the elements of the current software case (or query) and a past software case or more generally speaking a mapping between two (partial) RSL requirements models. This mapping should relate an element of one requirement model with an element of the other requirement model if the elements are equal or similar. Elements that cannot be mapped constitute the differences between the models.

The mapping is a relation R over the set of elements of the CC and the set of elements of the PC with a corresponding similarity value. The relation is not functional since one element of the CC may have the same similarity to two elements of the PC. Furthermore, the relation is not left-total since not all elements of the CC can be mapped to elements of a PC and not right-total for analogical reasons.

Since the mapping defines equal and similar model elements it suggests itself to use similarity measures to determine the mapping between two (partial) requirements models. The similarity calculation algorithms described in deliverable D4.2 [WKB+07] differ in their suitability for this mapping task. The next section first summarises the main characteristics of these similarity calculation algorithms and then motivates which algorithms can be used to determine the mapping.

3.2 Information provided by Similarity Measures

3.2.1 Overview of Similarity Measures

In ReDSeeDS an (initial) requirements specification is used to retrieve similar software cases from the fact repository. Thus, a similarity measure is needed that compares a query with the requirements models of Past Software Cases. A query can specify:

- a set of Terms
- a set of DomainElements
- a set of DomainElementPackages
- a complete DomainSpecification
- a set of sentences with associated DomainSpecification
- a set of Requirements with associated DomainSpecification
- a set of RequirementsPackages with associated DomainSpecification
- a complete requirements model (containing a RequirementsSpecification and a DomainSpecification

Measures that capture the similarity of artefacts have been developed in many research areas and for different types of artefacts. In deliverable D4.2 [WKB+07] we provided a detailed introduction of similarity measures that are relevant for the ReDSeeDS project, namely measures from the following research areas:

- Information Retrieval (IR),
- Case-based Reasoning (CBR),
- Description Logics (DL) in combination with CBR and
- Graph-based Similarity.

ReDSeeDS supports different types of requirements representations: descriptive representations and model-based representations. Due to the diversity of representations a combination of similarity measures is needed to compare a query with a software case. The similarity measures have different preconditions and provide different results. Information Retrieval, for example, can be used for any document containing a significant amount of text. However, conventional IR measures only consider the lexicographic similarity of terms or phrases. Graph-based similarity requires querying and cases in a graph representation consistent with a given metamodel, while description logics in our application require query and cases in OWL representation consistent with the RSL metamodel. All measures provide a similarity value, i.e. a double value between 0 and 1. Additionally, the graph-based similarity provides a result table that contains matched element pairs and their similarity.

3.2.2 Suitability of Similarity Measures for Mapping Calculation

Based on the brief overview of similarity calculation algorithms we now determine which measures provide the mapping information between query elements and elements of past software cases needed by the solution marking.

Information retrieval and CBR do not consider the RSL type of elements (e.g. Notion, Noun, Verb, Actor, etc.) contained in the current software case and a past software case, nor the structure between these elements. The approaches mainly compare the strings contained in the cases. The following two sentences illustrate that this is not sufficient for the solution marking mechanism:

The system displays the print options.

The user can overview all options on the display.

A mapping based on these similarity measures would map the Verb *displays* in the first sentence with the SystemElement *display* in the second sentence. Thus, these similarity measures are not appropriate to define a mapping between the elements of a query and a Past Software Case.

The graph-based similarity measure does consider the RSL type of elements. As stated above the graph-based similarity measure provides a result table with matched element pairs and their similarity. Thus, this algorithm provides the needed information.

The similarity measure based on description logics does not provide the needed mapping information itself. However, the main goal of *ontology alignment* systems is to provide a mapping between elements belonging to different ontologies and that share the same or a similar semantics [Euz04]. A detailed formal definition of ontology alignment can be found in [Bou05]. In [ES04] the authors define a ontology mapping function as follows:

$$map : O_{i_1} \rightarrow O_{i_2} \tag{3.1}$$

$$map(e_{i_1 j_1}) = e_{i_2 j_2}, \text{if } sim(e_{i_1 j_1}, e_{i_2 j_2}) > t(\text{threshold}) \tag{3.2}$$

Thus, ontology alignment systems could also be applied to compute the needed mapping information. An extensive overview of tools can be found in [Euz04, ES07].

Figure 3.1 shows the similarity measures. For each measure the table shows which types of artefacts are compared, which input is required and which results are provided. The last row states whether the results provided by each similarity measure is sufficient for the solution marking.

The table in Figure 3.1 shows that two different methods can be applied to determine the mapping information. Independent from the method the mapping information should be stored in the data structure described in the next section.

Approach	Information Retrieval / Vector Space Model	Structural CBR	Description Logic	Graph-based Similarity
Elements compared	▪ text-based documents ▪ structure is not analysed by state-of-the-art approaches	▪ cases of identical structure (attributes) with different values for the attributes	▪ concept definitions that are consistent with RSL metamodel (contained in TBox)	▪ graphs of any structure
Input required	▪ vector representation for query and cases	▪ case representations needed ▪ automated generation of representations possible ▪ additional information can be added	▪ OWL representation of query and cases needed ▪ JgraLab -> OWL converter has been developed by University of Koblenz	▪ graph representation of query and cases needed
Result provided	▪ value in [0..1]	▪ value in [0..1]	▪ value in [0..1]	▪ value in [0..1] ▪ table of matched element pairs with similarity value
Suitable for solution marking	▪ lexicographic similarity is not sufficient for the solution marking	▪ lexicographic similarity is not sufficient for the solution marking	▪ ontology alignment provides the mapping information	▪ provides the needed mapping information

Figure 3.1: Comparison of Similarity measures with respect to their application in ReDSeeDS.

3.3 Data Structure for the Mapping Information

The mapping information consists of three main parts: the two requirements models that are mapped and a mapping table. This mapping table contains a set of mappings. A mapping consists of two RSL elements, one from each requirements model and a similarity value (double value between 0 and 1).

Thus, the mapping can be described by a small metamodel as depicted in figure 3.2. The whole mapping information is depicted by the class MappingInformation which is composed of two SoftwareCases and one MappingTable. One of the SoftwareCases contains the RSL RepresentableElements of the query. The other RepresentableElements stem from a past software case stored in the fact repository. The mapping table consists of several MappingEntrys, each describing the mapping of one RepresentableElement of the query to one RepresentableElement of the past software case. Additionally, each MappingEntry contains a similarity value, whose most important part is a double value in the interval [0,1]. A more detailed description of the similarity can be found in deliverable 4.2 [WKB+07].

As a design decision, this mapping information is not part of the requirements model itself, but stored separately. The main advantages of this approach are, that the requirements models must not be updated every time a new mapping is computed and that the RSL meta model needs not to be changed. This helps to keep different concerns separate and thus to keep the systems architecture as simple and understandable as possible. There are several alternatives to store the mapping information. For example, a relational database could be used as well as a separate graph or an OWL file. Alternatively, the mapping could be stored as a set of Java mapping objects. As requirements are already stored as graphs in the fact repository, which is JGraLab as decided in deliverable D4.4 [BER+07], there is already a Java object for each requirement in the fact repository. Thus, it seems to be reasonable to use these objects and to create a separate Java class Mapping. Instances of this class would have references to two RepresentableElement vertices stored in the fact repository, which are in fact Java objects. Hence, no additional technology or separate “mapping repository” is necessary to store the mapping, which is by the way only temporarily needed and therefore needs not to be stored permanently.

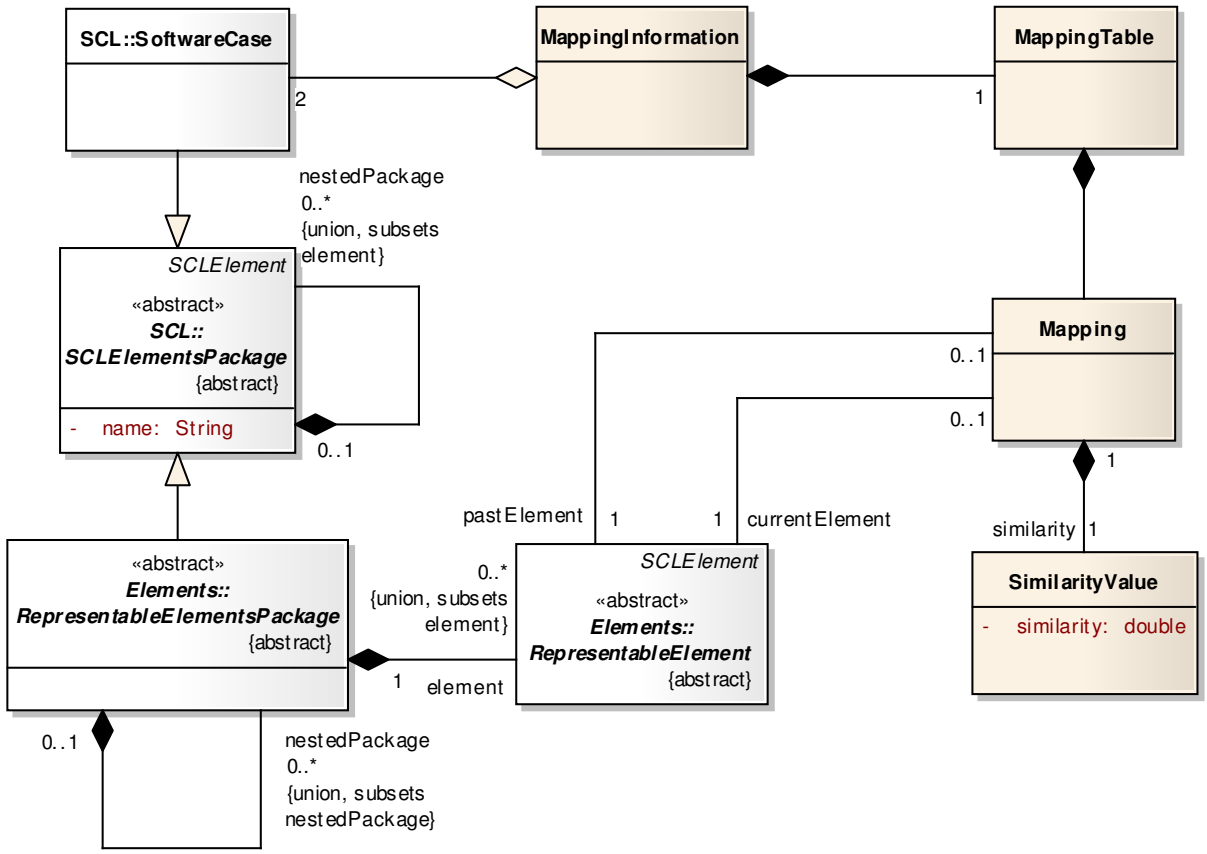


Figure 3.2: Data structure for the mapping information between two (partial) RSL requirements models.

Chapter 4

Selection of the requirements set to reuse

Whereas the previous chapter has depicted the mapping between requirements elements as a result of the similarity calculation, the focus of this chapter is the selection of the requirements elements that should be reused. Section 4.1 gives a detailed introduction why this selection is necessary, while section 4.2 describes how this set of requirements could be calculated automatically. The last section in this chapter 4.3 explains how the automatic selection can be adjusted.

4.1 General concept

The goal of ReDSeeDS is the reuse of software cases stored in a repository on the basis of the requirements. To reach this goal, requirements of the current software case are compared with the requirements of the stored software cases. One result of this comparison is a similarity mapping which maps at most one requirement of the past software cases to each requirement of the current software case. This mapping is based on the similarity values calculated for the requirements. Each requirement of the current software case is mapped to the most similar one in the past software cases.

In the reuse step, the requirements of the past software cases that are mapped to at most one requirement of the current software case are the starting point for reuse. The parts of the architecture, detailed design and code that are associated to these requirements by traceability links are then computed by the ReDSeeDS engine. These calculated parts are candidates for reuse in the current software case.

Obviously, the number and kind of the artefacts that are candidates for reuse highly depends on the requirements that are part of the mapping. Hence, to get a valuable selection of reuse candidates, the requirements set of the mapping must be of good quality, i.e. match the requirements of the current software case to a maximum degree. To ensure this quality, there are two non-exclusive alternatives. One is a permanent test and improvement of the similarity calculation algorithms. The other is to allow the ReDSeeDS user, i.e. the developer of the current software case, to adjust the set of chosen requirements. On the one hand, the test and improvement of the algorithms can not guarantee that the "right" requirements are mapped in every case. Thus, the manual adjusting of the mapping must be possible. On the other hand, also the manual adjusting may lead to mistakes and a complete manual adjusting will take too much time and human resources. Hence, both alternatives are of great value. The following two sections will describe both alternatives.

4.2 Automatic selection of the requirements set

The introduction has already explained why a automatic selection of the requirements to reuse is necessary. This section will describe the details of the selection. One result of a query to the fact repository formulated in the Software Case Query Language (SCQL) is a mapping between the requirements of the past software cases stored in the repository and the requirements of the current software case. The set of requirements that is the starting point for reuse of a past software cases corresponds to the requirements that are part of this mapping. To only get the artefacts that are of a reasonable value for reuse, it is necessary to only include the requirements with a high enough similarity to one of the requirements of the current software case in this set.

To reach this goal, an internal threshold which ensures that only the requirements that are of a reasonable value for reuse are selected. To specify an exact value for this threshold, further experiments with the ReDSeeDS prototype are needed. It is not sufficient to fix the threshold to, say, 0.5, for there are two problems with the threshold value. On the one hand, there is no possibility to reuse something if there are no requirements that have a similarity which is greater than the threshold. This will lead to the development of a completely new software case without reusing anything. On the other hand, if all requirements are selected for reuse, there are too much artefacts that should be reused but have no real value for the current software case. Thus, the threshold value should be a result of detailed experiments with the ReDSeeDS prototype.

It is possible that these experiments show that a fixed threshold value is not sufficient. Instead of this, a dynamic threshold depending on the number and similarities of the several requirements

may be needed. Which algorithm is used to calculate such a threshold can only be guessed at this point in the course of the ReDSeeDS project, especially as it is not clear if it is necessary at all. Thus, the calculation algorithm can only be a result of detailed experiments with the ReDSeeDS prototype.

As described in chapter 3 of deliverable 4.1.1 [BEK+07], SCQL supports the specification of a threshold value by the ReDSeeDS user. This value is used to filter complete software cases, i.e. to use only the requirements of software cases that have a minimum similarity to the query. This is useful since the reuse of great parts of a small number of software cases probably needs less effort than the reuse of small parts of a great number of software cases. Besides this usage, the threshold specification of SCQL could possibly be used to specify the threshold for the requirements selection. Probably it is not sufficient to use the same threshold value for software case search and requirements selection, but the SCQL-threshold could be used as one input of an automatic adjusting of the threshold for the requirements selection.

As a third alternative, the user should be able to specify a separate threshold for the requirements selection. It should be possible to define such a threshold globally in the ReDSeeDS tool for all requirement selections as well as separately for each selection. The definition for the current selection should override the global setting and the global setting should override the internal threshold definition as well as the automatic threshold calculation.

4.3 Manual adjusting of the requirements set

Whereas the previous section has depicted the possibilities of adjusting the set of requirements to reuse by (semi)automatic mechanisms, this section explains which possibilities a ReDSeeDS user should have to change the selection of the requirements to reuse. As mentioned above, the automatic selection of the requirements and the selection intended by the ReDSeeDS user may differ, independently of the quality of the selection algorithms. Thus, the ReDSeeDS user must be able to change the selection, e.g. remove or add requirements from or to the selected set.

To allow the user to change the requirement selection, the list of selected requirements must be presented to the user. One possible variant is to display a tree view of the requirements and the packages the requirements are grouped in. To display the selected requirements and to allow modifications of the selection, every requirement should be displayed with a checkbox in front of it. For convenience, also the requirements packages should be displayed with a checkbox. By checking or unchecking such a checkbox, the user can (de)select the requirement or all the requirements in the package, respectively.

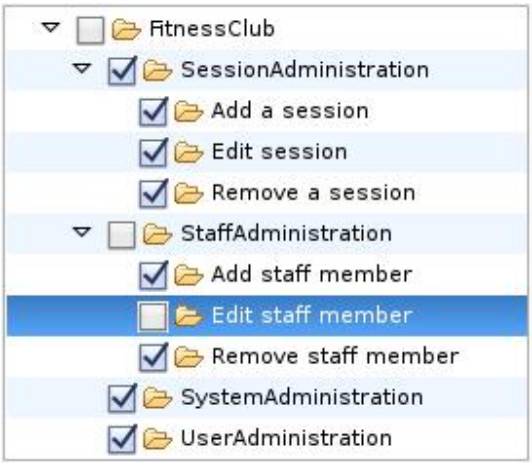


Figure 4.1: Requirements TreeView with checkboxes

The screenshot in figure 4.1 shows how such a presentation of requirements may look like. The requirements that are checked are included in the set of requirements that should be reused. Further, also all requirements in checked packages are included in the set to reuse.

By default, the selection in this view should represent the automatic selection described in section 4.2. After the user has adjusted the automatic selection to fit his needs, the reuse of the software artefacts connected to the selected requirements by traceability links starts. The first step that is performed to reuse the artefacts is the computation of the partial software case. This step is described in the next chapter in detail.

Chapter 5

Partial software cases

The previous chapter 4 deals with the selection of a set of requirements of a past software case which are eligible for reuse in a current software case. This selection can be done either manually, automatically on the basis of calculated similarities measures between requirements of the current case and the past case, or by a combination thereof.

This chapter deals with *partial* software cases, also called *slices*. A slice denotes those parts of a complete software case which are, generally speaking, somehow related to a particular set of elements of that software case, the *slicing criterion*. These relationships are recorded by traceability links. In the context of ReDSeeDS, the slicing criterion corresponds to the above-mentioned set of requirements. Consequently, the elements of a partial past software case are candidates for reuse in the current software case.

Section 5.1 details the notion of partial software cases more specifically. Section 5.2 gives information on how to compute such a slice starting from a selected set of requirements. The last section 5.3 of this chapter contains an example of a partial software case and its computation.

5.1 Notion of partial software cases

A naive top-level representation schema for software cases would be to link requirements and other software artefacts that compose a software case according to the refinement during the software development process. Artefacts can be requirements (R), architectural items (A), design items (D) and code items (C) which are part of the R-A-D-C 'abstraction levels'. To allow software artefacts to be part of several partial software cases, many-to-many relation-

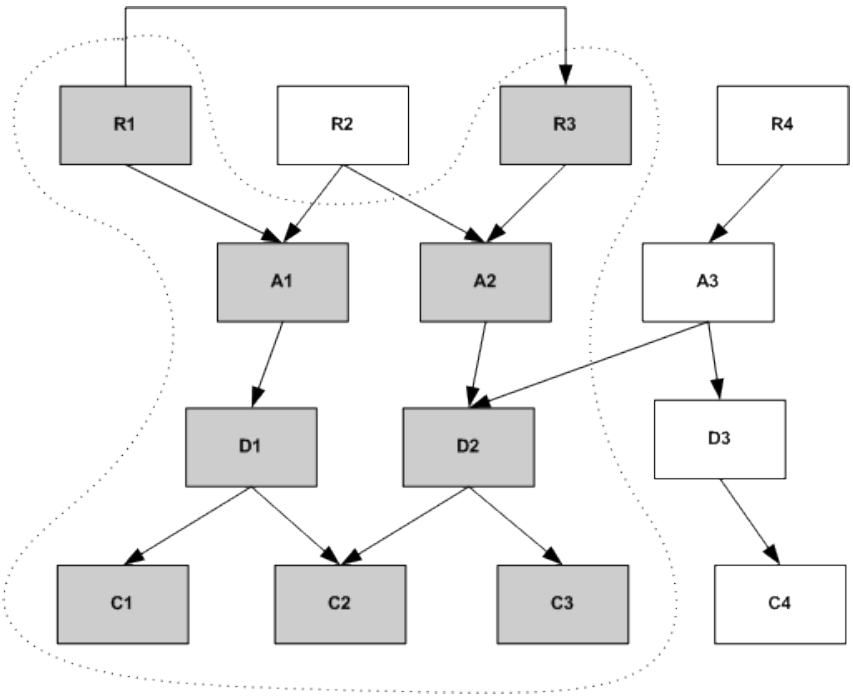


Figure 5.1: The entirety of all artefacts composes the whole software case. As an example, the highlighted artefacts are part of a partial software case. Arrows constitute traceability links between the artefacts.

ships between the artefacts are necessary. This allows selecting partial software cases using requirements. The 'whole' software case, i.e. the complete software system which comprises all software artefacts created during the complete software development process, consists of the entirety of all partial software cases (confer figure 5.1).

Looking at a whole software system using a kind of reusable software case view, i.e., R-A-D-C traceability links from requirements over architecture item, detailed design item to code item, this could show (1) rather hierarchical R-A-D-C traceability links or (2) rather 'Spaghetti' R-A-D-C traceability links (see figure 5.2). We expect that the hierarchical traceability links make the reused parts better maintainable because each partial software case could be replaced without affecting another reused partial software case. With the 'Spaghetti' traceability links, the replacement of the software case for one requirement would affect overlapping partial software cases, e.g., an overlap in the code would lead to a conflict because two cases require different versions or variants of the same code item. Although the first variant would be the ideal case in reality an architectural artefact has to fulfil more than one requirement. The same statement can be applied to code artefacts. As a result, like in figure 5.1, one artefact of one level can have multiple traceability links to the next level. This across-the-level relationships are called 'inter-level' traceability links. 'Intra-level' traceability links are between artefacts on the same level.

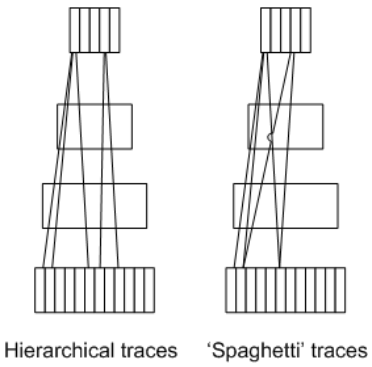


Figure 5.2: Two kinds of R-A-D-C traces

Modifications (defect removal, improvements, adaptation) lead to issues of variants and versions. This means that software cases will overlap regarding the software artefacts they consist of. Like in software product lines, reusable software cases will have parts that they share (commonalities) and parts where they are different (variations). If the problem of versions and variants was to be addressed, a version/variant relationship on reusable software case would be needed. Furthermore, a version/variant relationship on the artefact types (requirements, architecture item, detailed design item, code) would be needed as well. Such a version/variant relationship has to be able to describe (1) chains of versions, (2) branches by variants, and (3) merges of variants. The chains require 1:1 relationship. While the branches require a 1:n relationship, the mergers require n:1 relationships.
In *ReDSeeDS* this version/variant problem is not addressed.

5.2 Computation of partial software cases

This section deals with the computation of a *partial software case*. This activity is also called *slicing* (of a complete software case). Analogously, a partial software case can be referred to as a *slice*.

The term "slicing" stems from the concept of *program slicing* introduced by Weiser in [Wei84]. Given a so-called *slicing criterion* in the shape of a distinguished program statement and a set of variables used in the program, the original program slicing approach intends to find those statements prior to the one specified in the slicing criterion which influence the value of at least one variable in the set. It can be said that program parts which are not needed for computing the values of the variables are "sliced away". This is comparable to the slicing of software cases in ReDSeeDS, where only reuseable parts of the case shall be extracted.

In the context of ReDSeeDS, a slice is computed on the basis of a set of previously selected requirements. Analogous to program slicing, the basis of a slice, i.e. the set of elements of a past software case similar to a set of elements of the current software case, is also called *slicing criterion* henceforth. However, the approach given below allows for slicing criteria to include arbitrary parts of a software case. It is not restricted to requirements only.

The slicing approach is presented incrementally. Section 5.2.1 describes the basic principles of the approach, which solely rely on the existence of traceability links interconnecting the various elements of a software case. Each of the following sections adds a layer of complexity. Section 5.2.2 specifies the treatment of nesting relationships, e.g. how to deal with a class which is a member of a package included in a slice. In section 5.2.3, the different types of traceability links are included in the approach. Section 5.2.4 considers other structures such as generalisation/specialisation relationships which are not traceability links, but which play an important role all the same. Section 5.2.5 gives a summary of the slicing approach. Finally, 5.2.6 describes the usage of *GReQL*, the query language integrated with the prospective repository technology *JGraLab*, for slicing software cases.

5.2.1 Basic principles

Disregarding the type of traceability links, a slice includes all software case elements reachable from the starting criterion by traversing of traceability links. An important constraint is that each boundary between two abstraction levels of the software case may be crossed only once, i.e. at least one path between two arbitrary elements in the slice must contain at most one inter-level traceability link connecting the same two abstraction levels. More precisely, referring to the link types defined in deliverable D3.2 [KSC+07], such a path must contain at most one link of types SCL :: IsAllocatedTo or SCL :: Satisfies and at most one link of type SCL :: Implements. The same is true for links of type Dependencies :: Abstraction originating from the Software Development Specification Language, which includes UML.

Given a set of requirements, this approach is supposed to find elements on other abstraction levels satisfying or implementing these requirements. Without the restriction on the crossing of abstraction level boundaries explained in the last paragraph, other requirements, which are also satisfied or implemented, would be yielded, too. Moreover, the restriction also prevents the slice from getting too large. See figure 5.3 for an illustration: although the black element on the requirements level is reachable from the slicing criterion, it is not included in the slice.

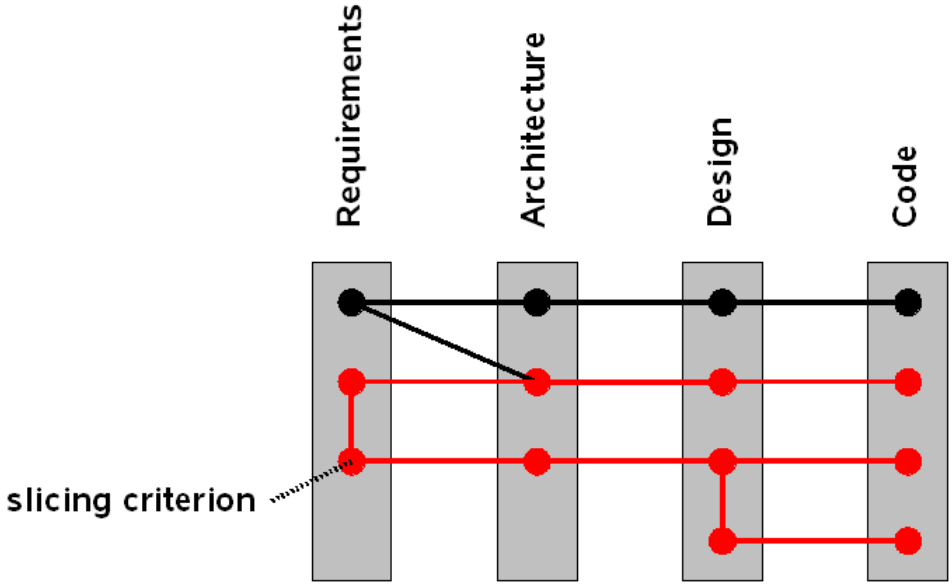


Figure 5.3: Basic principles of the slicing approach: Based on the element on the requirements level serving as slicing criterion, the coloured elements are included in the slice. Elements are represented as dots which are connected by traceability links (lines).

5.2.2 Taking into account nesting relationships

Employing the slicing approach described in section 5.2.1, only elements reachable via traceability links from the slicing criterion are included in the slice. If another element is nested within an element reachable that way, it is not included unless it is reachable by traceability links itself. Therefore, the slicing approach also has to consider such nesting relationships, e.g. packages or classes (which contain attributes and operations). Of course, transitive nesting relationships are eligible, too.

Often, but not always, nested structures are recognisable by the use of compositions in the meta model of the Software Case Language. One exception is, for instance, a Feature in UML, which is part of the SDSL. Features, e.g. operations, are connected to Classes by simple associations, which in fact possess compositional semantics. Another example are SVOSentences: it's Phrases acting as Subject or Predicate is connected via a Hyperlink.

Since a comprehensive enumeration of all elements denoting a nesting relationship is beyond the scope of this document, only some examples taken from the SCL meta model are given in table 5.1. For each element in its left column, the right column indicates possible nested elements which are to be included in the slice.

Language	Element	Possible nested elements
RSL	RequirementsPackage	RequirementsPackage, Requirement
	Requirement	RequirementRepresentation
	SentenceList	*HyperlinkedSentence*
	ConstrainedLanguageScenario	*ConstrainedLanguageSentence*
	DomainElement	*DomainElementRepresentation*
	SVOSentence	Subject, Predicate
SDSL	*Element*	*Element* (in the role of ownedElement)
	Classifier	*Feature*
Java	Block	*Member*
	JavaPackage	JavaPackage, TranslationUnit
	ForStatement	*Statement*

Table 5.1: Examples of nesting relationships to be considered by the slicing approach

5.2.3 Taking into account traceability link types

The slicing approach hitherto described builds a slice by adding every element reachable via traceability links, regardless of the latter's type. However, links of particular types in SCL are not suited for inclusion in the slicing approach due to their semantics. Consequently, they are not to be traversed by the slicing algorithm. Other link types may only be traversed in one specific direction. In the following, such constraints applying to traceability link types in SCL are discussed in detail. Table 5.2 summarises the results. See deliverable D3.2 [KSC+07] for the semantics of the different types.

Inter-level link types

Links of these types, including IsAllocatedTo, Satisfies, Implements and Abstraction connect elements from two different abstraction levels. Their semantics can be summarised by "the target element is an abstraction from the source element" (or vice versa for IsAllocatedTo). Thus, they are traversable in both directions. Note that this is not true for links of one of Abstraction's subtypes, as these denote intra-level links (see below).

Requirements-level link types

The intra-level links on the requirements abstraction level can be separated into two categories: links between arbitrary Requirements and links only used in conjunction with a RSLUseCase.

Links between two arbitrary Requirements:

- ConflictsWith: Links of this type shall not be traversed for building the slice. Actually, the existence of such a link could indicate an inconsistency in the software case.
- Constrains: If the slice already comprises the target Requirement, the source Requirement shall also be included. The source Requirement may impose a restriction for the reusability of elements satisfying or implementing the target Requirement.
- DependsOn: If the slice already comprises the source Requirement, the target Requirement shall also be included. The elements satisfying or implementing the target Requirement may be needed by those satisfying or implementing the source Requirement.
- DerivesFrom: Similar to DependsOn.
- Elaborates: Similar to DependsOn.
- IsSimilarTo: Links of this type are traversable in both directions.
- MakesPossible: If the slice already comprises the target Requirement, the source Requirement shall also be included. The elements satisfying or implementing the source Requirement may be needed by those satisfying or implementing the target Requirement.
- Operationalizes: Similar to MakesPossible.

Links involving at least one RSLUseCase:

- Fulfils: Links of this type are traversable in both directions.
- InvocationRelationship: If the slice already comprises the source RSLUseCase, the target RSLUseCase shall also be included. The elements satisfying or implementing the target RSLUseCase may be needed by those satisfying or implementing the source RSLUseCase.
- Participation: These links between an Actor and a RSLUseCase shall not be traversed for building the slice. An Actor possibly participates in many RSLUseCases which are not related to each other. Incorporating these links would thus lead to very large slices.
- Usage: Similar to Participation.

Abstraction Level	Link type	Possible direction(s) of traversal
(Inter-level)	IsAllocatedTo	both
	Satisfies	both
	Abstraction (without subtypes)	both
	Implements	both
Requirements	ConflictsWith	none
	Constrains	source ← target
	DependsOn	source → target
	DerivesFrom	source → target
	Elaborates	source → target
	IsSimilarTo	both
	MakesPossible	source ← target
	Operationalizes	source ← target
	Fulfils	both
	InvocationRelationship	source → target
	Participation	none
	Usage	none
Architecture & Design	Dependency (without Abstractions)	client → supplier
	Realization	client → supplier

Table 5.2: Traceability link types considered in slicing approach

Architecture- and design-level link types

- Dependency: If the slice already comprises the client *NamedElement*, the supplier *NamedElement* shall also be included. The supplier *NamedElement* is needed by the client *NamedElement*. Note that this is not true for links of Dependency's Abstraction subtype, as these denote inter-level links (see above).
- Realization: If the slice already comprises the client *NamedElement*, the supplier *NamedElement* shall also be included. The supplier *NamedElement* is implemented by the client *NamedElement*.

5.2.4 Taking into account other structures

Besides traceability links and nesting relationships, there are other structures which have to be taken into account by the slicing approach. For instance, a Generalization relationship between two *Classifier*s denotes that the specialised classifier inherits the properties of the generalised classifier. Thus, if a slice comprises the former, the latter has to be included, too.

As with nesting relationships, a comprehensive overview would be out of scope in this document. Therefore, table 5.3 only contains some examples.

Abstraction Level	Structure	Possible direction(s) of traversal
Architecture & Design	Generalisation	specific → general
	PackageImport	importingNamespace → importedPackage
	PackageMerge	receivingPackage → mergedPackage

Table 5.3: Other structures considered in slicing approach

Another structure to be considered is the caller/callee relationship between two methods on the code level. As yet, the Java part of the SCL meta model does not offer the means to express such a relationship.

5.2.5 Summary of the slicing approach

This section summarises the various aspects of the slicing approach presented in sections 5.2.1 to 5.2.4.

First, a *slicing criterion* in the shape of elements of a software case has to be selected. It constitutes the basis for computing the slice, i.e. the slice will contain all elements of a software case which are related to the slicing criterion. In ReDSeeDS, the slicing criterion will be a set of elements from the requirements level of a past software case which is similar to a requirements specification of a current software case.

Starting from the slicing criterion, the slice contains all elements reachable by traversing

- traceability links as given in table 5.2,
- nesting relationships between elements exemplified in table 5.1, and
- other relationships as exemplified in table 5.3

Furthermore, the restriction on traversing a specific inter-level traceability link only once has to be obeyed (see also section 5.2.1).

5.2.6 Computation using GReQL

The slicing approach described in sections 5.2.1 to 5.2.5 operates on the abstract syntax graphs of software cases and suggests traversal as a means of accessing their elements.

In the course of the ReDSeeDS project, JGraLab seemed to emerge as the technology of choice for the fact repository containing past software cases (see also [BER+07]). It is capable of storing their abstract syntax graphs as TGraphs [EF95] and offers the means to traverse them as needed by the slicing approach. An equally powerful, but more concise way of retrieving information from the repository is *querying*. Querying can be performed by employing the *Graph Repository Query Language* (*GReQL*, which is closely integrated with JGraLab. As an example, consider the following query realising the retrieval of all elements of a software case connected to a slicing criterion via traceability links:

```
from
    sc : V{Requirement}, v : V{SCLElement}
with
    sc.uid = "Req42" and sc <->{TraceabilityLink}* v
report
    v
end
```

This simple GReQL-query returns all vertices `v` of type SCLElement which are connected to the vertex `sc` of type Requirement with the attribute `uid` set to `Req42`, constituting the slicing criterion.

5.3 Sample computation of a partial software case

This section describes an example how to slice a partial software case out of an existing software case. This example arose from a case study from the mobile phones domain: GoPhone – *A Software Product Line in the Mobile Phone Domain*[1]. Based on a software case for the mobile phone product line "GoPhone Elegance", which is an advanced device, a further software case is sliced out of it. The "GoPhone Elegance" mobile phone is able to receive and display messages with multiple content (text, pictures, sound). For the partial software case it is assumed that a requirement stating that a user must be able to set the recipient of a SMS message shall be reused. Thus, this requirement serves as slicing criterion.

Table 5.4 shows an excerpt of the software case for the "GoPhone Elegance" for the messaging part only. Table 5.5 describes the partial software case for the "SetSMSRecipient" functionality.

[1] http://www.software-kompetenz.de/?21618

ABSTRACTION LAYER	CONTENTS
Requirements	... **RequirementsPackage** Messaging **Requirement** HandleEMails: The system must be able to handle e-mails. **Requirement** HandleSMSMessages: The system must be able to handle SMS-messages. **Requirement** HandleMMSMessages: The system must be able to handle MMS-messages. ... **Requirement** SetSMSRecipient: The user must be able to set the recipient of a SMS-message. ...
Architecture	No architectural relevant information for 'Messaging'
Design	see picture 5.4
Code	see Listing 5.1

Table 5.4: Excerpt of the software case for the product "GoPhone Elegance".

Consider that those issues related to multimedia-based message content are not adopted in the new software case.

Figure 5.5 shows an excerpt of the abstract syntax graph for the messaging part of the "GoPhone Elegance" software case, including the partial software case which is sliced from it. Starting from the slicing criterion illustrated by the Requirement with the thick border, the application of the approach described in section 5.2 results in the partial software case indicated by the cloloured elements.

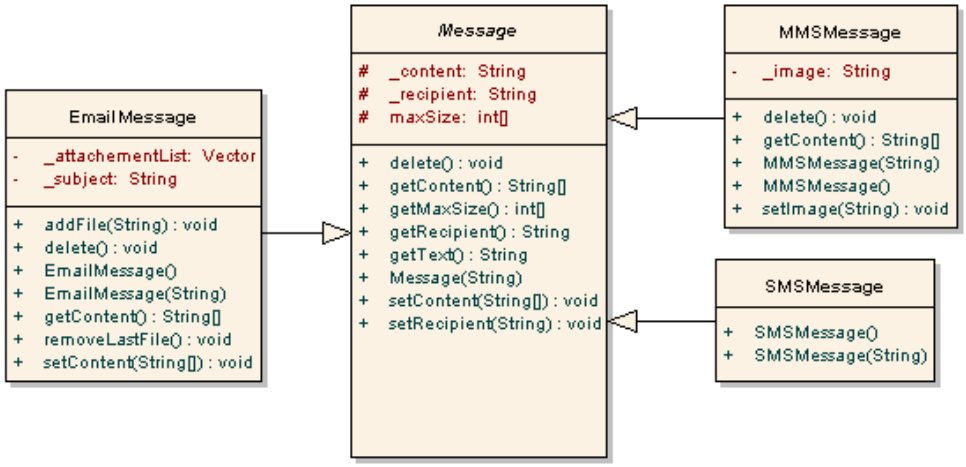


Figure 5.4: Design of the GoPhone Elegance messaging functionality

Listing 5.1: Code for the messaging functionality of “GoPhone Elegance”

```
 public abstract class Message {

        public final static String SMS = "SMS";
        public final static String MMS = "MMS";
        public final static String EMAIL = "EMS";
        protected String _recipient = "";
        protected String _content = "";
        protected String msgType = null;
        protected int[] maxSize = {0, 0, 0};

        public Message(String msg) {
                for (int i = 0; i <= msg.length(); i++) {
                        if (msg.charAt(i) == 'R') {
                                int recLength = Integer.parseInt(msg.substring(0, i));
                                i++;
                                _recipient = msg.substring(i, (i + recLength));
                                msg = msg.substring(i + recLength);
                                break;
                        }
                }
                for (int i = 0; i <= msg.length(); i++) {
                        if (msg.charAt(i) == 'C') {
                                int conLength = Integer.parseInt(msg.substring(0, i));
                                i++;
                                _content = msg.substring(i, (i + conLength));
                                break;
                        }
                }
        }

        public String getRecipient() {
                return _recipient;
        }

        [...]
}

public class SMSMessage extends Message {
        public SMSMessage() {
                super();
                msgType = Message.SMS;
                maxSize[0] = 25;
                maxSize[1] = 160;
        }

        [...]
}

public class MMSMessage extends Message {
        // Name of the image file.
        private String _image = "";

        public MMSMessage() {
                super();
                msgType = Message.MMS;
                maxSize[0] = 25;
                maxSize[1] = 500;
        }

        [...]
}

public class EMailMessage extends Message {

        private String _subject = "";
        private Vector _attachementList = new Vector();

        public EMailMessage() {
                super();
                msgType = Message.EMAIL;
                maxSize[0] = 35;
                maxSize[1] = 70;
                maxSize[2] = 1000;
        }

        [...]
}
```

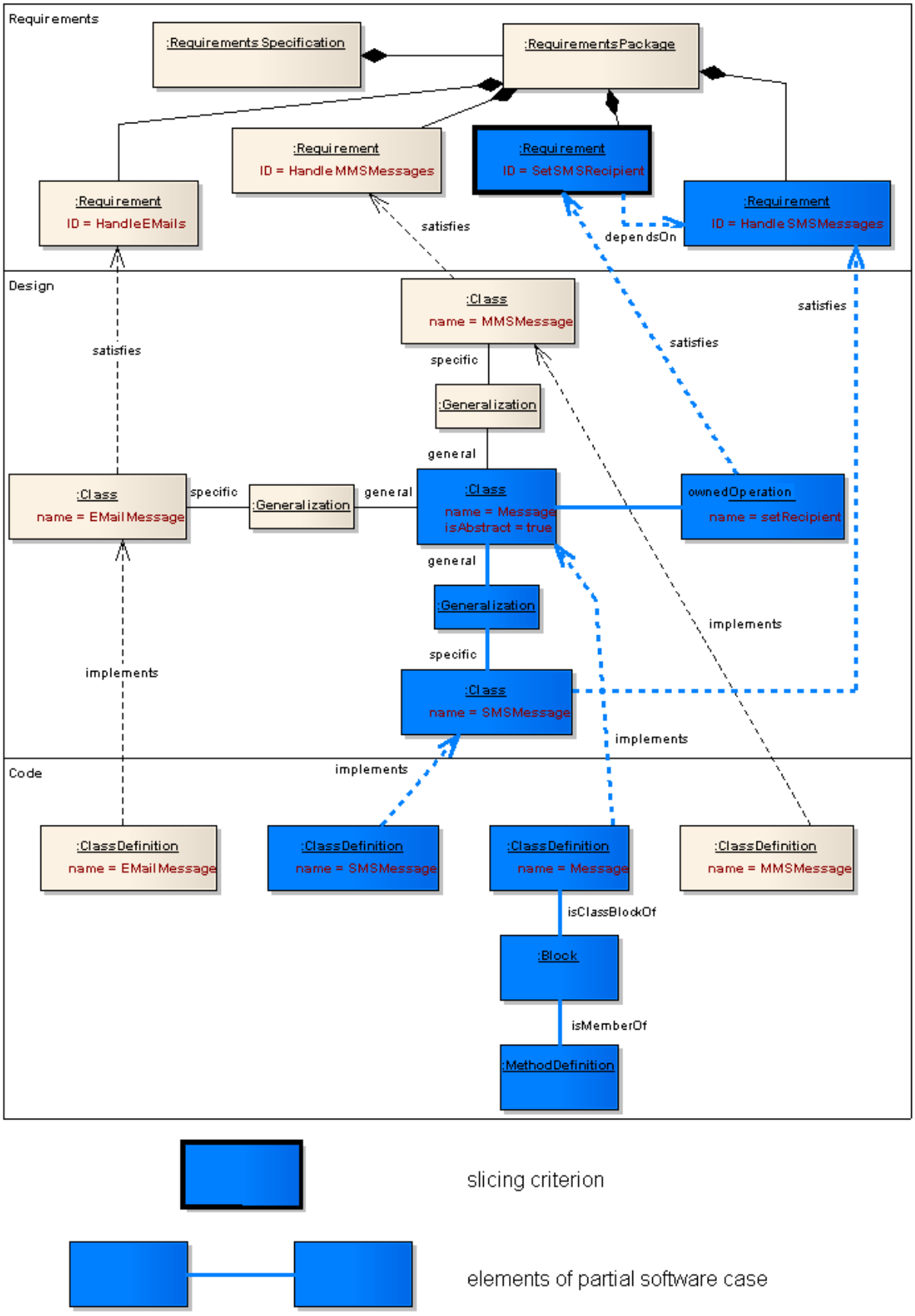


Figure 5.5: Abstract syntax graph for the “GoPhone Elegance” software case excerpt in 5.4 including a partial software case. For the sake of clarity, some elements, e.g. RequirementRepresentations, Operations, etc. are omitted.

ABSTRACTION LAYER	CONTENTS
Requirements	**RequirementsPackage** Messaging **Requirement** HandleSMSMessages: The system must be able to handle SMS-messages. **Requirement** SetSMSRecipient: The user must be able to set the recipient of a SMS-message.
Architecture	No architectural relevant information for 'Messaging'
Design	see picture 5.6
Code	see Listing 5.2

Table 5.5: Partial software case on the basis of the Requirement "SetSMSRecipient"

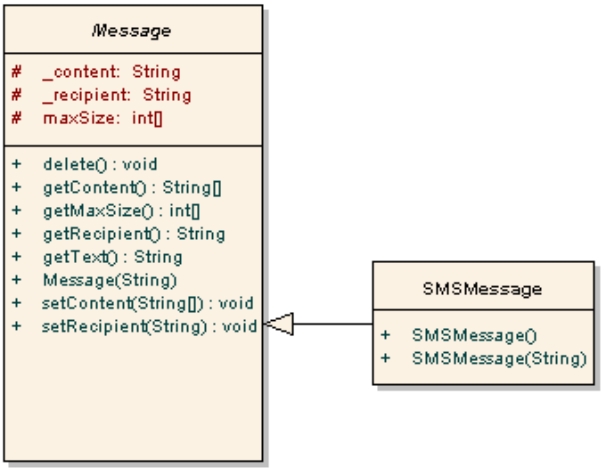


Figure 5.6: Design part of the partial software case on the basis of the Requirement "SetSMSRecipient"

Listing 5.2: Reused Code

```
1  public abstract class Message {
2
3          protected String _recipient = "";
4          protected String _content = "";
5          protected int[] maxSize = {0, 0, 0};
6
7          public Message(String msg) {
8                  for (int i = 0; i <= msg.length(); i++) {
9                          if (msg.charAt(i) == 'R') {
10                                 int recLength = Integer.parseInt(msg.substring(0, i));
11                                 i++;
12                                 _recipient = msg.substring(i, (i + recLength));
13                                 msg = msg.substring(i + recLength);
14                                 break;
15                         }
16                 }
17                 for (int i = 0; i <= msg.length(); i++) {
18                         if (msg.charAt(i) == 'C') {
19                                 int conLength = Integer.parseInt(msg.substring(0, i));
20                                 i++;
21                                 _content = msg.substring(i, (i + conLength));
22                                 break;
23                         }
24                 }
25         }
26
27         public String getRecipient() {
28                 return _recipient;
29         }
30
31         [...]
32 }
```

```
33
34 public class SMSMessage extends Message {
35         public SMSMessage() {
36                 super();
37                 msgType = Message.SMS;
38                 maxSize[0] = 25;
39                 maxSize[1] = 160;
40         }
41
42         [...]
43 }
```

Chapter 6

Visualisation of differences

Following the identification of similar requirements in a past software case, these results must be presented to the ReDSeeDS user. More precisely, the requirements specification of the current software and its counterpart in the past software case have to be juxtaposed in order to allow the user to check the outcome of the automatic similarity mapping. Then he could decide on manually discarding requirements from the computed set or on including requirements not yet taken into account.

Based on the final set of requirements selected for reuse, a partial software case has to be computed and its contained elements and relationships shown to the user.

After an introduction to existing visualisation approaches in section 6.1, section 6.2 describes how to visualise differences between requirements. Section 6.3 deals with the visualisation of partial software cases.

6.1 Known approaches

Similarity calculation, its algorithms and visualisation approaches are known at least since the Unix diff-tools in the early 1970s. Since then, various other approaches were published. This section gives an overview about different visualisation strategies rather than about concrete algorithms. Besides the strategy used by the diff tool and its successors to display the difference as a sequence of edit operations there are also strategies which display the difference in a format more convenient for the user.

6.1.1 Integrated parallel visualisation

One of those approaches is called (integrated) parallel visualisation. Both documents are displayed in a separate view and the differences and similarities are highlighted using different colours. Figure 6.1 shows a small example. The classes named Class1 and Class3 are part of both models while class Class2 is only part of the first model and Class4 is only part of the second model. The colours green and red are used to visualise the elements that are deleted in model two or that are added in model two relative to model one. Another variant of the integrated parallel visualisation is the tabular view. Figure 6.2 shows a small example of the tabular view of the two models depicted in figure 6.1.

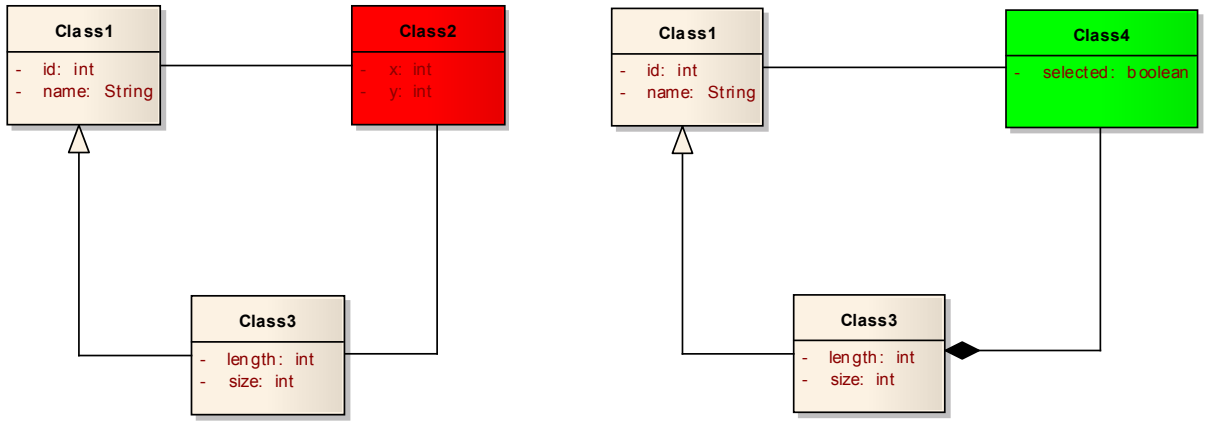


Figure 6.1: Integrated parallel visualisation

Model 1	Model 2
Class1	Class1
id: int	id: int
name: String	name: String
Class2	
x: int	
y: int	
Class1	Class1
id: int	id: int
name: String	name: String
	Class4
	selected: boolean

Figure 6.2: Integrated parallel visualisation in tabular view

6.1.2 List of local differences

A list of local differences can be seen as a mixture of the difference or similarity visualisation using sequences of edit operations and integrated parallel visualisation. Whenever integrated parallel visualisation is not possible or not appropriate for a task, the differences between two models can be visualised using a list of local differences. The list should contain a simple, small, and human-understandable description of the differences which could be supported using small cutouts of the original model. The table in figure 6.3 shows a tabular view of local differences for the example already used above.

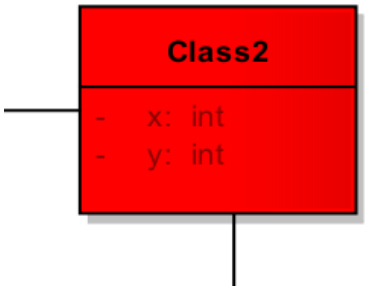
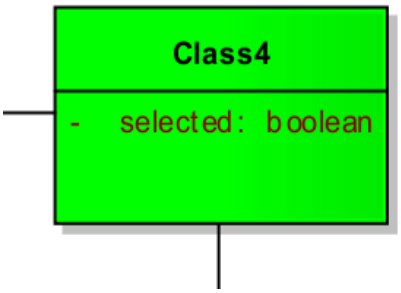
Description	Cutout
Class2 removed	
Class4 added	

Figure 6.3: Local difference visualisation in tabular view

6.1.3 Visualisation in a common model

Besides the approaches depicted above, it is also possible to show both models in a common, unified form. Both models are presented as one merged model, differences are highlighted by different colours, for instance red for all elements that are only part of model one and green for all elements that are only part of model two. Figure 6.4 shows an example of a unified view of the two models already depicted above. In contrast to the other two visualisation strategies depicted above, this view needs a very sophisticated merging algorithm, hence it is barely used in practise.

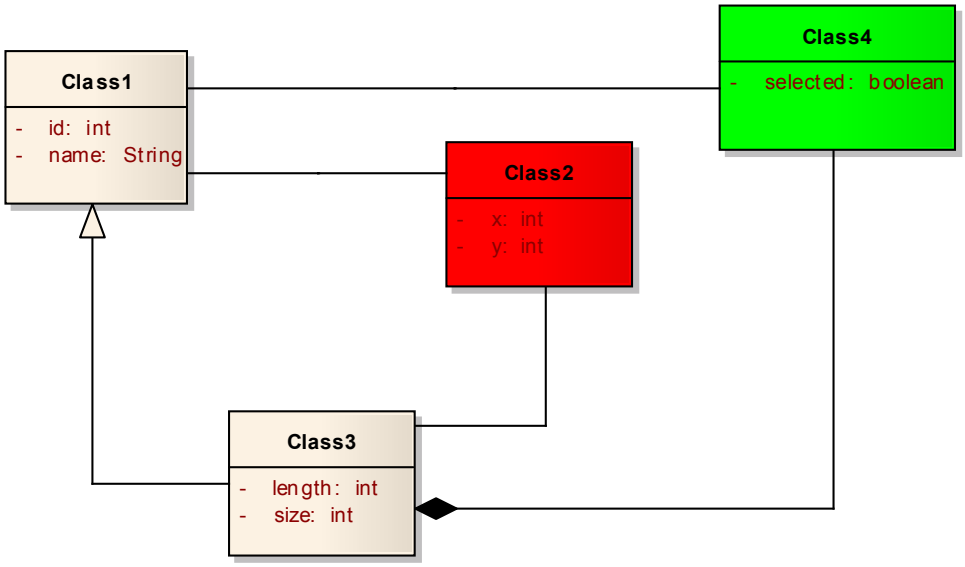


Figure 6.4: Common model view

6.1.4 Ontology difference visualisation

As stated in Section 3.2 ontology alignment / mapping tools can be applied to determine the mapping information needed for solution marking. Visualisation of ontologies is an active research field [FSH02, Ker06, Ala03, SNM+02]. However, according to [LS06], only few ontology alignment tools provide a visualisation of their results: OLA [ELTV04], PromptViz [NM03] and their own tool AlViz. The authors point out that the list of matched element pairs provided by most tools is not sufficient for human understanding.

The OLA (OWL Lite Alignment) tool uses a graph-based approach for visualising ontologies. The tool displays two aligned ontologies side by side in order to allow for the comparison of their structures. Thus, this tool uses the integrated parallel visualisation described in Section 6.1.1.

PromptViz is a plug-in for Protégé and visualises differences between different versions of ontologies. The plug-in uses treemaps in order to provide an overview of large ontologies (see Figure 6.5 for an example). Colour is used to mark added, deleted, changed and relocated items.

Like PromptViz, AlViz is a plug-in for Protégé. AlViz provides two types of views for comparing ontologies: J-Trees display the hierarchical structure and small world graphs provide clustering of nodes according to the selected level of detail (see Figure 6.6 for an example). Colour is used in both types of views to represent the similarity of concepts.

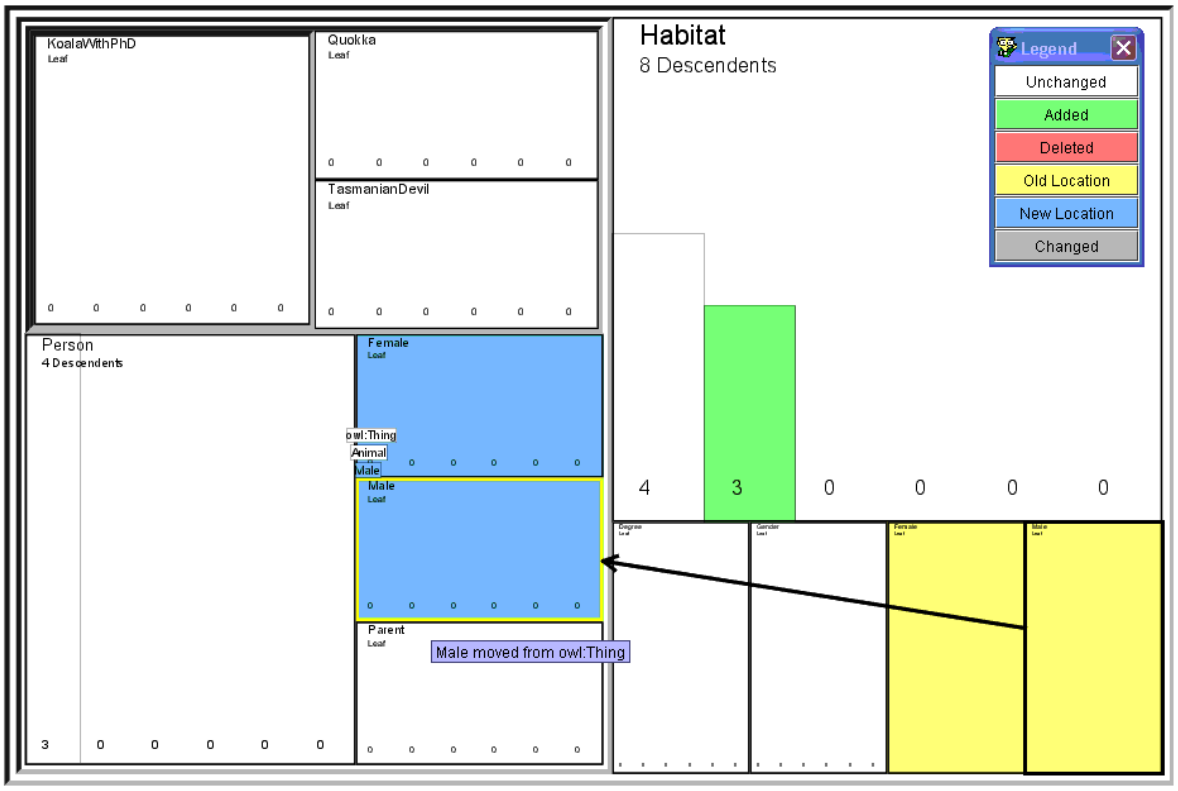


Figure 6.5: PromptViz shows differences between ontologies using treemaps (taken from: http://webhome.cs.uvic.ca/c̆hisel/imgs/promptviz/koala_screenshot.gif).

6.2 Visualisation of differences between requirements

This section describes the proposed variants for displaying differences between elements of RSL-compliant requirements specifications. Differences can be examined on different levels of abstraction. Section 6.2.1 shows how few past cases can be compared regarding their similarity with the query. Then, Sections 6.2.2 and 6.2.3 describe how the use cases of the current case can be compared with use cases from one selected past case. Finally, the comparison on the level of ConstrainedLanguageScenarios is shown in Section 6.2.4. For details on how requirements themselves are visualised see section 4.3. Visualisation of elements dependent on requirements is described in section 6.3.

6.2.1 Comparing similarity of past cases with a query

The result of a query is a list of past cases ordered by similarity values. This overall similarity value is not sufficient for selecting the most appropriate past case(s). Figure 6.7 shows a diagram that allows the ReDSeeDS user to compare few past cases.

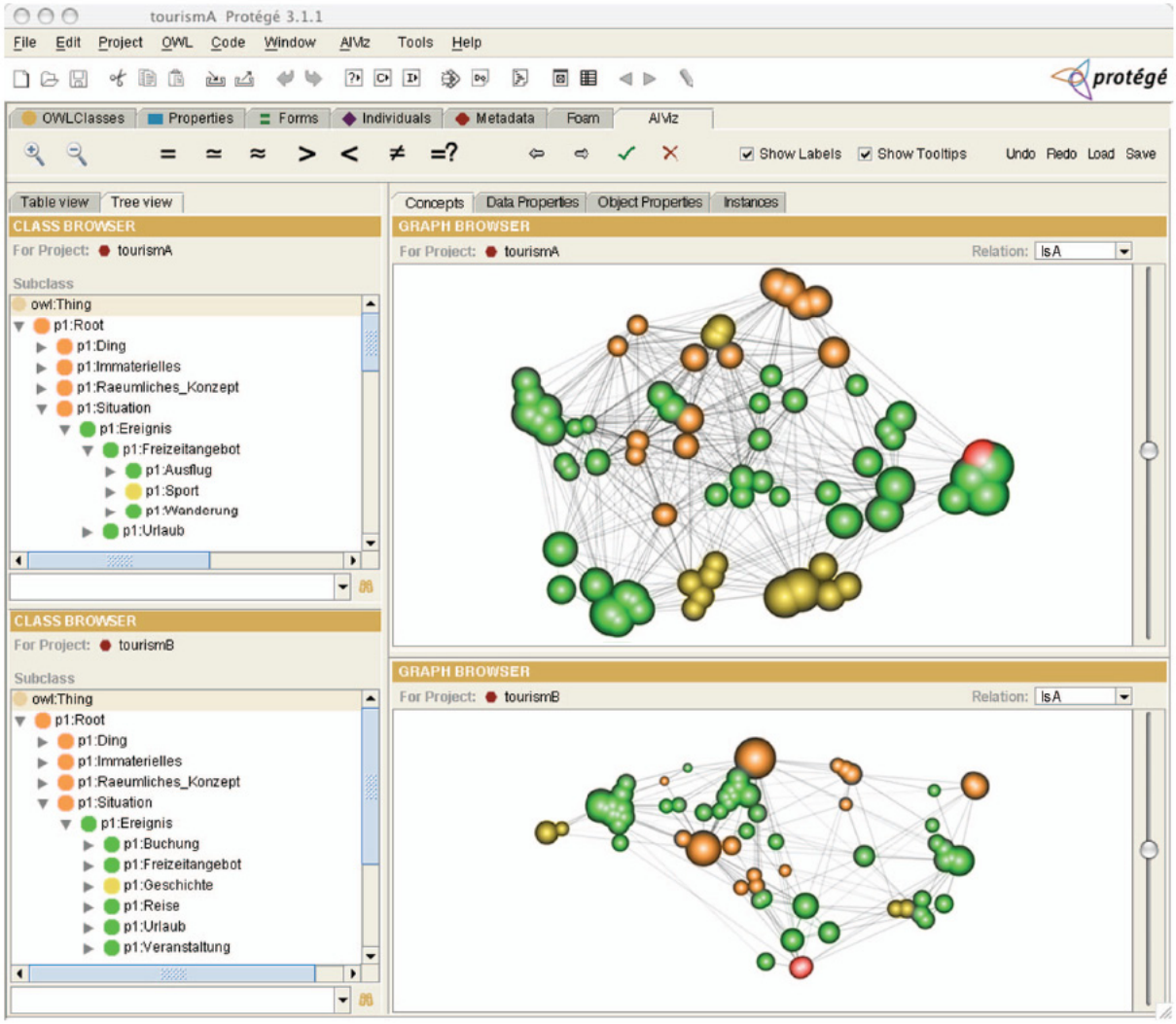


Figure 6.6: AlViz provides two different views on each ontology (taken from: [LS06]).

The diagram visualises to which query elements a selected past case is similar and to which not. Examining the diagram, the ReDSeeDS user can for instance easily recognise if two past cases together cover the query elements better than any single case.

6.2.2 Differences between use cases

This section describes proposed variants for displaying differences between a "current" software case (currently edited by a user and used in a query to the repository in a "WHERE" clause - see [BEK+07], p.12) and one retrieved from the software case repository (a "past" one).

During the comparison only use cases, as highly structured and precisely defined requirements, are shown. The similarity display for a use case model presents the current model in form of a

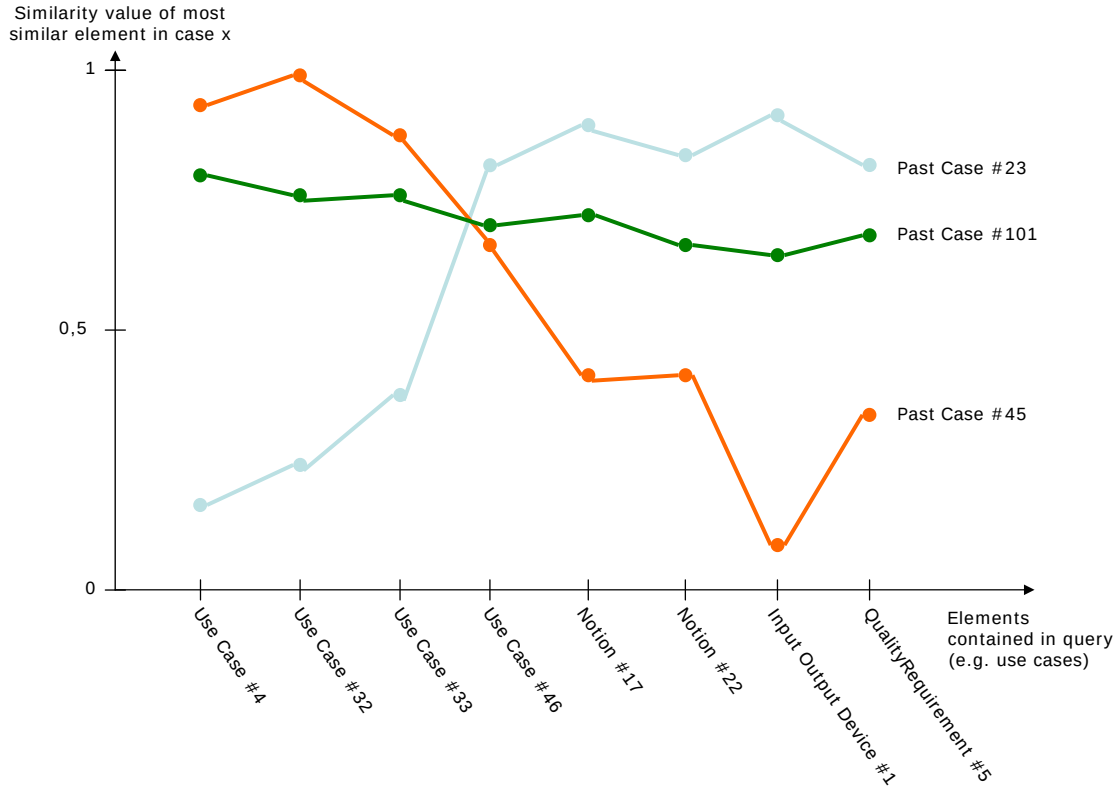


Figure 6.7: Data structure for the mapping information between two (partial) RSL requirements models.

tree with use cases contained in the past software cases attached to the corresponding (in terms of similarity) "current" use cases (see Figure 6.8). Only a subset of "past" use cases matching "current" use cases with a score above the similarity threshold set during the definition of a query is displayed (see [BEK+07], p.14). The similarity score is displayed next to an every "past" use case in form of a partially filled horizontal bar (fill-ratio depends on the value of the similarity score) along with a numerical expression of the similarity.

6.2.3 Differences between use cases visualised by scenarios

This section describes how differences between one use case which is represented by an activity diagram and another one which is represented by a couple of sequence diagrams can be visualised. To make it more understandable, the fitness club example is used. The left part of Figure 6.9 illustrates the simplified scenarios of the "Make Reservation" use case of the "past" design. It is described by an activity diagram which outlines the steps for making a reservation. The decision point is used to model different activities depending on the selection of the user. The right part of Figure 6.9 illustrates the scenarios of the "Make Reservation" use case of the

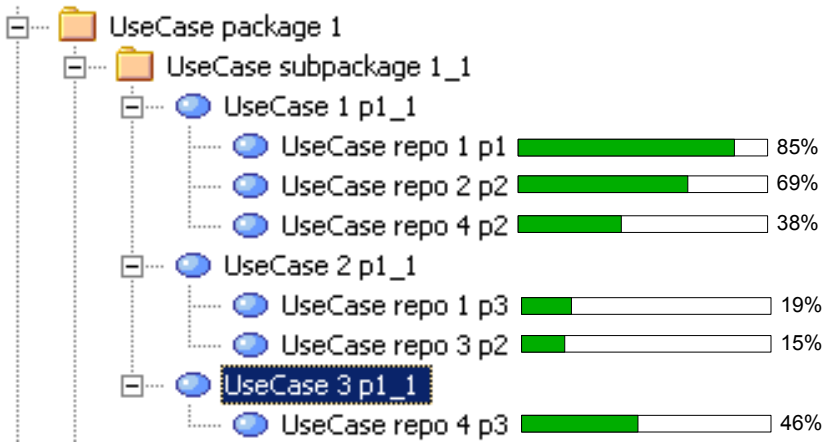


Figure 6.8: UseCase TreeView with similarity indicators. The uses cases containing "repo" in their name belong to the past software case rom the repository, all other ones come from the current software case.

"current" design. Two Sequence diagrams are used to describe the scenarios. The upper one is equal to the path of the activity diagram for selecting a course. The lower one is equal to the path of the activity diagram for selecting a court. The part of the activity diagram which is not covered by the sequence diagrams is marked red. It is also possible that sequence diagrams exist which are not covered by the activity diagram. In that case, which is not shown in Figure 6.9, the additional sequence diagram can be marked green. It should be clarified that this approach is only an optional visualisation type.

6.2.4 Differences between ConstrainedLanguageScenarios

When one of the "past" use cases attached to a "current" use case is selected, a difference between details of the "current" and the chosen "past" use case is displayed. Only textual scenario representation (ConstrainedLanguageScenario – see [KSS+07]) is used when showing details of the selected use cases' similarity. For that, coloured diff-like notation is used (see Figure 6.10). The meaning of text-background colours used in this notation is as follows:

- no colouring – information is the same
- yellow colouring – information differs in this part
- red colouring – a sentence or part of a sentence is missing
- green colouring – added a sentence or part of a sentence

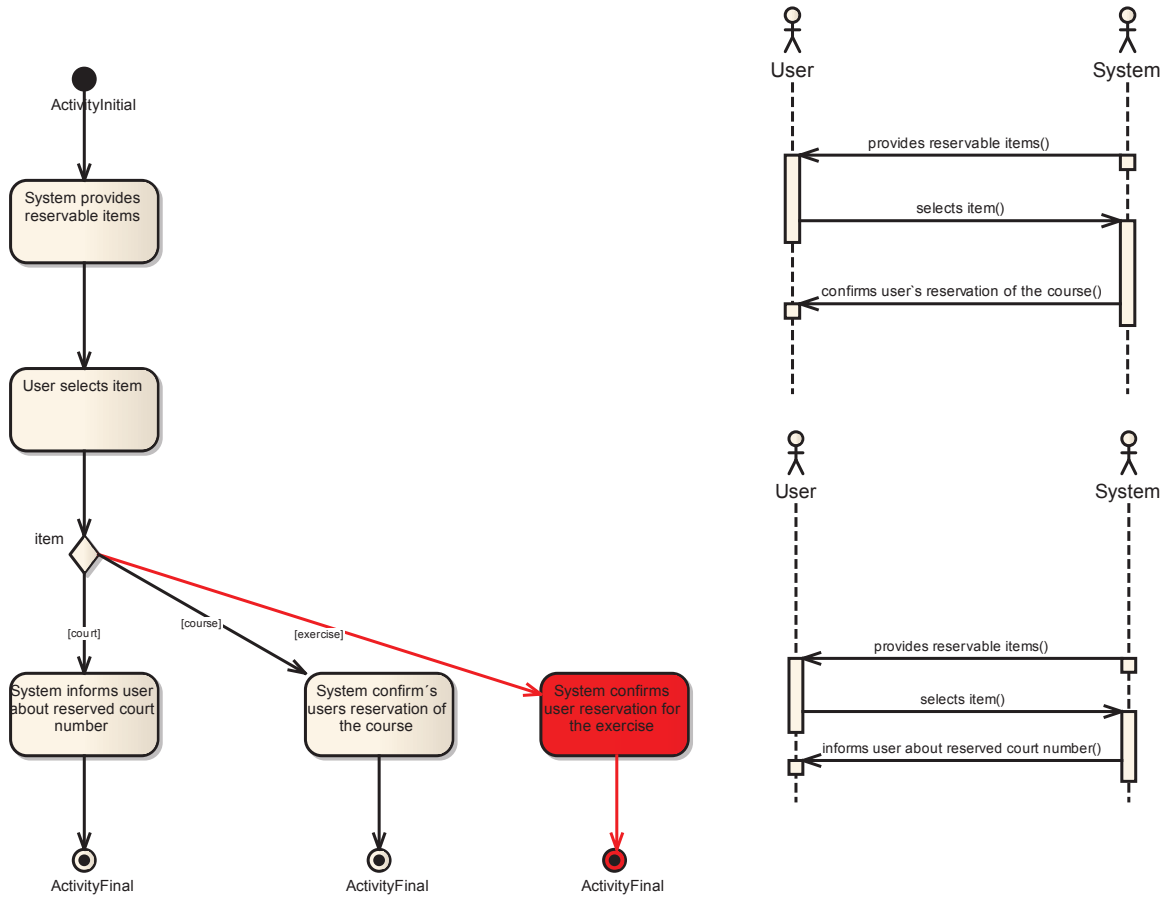


Figure 6.9: Illustration of differences between use cases visualised by scenarios

Please note that any of the “current” scenarios can be compared to any of the “past” ones for two selected use cases and displayed using the coloured diff.

To present requirements differences depicted in Figures 6.8 and 6.10, local similarities information is used (see equation 6.1 in [WKB+07]), namely a group of local similarities for requirements.

Figure 6.10: Example of coloured diff-like display of similarities of ConstrainedLanguageScenario instances for two compared use cases

6.2.5 Differences between requirements

Differences for other requirements types may be visualised in a way similar to the approach presented above, as long as requirements are represented by descriptive representation (textual comparison is used then). Sole textual representation differences are depicted using diff-like display (see sections 6.2.2 and 6.2.4 for details).

6.3 Visualisation of partial software cases

Chapter 5 introduced the notion of partial software cases and a mechanism for slicing a partial software case from a complete software case based on the set of selected requirements. A partial software case is composed of a set of selected requirements together with elements originating from lower layers (architecture, design and code) connected by intra- and inter-layer traceability links. Intra-layer traceability links are a set of elements of the requirement layer, which selected requirements depend on (for example vocabulary elements used inside of selected requirement representation). Inter-layer dependencies are elements of lower layer models which are connected with the selected requirements through traceability links.

Figure 6.11 presents all model layers of a complete software case. Arrows represent inter-layer dependencies of particular model elements. A partial software case is a subset of these elements interconnected through inter- and intra-layer dependencies. Such an amount of information should be visualised in a suitable form, thus giving the ability to navigate through elements of a computed partial software case.

6.3.1 Information which should be visualised

As it was stated earlier that a partial software case is defined through software case model elements and dependencies between them. In order to visualise a partial software case, we have to show all elements of a computed slice together with the dependencies constituting a slice. Such a visualisation should show a slice in the context of the complete software case.

Slice visualisation should allow for browsing and navigating through the elements constituting a slice:

- selected requirements

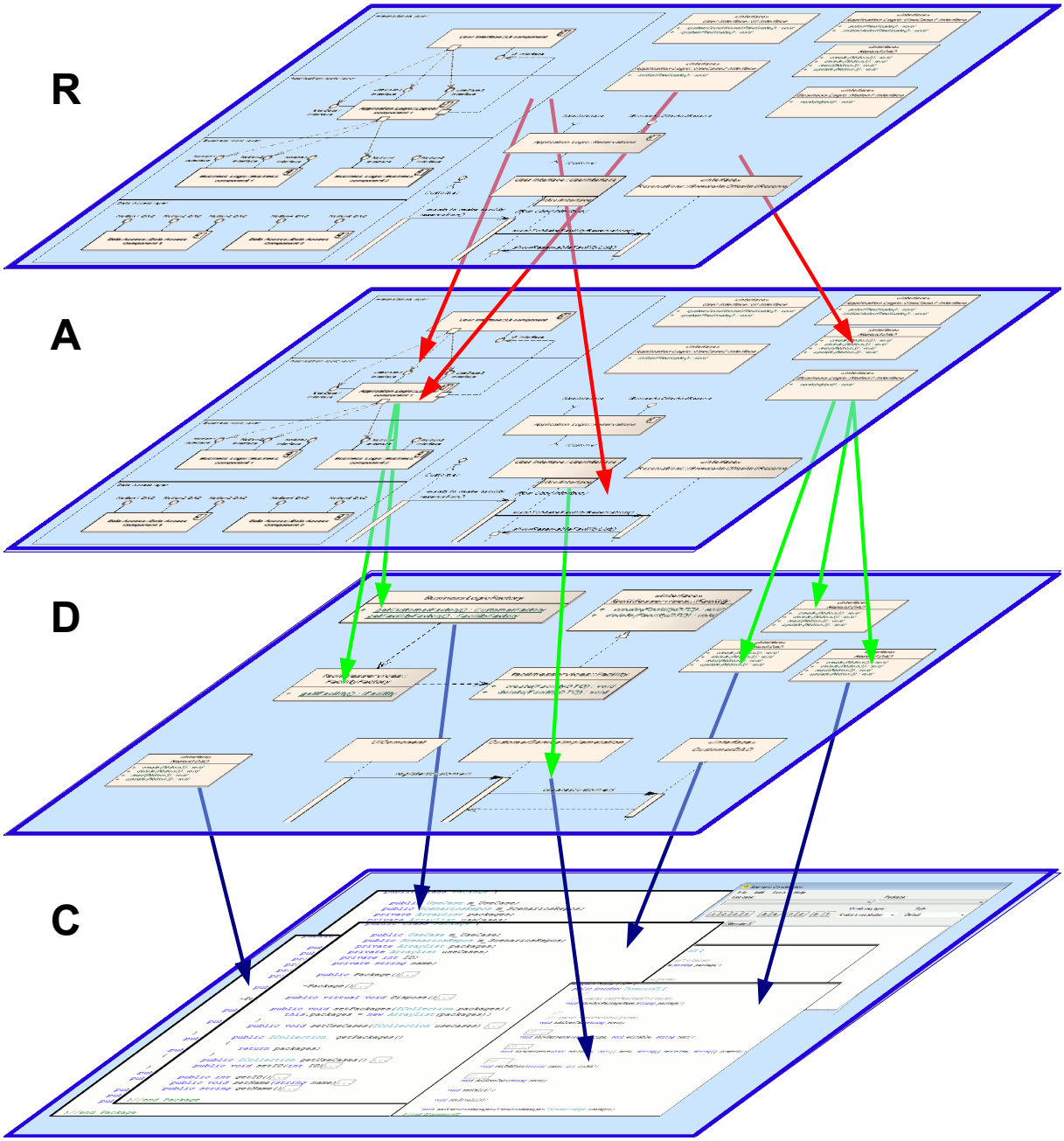


Figure 6.11: Inter-layer dependencies

- nested elements
- intra- and inter-layer dependencies of selected element
- other related elements not included directly in computed slice

Selected requirements, nested elements and intra- and inter-layer dependencies of selected requirements constitute a minimum slice (computed partial software case). Additionally, direct dependencies of slice elements which are not mandatory but can be useful for reuse should be emphasised. Such elements, which are not included in a computed partial software case, could be manually attached to a slice by the reuse engineer. A minimum slice completed with all directly related elements constitutes a maximum slice. Slice visualisation should visualise all elements of a minimum slice in the context of a complete software case. Additionally, it should emphasise all potential slice elements which could be attached manually to a slice (maximum slice).

6.3.2 Additional information useful for analysing partial software case

To emphasise the level of complexity of a selected slice, we can support the presented information with the number of elements and dependencies and other suitable statistics.:

- number of nested elements
- number of directly related elements
- number of indirectly related elements
- other statistics

Such additional information should be helpful for estimating the complexity of a partial software case selected for possible reuse.

6.3.3 Idea of 4-tree view

According to previous sections, the main expectation for visualisation of a software case slice (partial software case) is the possibility of viewing the whole slice as a set of related requirements specification items, architecture items, detailed design items and code items (R-A-D-C).

These items can be packages, subpackages and package elements. User of the SC browser, which allows to navigate through the whole SC, should have an opportunity to easily navigate through the all SC artefacts in order to view a computed slice (elements of the slice together with dependencies constituting it) in the context of a complete SC. Taking into account above features, the form of a SC slice visualisation should enable presenting traced elements:

- in the context of all SC elements,
- in the perspective of complete requirements specification, architecture, detailed design and code,
- in nested structures (packages, subpackages),
- with trace/relation type,
- with other information which can be helpful for estimating the complexity of partial software case selected for possible reuse.

To realise all postulates above, the "4-tree View" was introduced as a possible way for slice visualisation. It mixes the tree form for presenting models' structure with a marking style used for depicting elements specified and linked by traces. The proposed "4-tree View" visualisation is composed of two main parts:

- Artefacts Browser,
- Properties View.

The artefacts Browser is a space where the structure of each (R, A, D, C) artefact can be presented in form of a tree. The user may select artefacts to be shown. There is also the possibility to set an order and a layout of these views. Elements of the slice computed for some requirements item (starting point), as well as other elements which are directly related to slice elements, are distinguished on each lower layer tree view. The type of the element distinction (e.g. different colours) denotes if an element is included in a minimum slice or in a maximum slice (see section 6.3.1). Packages which contain traced elements in any level of nesting should also be marked.

Properties View is a text field containing basic information about the place of the selected element in the SC slice, as well as some additional information which might be helpful in browsing the slice. All this information can be presented in the form of a table. Such a table can contain the following information:

Figure 6.12: Slice Visualisation legend

- basic properties
  - source of traceability link (minimum slice)/other dependency (maximum slice) for a selected element with its type,
  - list of targets of traceability links (minimum slice)/other dependencies (maximum slice) for selected elements with their types,
- additional properties
  - number of nested elements,
  - number of traced elements constituting a minimum slice,
  - number of related elements constituting a maximum slice,
  - other statistics.

User should be allowed to select appropriate properties and to set their order.

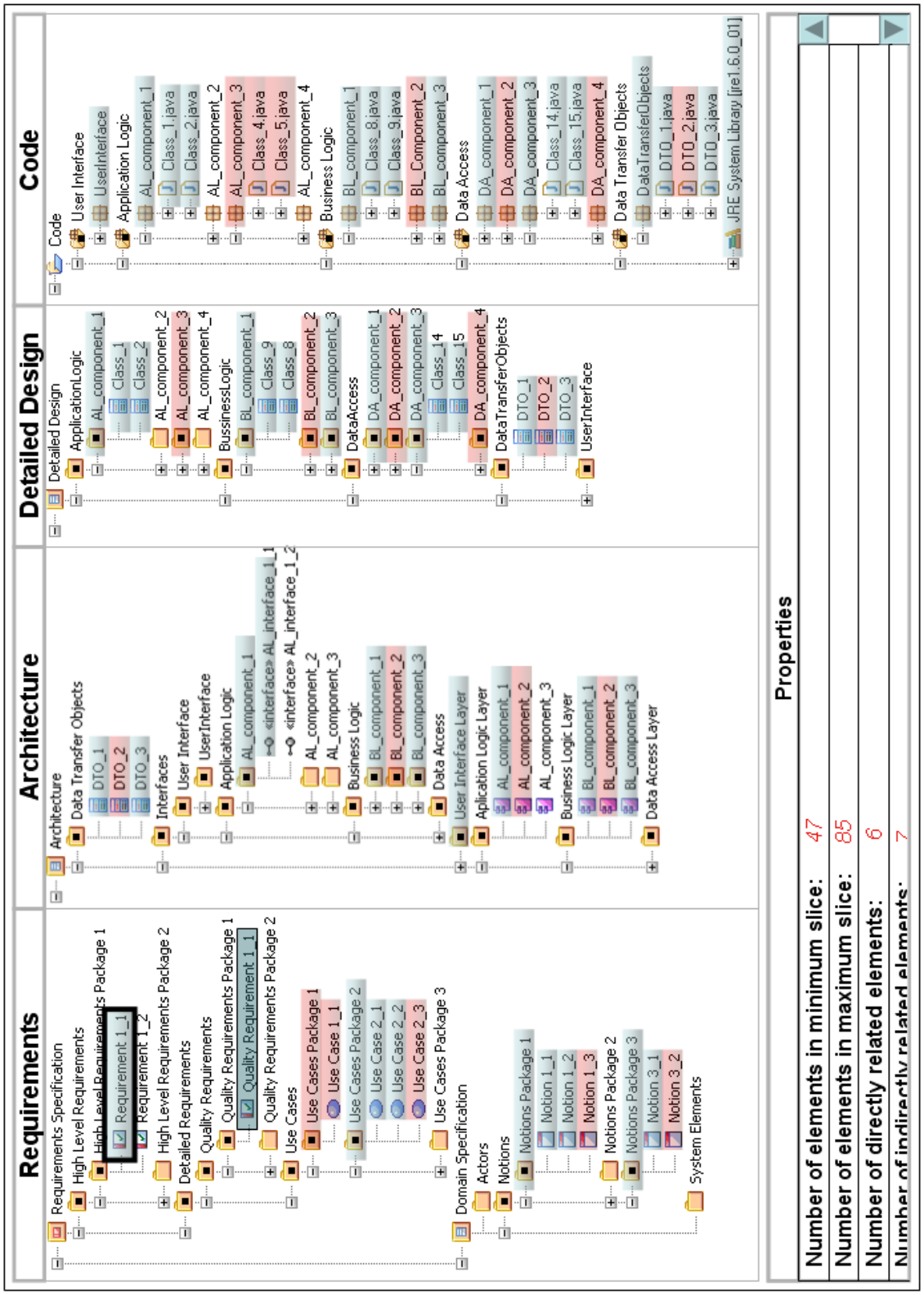


Figure 6.13: Slice Visualisation

The concept of SC slice visualisation is presented in figures 6.12 and 6.13. The main part of figure 6.13 presents the artefacts browser. The starting point of the slice is depicted by a rectangle with thick borderline filled with blue colour. All elements which constitute a minimum slice are marked with a blue rectangle without borderline. A red rectangle without borderline is used for depicting elements which are included in a maximum slice.
Packages which are marked with blue or red rectangles are parts of the slice without their content. A black dot on a package icon means that this particular package contains some packages or child elements which are part of a slice, but the package itself is not.
The text field on the bottom of figure 6.13 is the properties view. This view is a combined list of basic and additional properties listed above. This information describes an element of the slice selected in the artefacts browser. The selected element is depicted by a rectangle with thin borderline filled with dark blue or red colour.

Chapter 7

Conclusion

This document has described the strategies and mechanisms that are used to mark the reusable parts of the software cases stored in the fact repository.

To describe the whole solution marking mechanism, the process is divided into several steps. The starting point for the process is the similarity mapping which is a result of the algorithms defined in deliverable D4.2 “Software Case Similarity Measure” [WKB+07]. The mapping and its relevance for the solution marking mechanism is described in chapter 3.

The next step in the process is the selection of the requirements set to reuse. The importance of this step as well as the automatic selection and the interaction with the user is described in chapter 4.

Chapter 5 depicts the notion of a partial software case. This is that part of a case that is calculated on base of the requirements selected in the previous step. The calculation of this partial software case, which is also called a slice of a case, is depicted in theory as well as with a small example in this chapter.

As the last step in the solution marking process, the slice is visualised in a human-understandable way. Existing approaches for such kind of visualisation and a proposal for a visualisation in the ReDSeeDS engine are depicted in chapter 6.

All things considered, this document provides essential concepts for the development of the ReDSeeDS engine which is to be developed in workpackage 5 of the ReDSeeDS project.

Bibliography

[Ala03] Harith Alani. TGVizTab: An ontology visualisation extension for Protégé. In *Proceedings Visualizing Information in Knowledge Engineering (VIKE'03)*, 2003.

[BEK+07] Daniel Bildhauer, Jürgen Ebert, Thorsten Krebs, Markus Nick, Sören Schneickert, and Hannes Schwarz. Software case query language definition. Project Deliverable D4.1.1, ReDSeeDS Project, 2007. www.redseeds.eu.

[BER+07] Daniel Bildhauer, Jürgen Ebert, Volker Riediger, Thorsten Krebs, Markus Nick, Hannes Schwarz, Audris Kalnins, Elina Kalnina, Markus Nick, Sören Schneickert, Edgars Celms, Katharina Wolter, Albert Ambroziewicz, and Jacek Bojarski. Repository selection report. Project Deliverable D4.4, ReDSeeDS Project, 2007. www.redseeds.eu.

[Bou05] Paolo Bouquet. Specification of a common framework for characterizing alignemnt. Technical report, KWEB EU-IST-2004-507482, 2005.

[EF95] Jürgen Ebert and Angelika Franzke. A Declarative Approach to Graph Based Modeling. In E. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graphtheoretic Concepts in Computer Science*, number 903 in LNCS, pages 38–50, Berlin, 1995. Springer Verlag.

[ELTV04] Jérôme Euzenat, David Loup, Mohamed Touzani, and Petko Valtchev. Ontology Alignment with OLA. In *Proceedings of the 3rd EON Workshop*, 2004.

[ES04] Marc Ehrig and York Sure. *The Semantic Web: Research and Applications*, chapter Ontology Mapping - An Integrated Approach. Springer, 2004.

[ES07] Jérôme Euzenat and Pavel Shvaiko. *Ontology matching*. Springer-Verlag, 2007.

[Euz04] Jérôme Euzenat. State of the art on ontology alignment. Technical report, KWEB EU-IST-2004-507482, 2004.

[FSH02] Christiaan Fluit, Marta Sabou, and Frank van Harmelen. *Visualizing the Semantic Web*, chapter Ontology-based Information Visualization, pages 36–48. Springer, 2002.

[HM76] James W Hunt and M Douglas McIlroy. An algorithm for differential file comparison. Computing Science Technical Report 41, Bell Laboratories, June 1976.

[JN06] Andreas Jedlitschka and Markus Nick. Scenarios, representation, and usage issues for software case-oriented comprehensive reuse. In *Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2006)*, pages 1–4, 2006.

[Ker06] Mick Kerrigan. WSMOViz: An Ontology Visualization Approach for WSMO. In *Proceedings of the Information Visualization (IV'06)*, 2006.

[KSC+07] Audris Kalnins, Agris Sostaks, Edgars Celms, Elina Kalnina, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Volker Riediger, Hannes Schwarz, Daniel Bildhauer, Sevan Kavaldjian, Roman Popp, and Jurgen Falb. Reuse-oriented modelling and transformation language definition. Project Deliverable D3.2.1, ReDSeeDS Project, 2007. www.redseeds.eu.

[KSS+07] Hermann Kaindl, Michał Śmiałek, Davor Svetinovic, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Hannes Schwarz, Daniel Bildhauer, John P Brogan, Kizito Ssamula Mukasa, Katharina Wolter, and Thorsten Krebs. Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project, 2007. www.redseeds.eu.

[LS06] Monika Lanzenberger and Jennifer Sampson. AlViz - a Tool for Visual Ontology Alignment. In *IV '06: Proceedings of the conference on Information Visualization*, 2006.

[NM03] N.F. Noy and M.A. Musen. The prompt suite: interactive tools for ontology merging and mapping. *Int. Journal of Human-Computer Studies*, 59:983–1024, 2003.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT-/SIGPLAN software engineering symposium on Practical software development environments (SDE 1)*, pages 177–184, New York, NY, USA, 1984. ACM.

[SNM+02] Margaret-Anne Storey, Natasha F. Noy, Mark Musen, Casey Best, Ray Fergerson, and Neil Ernst. Jambalaya: an interactive environment for exploring ontologies. In *IUI '02: Proceedings of the 7th international conference on Intelligent user interfaces*, 2002.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.

[WKB+07] Katharina Wolter, Thorsten Krebs, Daniel Bildhauer, Markus Nick, and Lothar Hotz. Software case similarity measure. Project Deliverable D4.2, ReDSeeDS Project, 2007. www.redseeds.eu.