FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Coupling Methodology and Tool for Software Development

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Stefan Melbinger
Matrikelnummer 0225692

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr. Hermann Kaindl
Mitwirkung: Proj.Ass. Dipl.-Ing. Dr. Jürgen Falb

Wien, 20.11.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Coupling Methodology and Tool for Software Development

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Stefan Melbinger
Registration Number 0225692

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr. Hermann Kaindl
Assistance: Proj.Ass. Dipl.-Ing. Dr. Jürgen Falb

Vienna, 20.11.2011

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Stefan Melbinger
Herbeckstraße 69/13, 1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

This thesis is dedicated to my beloved family who has always been there for me and has offered unconditional support throughout all of my life. I would not have come this far without you.

I especially want to thank Professor Hermann Kaindl for his support, his guidance and the sharing of his vast knowledge in the field of software engineering. Thanks also to Thomas Bruckmayer, a great co-worker and an even better friend.

Abstract

Nowadays, there are a variety of software development tools and methodologies. If a tool is used without further guidance, it is often not clear *what* to do with it. On the other hand, if the developer is provided only with a methodology, he may still wonder *how* to perform the prescribed tasks. However, even if both tool and methodology are present, the connection between these two supporting systems is still not immediately obvious to the developer.

This gap became apparent during work on the EU-funded ReDSeeDS Project. While providing a set of powerful languages, a profound high-level methodology and a tool supporting the suggested development process, there was no connection between primary tasks – the major objectives that a user wants to achieve – and secondary tasks – which have to be performed in order to accomplish the primary tasks.

This thesis presents a prototypical solution to this problem by providing guidance on how to tightly couple a software development methodology with a certain Eclipse Platform-based tool by transforming the methodology to a Web-based representation and linking to so-called cheat sheets within the tool. Thus, the Web pages present the primary tasks that need to be accomplished and link into the tool, where secondary tasks are presented and the user receives detailed step-by-step instructions on how to proceed. In addition, it is possible to invoke tool functionality directly from these cheat sheets.

Kurzfassung

Heutzutage gibt es eine große Anzahl an Softwareentwicklungstools und -methodiken. Wird ein Tool ohne weitere Anleitung benutzt, ist oft nicht klar, *was* damit zu tun ist. Falls dem Entwickler hingegen nur eine Methodik zur Verfügung steht, stellt sich die Frage, *wie* man die vorgeschriebenen Aufgaben absolvieren kann. Doch selbst wenn sowohl Tool als auch Methodik vorhanden sind, ist dem Entwickler die genaue Beziehung zwischen diesen beiden Hilfestellungen nicht unmittelbar klar.

Diese Kluft wurde während der Arbeit an dem EU-finanzierten ReDSeeDS Projekt offensichtlich. Das Projekt stellt zwar eine Reihe an mächtigen Sprachen, eine umfassende High-Level Methodik und ein unterstützendes Tool zur Verfügung, es gab jedoch keine Verbindung zwischen sogenannten Primary Tasks – grundlegenden Aufgaben, die ein Benutzer erledigen will – und Secondary Tasks – die ausgeführt werden müssen um die Primary Tasks zu absolvieren.

Diese Arbeit präsentiert eine prototypische Lösung dieses Problems und zeigt, wie man eine Softwareentwicklungsmethodik eng mit einem bestimmten Eclipse Platform-basierten Tool koppeln kann, indem man die Methodik in eine Web-basierte Darstellung überführt und dann mit sogenannten Cheat Sheets im Tool verlinkt. Die Webseiten stellen so die Primary Tasks dar und verweisen in das Tool, wo der Entwickler die Secondary Tasks als detaillierte Schritt-für-Schritt Anleitungen erhält. Zusätzlich ist es möglich, Aktionen im Tool direkt aus den Cheat Sheets ausführen zu lassen.

Contents

CHAPTER 1

Introduction

This chapter gives a short introduction to this thesis and its chapters. Section 1.1 discusses the basic problem statement the author has been working on. The project which sparked research on the topic of coupling a tool and a methodology is summarised in Section 1.2, while related work regarding this topic is given in Section 1.3. Finally, Section 1.4 gives an overview of the structure of the remaining chapters.

1.1 Problem Statement

Software engineers may choose from a wide number of software development methodologies that are intended to guide them through the process of developing software and the various phases within such a process. Traditional methodology instructions in books, however, are "doomed to sit on a shelf and get dusty" [20]. While Web technology has increased the level of comfort for developers, they might still ask *how* to operationalise the instructions they receive if no appropriate tool support is available.

On the other hand, if software development tools like powerful CASE[1] tools are available, they may enable the user to produce complex artefacts. Still, it might not be clear *what* exactly the user is supposed to do with these tools in order to advance the software development process he is working on if there are neither guidance nor sufficient help available.

Ideally, both methodology (in the form of books, tutorials, Web pages, courses or by other means) and tool support are available to the developer. But still, there is a gap between these two that needs to be filled. Users require a methodology that, on the one hand, guides them and explains *what* the main objectives are they need to fulfil, and on the other hand clearly instructs them *how* to perform the necessary tasks with a specific tool in a specific environment. This thesis shows a way of *coupling* a software development methodology and a specific tool in order to guide the end-user from high-level objectives down to low-level step-to-step instructions.

[1]Computer-aided Software Engineering

1.2 The ReDSeeDS Project

The need for coupling a software development methodology and a software engineering tool became obvious while working on the *ReDSeeDS Project*[2]. The name ReDSeeDS is an abbreviation for "Requirements-driven Software Development System". A number of universities (including the Vienna University of Technology) and industrial partners have contributed to this EU-funded project, whose main objective was to "create knowledge and technology allowing for comprehensive software reuse by bringing it to the level of requirements linked with precise model-based solutions" [9]. Extensive research in a number of fields like Requirements Engineering, Case-based Reuse and Model-driven Development has led to the definition of a family of specification languages and the implementation of a powerful set of tools. They allow for the exact specification of requirements and software design models, for the reuse of software cases based on the similarity of requirements, and for the automatic transformation of software development artefacts between different stages of development.

Given these powerful languages and tools, it is obvious that potential users need clear guidance on how to use these tools for their intended purpose. This is why one objective of the ReDSeeDS Project was to create a software development methodology. Still, with the methodology and the tools available, there clearly was a gap between them. Those activities in the methodology and those in the tool were not clearly mapped to each other and the need for "comprehensive and connected task descriptions" was apparent [20]. Thus, research was done on how to enable tight coupling between methodology and tool. A possible solution to this question is given in Chapter 5.

1.2.1 Main Ideas

ReDSeeDS builds upon a number of ideas and paradigms which are briefly introduced in this section.

Requirements Engineering

The ReDSeeDS Project's most important concept is that of *requirements*. Requirements are the fundament on which all artefacts and processes in ReDSeeDS build on. They have to be elicited from stakeholders and then specified formally, so that their similarity to formerly specified requirements can be measured and used to identify similar software products for possible reuse. Furthermore, requirements may be the source for transformations which advance the software development process up to the point where actual code is generated.

This explains the importance of Requirements Engineering in this project. The main results in this area have been the concept of strict separation between requirements and their representations (see [25]) and the creation of the complex semi-formal *Requirements Specification Language (RSL)* for specifying requirements that are coherent and unambiguous (see [24]).

The traditional reuse of design patterns, reference architectures and components in software development implies the reuse of some background requirements. RSL, however, was specifically designed with enabling systematic reuse of requirements in mind.

[2]http://www.redseeds.eu/

Case-based Reuse

The notion of *software cases* has been a central concept within the ReDSeeDS Project and has been used throughout its technologies and deliverables. In general, as discussed in [14], a software case is a set of artefacts specifying a software system. In ReDSeeDS, its specific components are the *Requirements Model*, the *Architectural Model*, the *Detailed Design Model*, *Code* and *Transformation Rules*, as depicted in Figure 1.1. These artefacts have to be specified

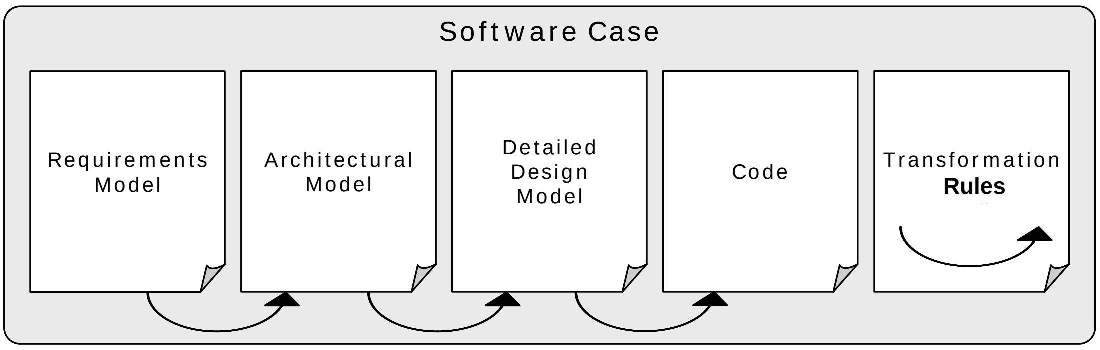


Figure 1.1: Components of a software case, from [39]

using appropriate languages and models. The *ReDSeeDS Software Case Language (SCL)* defines such textual and graphical facilities.

All ReDSeeDS languages and artefacts have been designed with future reuse in mind. As described below, the ReDSeeDS Engine provides a prototypical implementation of a tool that allows the storage and retrieval of complete or partial software cases. When work is started on a new software case, its requirements can be compared to those of other software cases. The most similar ones may then be reused easily. This case-based approach "allows reuse without the usual and significant effort for making software explicitly reusable" [23].

Model-driven Development

Figure 1.1 also features directed links between the artefacts. This symbolises the automatic transformations that can be applied through the ReDSeeDS Engine. They allow generating the basis for the artefact which is next in line in this chain of more and more detailed and close-to-code artefacts.

Building software cases based on systematic transformation as it is done in ReDSeeDS is a fundamental feature in *Model-driven Development (MDD)*. In contrast to other model-driven software development approaches, however, ReDSeeDS applies transformations already during the first step in the software development process – namely from requirements which are specified using a well-defined semi-formal language to an initial architecture model in a subset of the popular modelling language UML. This is one remarkable aspect of MDD in ReDSeeDS and further discussed in [19].

For specifying and executing transformations, the *Transformation Language MOLA* is used.

1.2.2 Languages, Tools and Methodology

The ReDSeeDS Project delivered a number of documents and prototypes which show that the original ideas can, in fact, be realised. As noted above, the Software Case Language (SCL) can be used to specify software case artefacts. It consists of the *Requirements Specification Language (RSL)*, the *Software Development Specification Language (SDSL)* and the *Transformation Language MOLA*. While these languages work together and complement each other, they could be used on their own for projects outside the ReDSeeDS framework as well. Chapter 2 gives an overview over SCL.

In order to ease working with these languages and to make full use of all ReDSeeDS ideas and features, the project provides tool support to the end-user (see Chapter 3 for details). The ReDSeeDS Engine is a prototypical implementation of a development platform which allows the specification of complex requirements, the transformation of software artefacts and the case-based reuse of former software cases. To model the architectural and detailed design models and to generate code from these models, the external application *Enterprise Architect* is used. It was not created in the ReDSeeDS Project but is tightly linked to the ReDSeeDS Engine. Finally, the external tool *AMUSEd* is used to specify user interfaces.

The ReDSeeDS languages and tools can hardly be used as intended and coherently without guiding end-users through the different stages of software development. The *ReDSeeDS Software Development Methodology*, which is available both as a hypertext representation and as a linear document (see Chapter 4), defines a number of development processes which can be viewed at different levels of granularity – right down to the description of single activities. By linking these activities to detailed explanations of *how* to perform a certain task within the Engine, i.e. by coupling methodology and tool, ReDSeeDS aims for the user to experience an easy and pleasant way of learning to be productive with a set of complex but powerful technologies and tools.

1.3 Related Work

The ReDSeeDS Software Case Language, the Engine and the Methodology build upon a vast amount of previous research. To focus on the one field of study especially important for this thesis, the software development methodology defined in this project has been influenced by a number of previously specified methodologies like SEM [22] and the Rational Unified Process [29]. The latter is "strongly centered around use cases and UML", "modelled using SPEM" and "comes in the form of a web-based hypertext" [39] as well. Despite these similarities, the ReDSeeDS Methodology stands unique for its requirements-driven nature and the usage of its own languages.

Similarly to the ReDSeeDS Methodology, the software engineering environment described by Chen et al. in [3] also presents itself to the user in the form of Web pages. The system allows modelling, execution and monitoring of process workflows. While the work to be done is shown to users on HTML pages and also allows execution of the necessary tools, there is no tight coupling with these tools as it is described in this thesis; i.e. there is no actual linking *into* these tools.

The use of Eclipse cheat sheets to facilitate a comfortable learning process for users has proven successful in Safer et al.'s work [35], which "integrates cheat sheets and the task management framework Mylyn [...] to prevent overloading the development environment with cheat sheet documentation, editor views, project structure views, lists of files, etc." [20]. Instead, the plug-in determines which information is important and relevant to a learning task at hand and hides resources which are not currently needed from the user.

The requirements engineering tool RETH presents a way of allowing users to learn about the program by interactively following guidance through a process represented within the tool itself [21]. Thus, the methodology describing the primary tasks as well as the instructions describing the secondary tasks are directly embedded in the tool. This solution, however, was no option for the ReDSeeDS Project where methodology and tool are logically and physically separated from each other.

1.4 Structure of the Thesis

Following this chapter, Chapter 2 gives an overview over the *ReDSeeDS Software Case Language (SCL)*, i.e. the Requirements Specification Language (RSL), the Software Development Specification Language (SDSL) and the Transformation Language MOLA. It is useful to know these languages' basics because they lay the foundation for all ReDSeeDS tools and methodology descriptions and will be referred to in the other chapters as well.

Chapter 3 describes the *ReDSeeDS Engine* which exists as a working prototypical implementation. In collaboration with external tools, it provides the user with a set of powerful means for defining, reusing and transforming software cases.

Chapter 4 summarises the *ReDSeeDS Software Development Methodology*, which guides the end-user through the process of software development with requirements-driven reuse in mind.

As mentioned above, these chapters already describe the most important parts of ReDSeeDS and could be used as they are by experienced developers. However, what to do in the tool according to the methodology and especially how to do it may not be clear at this point. This is where the *tight coupling* between the ReDSeeDS Engine and the ReDSeeDS Methodology comes into play, as elaborated in Chapter 5. While the prototypical implementation given in this chapter is specific for ReDSeeDS, its basic idea is universal and can easily be transferred onto other projects.

Finally, a short conclusion of the thesis is given in Chapter 6.

CHAPTER 2

ReDSeeDS Software Case Language

The ReDSeeDS *Software Case Language (SCL)*, which is not actually one language but rather a "family of separate languages" [28], is used to precisely specify all artefacts of a software case in a "structured and traceable manner" [27]: requirements, architectural models, detailed design models and the model transformations which lead from one model to another. Of course, source code is also considered to be part of a software case and may be generated from the detailed design model. The languages used for the different artefacts are described in the next sections. Due to ReDSeeDS' metamodel-driven approach, however, they could in theory be replaced by other languages adhering to the structure and well-formedness rules of the given metamodels.

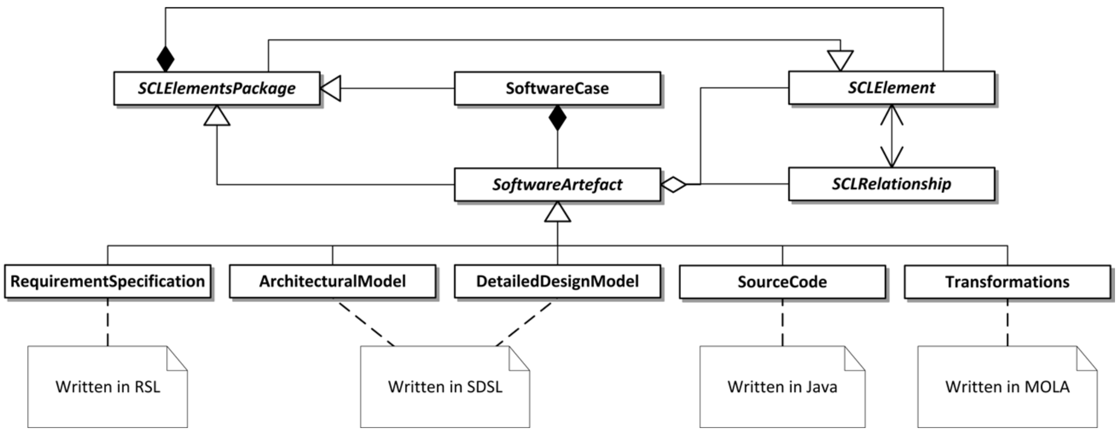


Figure 2.1: Simplified SCL and software case structure, adapted from [27]

Figure 2.1 illustrates the various parts of SCL and their relationships to software cases. Since the main idea behind ReDSeeDS is to enable reuse of software cases based on their requirements, the corresponding *Requirements Specification Language* (see Section 2.1) is considered to be the most important part of SCL. Architecture and Detailed Design are specified in the *Software Development Specification Language* (see Section 2.2), which is actually a subset of the widely spread modelling language UML. ReDSeeDS uses *Java* to implement source code; however, developers might choose to put another programming language in its place. For specifying transformations between the various models in a software case, the transformation language *MOLA* (see Section 2.3) is used.

It has to be noted that developers do not need to know a lot about SCL's "inner workings" because the ReDSeeDS Engine (see Chapter 3) provides a convenient interface to the different models and because the ReDSeeDS Methodology (see Chapter 4) gives comprehensive guidelines on how to use the available tools in everyday situations.

2.1 Requirements Specification Language

This section gives a conceptual overview over the ReDSeeDS *Requirements Specification Language (RSL)* [24]. Most current requirements specification languages "focus on formal representation only and are not used much in practice" or "provide a subset of natural language only and do not provide means for conceptual modelling". Additionally, most of these languages "do not integrate user-interface specifications" [24]. RSL finally combines constrained and unconstrained requirements descriptions, conceptual requirements modelling and user-interface specifications into one comprehensive language. In any case, formal representations are preferred to non-formal ones in order to allow better matching of software cases for reuse.

RSL is defined using the meta-modelling language MOF (see [11]) and takes advantage of some UML packages which have been merged into its description.[1] RSL consists of three parts: *Requirements Language*, *Requirements Representation Language* and *Application Domain Language*. This clearly facilitates the separation of concerns and reduces the complexity of the language. Also, in order to ensure that requirements specifications are "unambiguous and consistent", the application domain is not part of the requirements representations. Instead, the representations are linked to phrases and terms in the domain model through hyperlinks.

The division between requirements and requirements representations, however, represents RSL's main insight. The widespread confusion between requirements and their representations is extensively discussed in [25] and a number of reasons why "the documented representation of the requirement should not be considered to be the requirement itself" are given in [24]:

- A requirement is something that exists even if it is never documented, i.e., represented.
- A requirement can have multiple representations.
- A requirement can be represented at different abstraction levels.
- Each representation of a requirement is usually not complete from each possible perspective.

This section is further structured into a basic description of RSL's behavioural part (describing the Requirements Language and the Requirements Representation Language for functional and behavioural requirements), its structural part (describing domain entities and dictionaries)

[1]For an introduction to UML, see Section 2.2.

and its user interface part. For a complete reference of RSL including the concrete syntax of all its elements, see [24].

2.1.1 Behavioural Part

Requirements and Goals Model

The Requirements and Goals Model is made up of three interconnected metamodels, whose separation is colour-coded in Figure 2.2: the *Requirements Metamodel* (pink), the *Goal Metamodel* (blue) and the *Task Metamodel* (yellow).

The Requirements Metamodel's top entity is `Requirement`, which can be used in the form of one of four different specialisations, which will be discussed below.

- *Use Cases*

  ReDSeeDS follows the definition of Use Case given in [5]:

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

  Accordingly, Use Cases "consist of Envisioned Scenarios that belong together in terms of use" [24]. For example, the Use Case "Get cash money from ATM" might have different ways of actually being achieved.

- *Envisioned Scenarios*

  Envisioned Scenarios allow the fulfilment of *Functional Requirements on Composite System* (for example "Cash withdrawal") and concern not only the software system itself but also external systems and actors. A scenario is a "sequence of interactions happening under certain conditions, to achieve the primary actor's goal, and having a particular result with respect to that goal" [5] and discussed in [18]:

  > Early system design and capturing user requirements are widely believed to be well supported by the use of *scenarios* .... Such interaction scenarios have been proposed (among others) for capturing and representing the interactions of potential users with a computer system that has yet to be built. When these usage descriptions are concrete, scenarios help to discuss use and to design use.

  The scenario is said to be "envisioned" because it has not been implemented yet.

- *Functional Requirements*

  The majority of requirements are Functional Requirements, which represent capabilities needed "by a user to solve a problem or achieve an objective" or "by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents". There are two types of Functional Requirements. *Functional Requirements on*

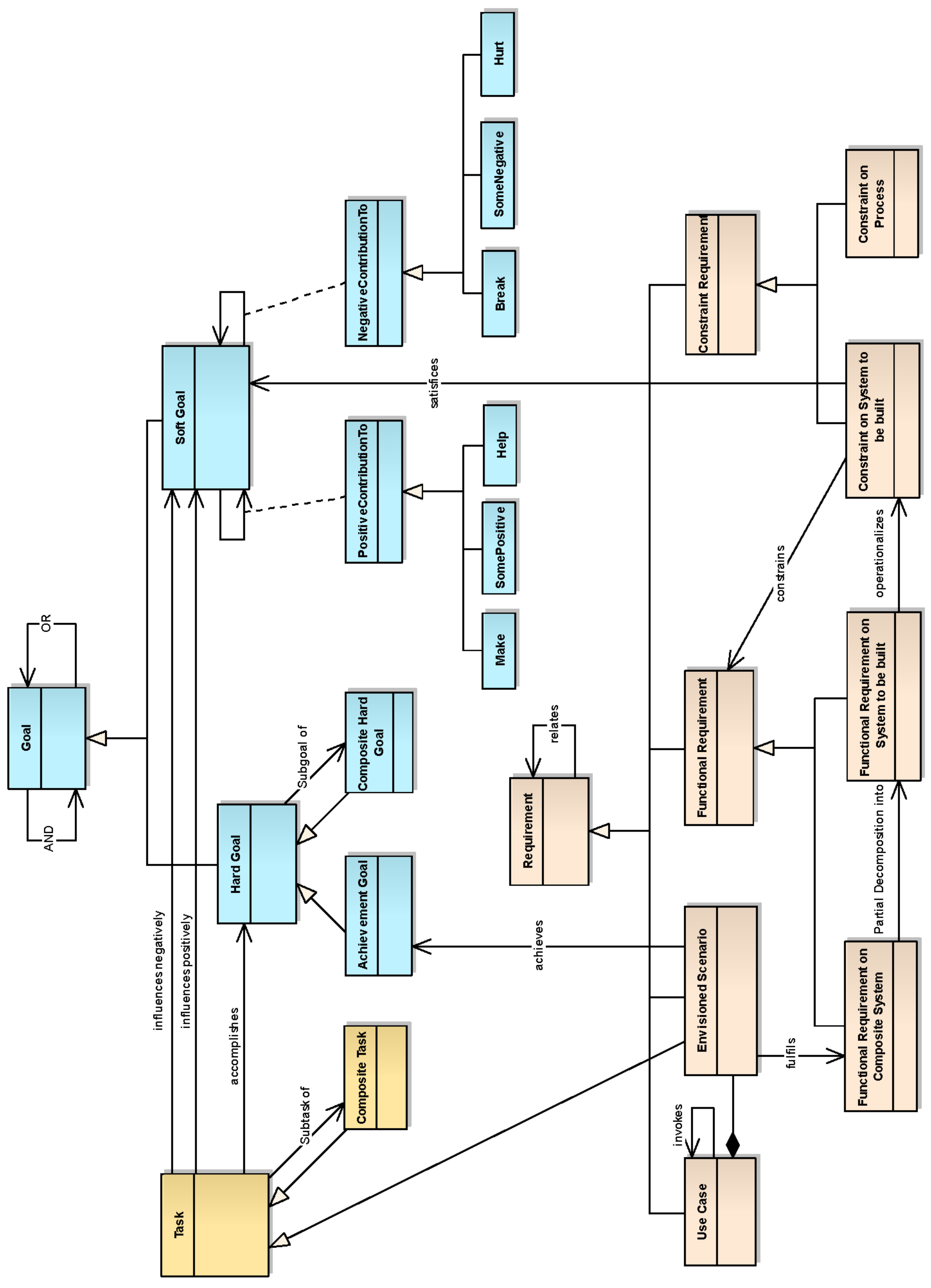


Figure 2.2: Requirements and Goals Model, from [26]

Composite System are "supposed to be fulfilled by the system composed of system to be built and its actors" and are "primarily fulfilled through the envisioned scenarios".

  They are partially decomposed into *Functional Requirements on System To Be Built*, which are "supposed to be fulfilled by the system that is being specified" [24]. They primarily facilitate the enactment of Envisioned Scenarios. Sometimes, they may also operationalise constraint requirements – for example, a security requirement might be met by the function accepting a password.

- *Constraint Requirements*

  Constraint Requirements represent "mandatory or optional properties of a system" [24] and are traditionally called "non-functional requirements". Constraint Requirements can be divided into *Constraints on Process*, which may, for example, constrain development methods and tools, and *Constraints on System To Be Built*. The latter "may relate to e.g. security, reliability and performance conditions or even to political, legal and economical aspects" [7] and are tightly related to functional requirements, which they may constrain.

The Goal Metamodel distinguishes *Soft Goals* from *Hard Goals*, both of which may be set in relation to other goals by logical *AND* or *OR* connections. Hard Goals can be arranged into a hierarchy of goals which are fulfilled when when their low-level *Achievement Goals* are achieved by successful execution of a related Envisioned Scenario. These are goals whose achievement can be determined exactly. Soft Goals, on the other hand, can only be achieved to a certain degree or, as [30] puts it, be "satisficed". They are designed to allow trade-offs between constraint requirements like security and usability by contributing positively or negatively to each other.

Finally, the Task Metamodel introduces a task as a "piece of work that a person or other agent has to (or wishes to) perform" [24]. Thus, Envisioned Scenarios are a specialisation of `Task`.

Requirements Representation Model

The Requirements Representation Model, as seen in Figure 2.3 and further discussed in [25], conceptually explains how requirement entities from the Requirements Language may be represented as either *Descriptive Requirements Representations* or *Model-based Requirements Representations*. Bundled Model-based Requirements Representations make up *Requirements Models* and together with Descriptive Requirements Representations they make up whole *Requirements Specification Documents*.

Descriptive Requirements Representations "describe the needs of certain requirements" [24] and are expressed through *Natural Language Requirements Representations* (i.e. English or German text, together with hyperlinks to the structural part of RSL, as suggested for example in [16]) and/or constrained language statements. The latter may either be *Constrained Language Requirements Representations* in the form of SVO(O)[2] expressions, which combine the grammar of a formal language with the readability of a natural language, or *Constrained Language Scenario Representations*, which use such a constrained language to describe Envisioned Scenarios.

[2]Subject – Verb – Object – (Object)

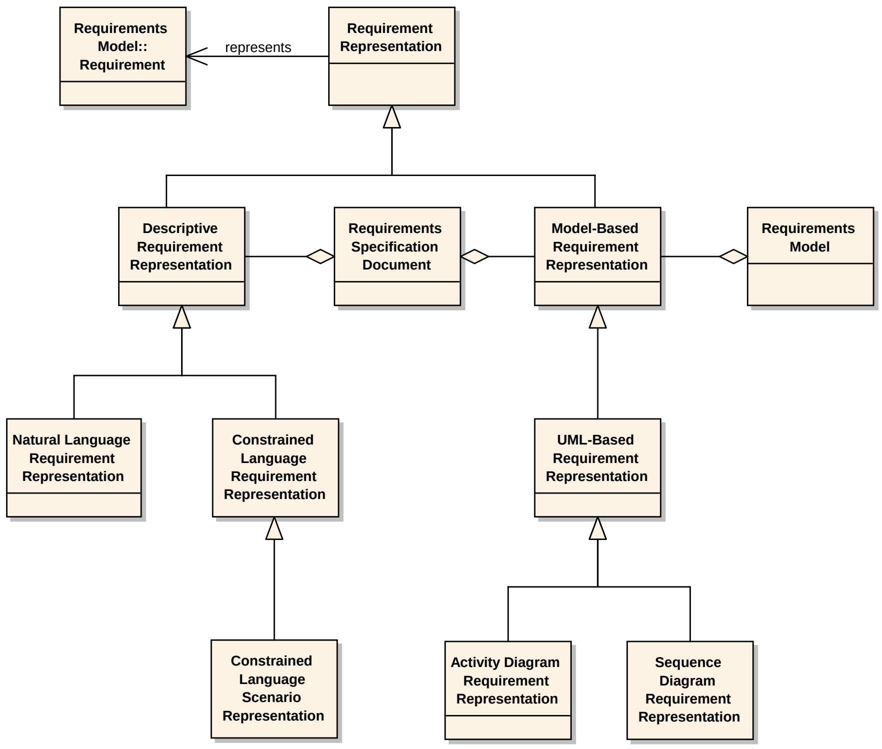

Requirements Model:: Requirement
represents
Requirement Representation
Descriptive Requirement Representation
Requirements Specification Document
Model-Based Requirement Representation
Requirements Model
Natural Language Requirement Representation
Constrained Language Requirement Representation
UML-Based Requirement Representation
Constrained Language Scenario Representation
Activity Diagram Requirement Representation
Sequence Diagram Requirement Representation

Figure 2.3: Requirements Representation Model, from [24]

Model-based Requirements Representations represent models of the system to be built. The models used for this purpose are the UML activity and interaction diagrams, which have been specialised for requirements specification purposes in the ReDSeeDS Project.

2.1.2 Structural Part

The structural part of RSL deals with "models and descriptions of objects existing in the domain (environment) of the software system to be built – domain objects" [24]. Since contradictory definitions of terms are responsible for most inconsistencies in requirement specifications, RSL focuses on clearly distinguishing requirements and their representations from representations of the problem domain. In contrast to defining objects ("notions") in various places within requirement descriptions, they are specified in one place only and can be linked to via hyperlinks. That is, requirements representations link to definitions of phrases and terms in RSL.

A simple example can be seen in Figure 2.4: On the left side, there is a scenario description in RSL grammar. The underlined nouns link to descriptions in the separate domain specification, where they are defined unambiguously. For example, "Exercises" may be defined as follows: "Exercises – Form of physical activity performed in fitness club. Exercises may be [cyclic exercises] or [sporadic exercises].", where the expressions in square brackets symbolise links to other expressions in the vocabulary. Notions can have several forms, synonyms and homonyms

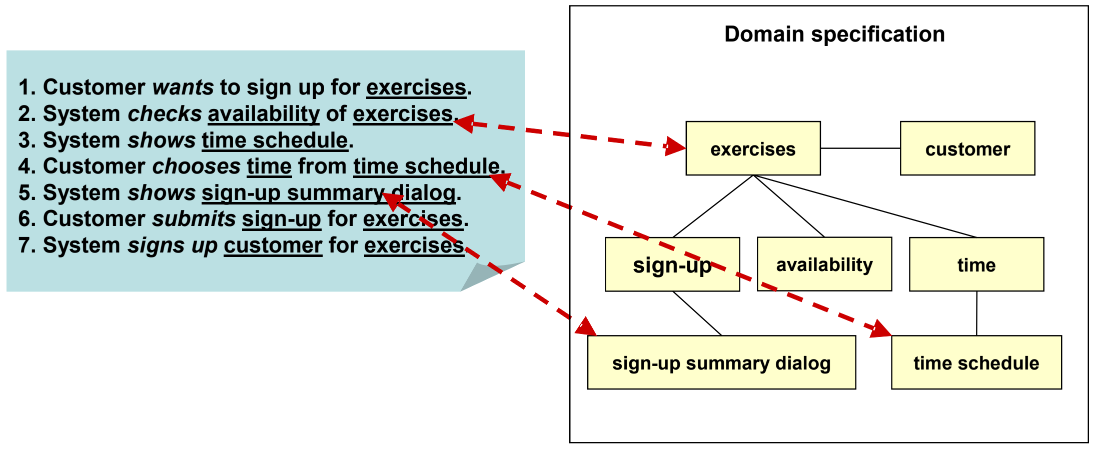

1. Customer wants to sign up for exercises.
2. System checks availability of exercises.
3. System shows time schedule.
4. Customer chooses time from time schedule.
5. System shows sign-up summary dialog.
6. Customer submits sign-up for exercises.
7. System signs up customer for exercises.
Domain specification
exercises
customer
sign-up
availability
time
sign-up summary dialog
time schedule

Figure 2.4: Scenario with separated domain specification, from [24]

and also include verbs, but only in combination with nouns since they have different meanings depending on the context.

Representation of Domains

Domains can be specified using *conceptual models* and *phrases*. Using conceptual models, RSL allows domain elements (nouns) to be structured in way similar to object-oriented modelling. Domain elements can be generalised and connected through associations which can be aggregations and can have multiplicities, thus resulting in a simple form of ontology. Still, the meaning of domain entities might not be clear from these relationships alone, so that there is a need for an additional dictionary or glossary, as it has been introduced in [17], for example.

For this reason, RSL allows domain elements to be connected to a number of phrases containing the noun's plain description or more complex statements with verbs, modifiers and qualifiers. Of course, these statements can and shall make use of the hypertext mechanism mentioned above, interconnecting domain entities.

RSL introduces *terms* in order to build phrases. This is done by providing a dictionary, which describes terms and gives information about their inflections and possibly also translations into other languages, and a thesaurus which sets the terms into relation with each other.

2.1.3 User Interface Part

RSL allows user interface requirements to be specified by modelling "user interface elements and their dynamics". This is done by providing "prototype-oriented model-based elements" which have been abstracted to be independent from concrete modalities, toolkits and platforms [24]. These elements distinguish between *atomic* elements for direct user interaction (like input fields and lists) and *container* elements for structuring the user interface.

Additionally, user interface behaviour can be specified using narrative prototypes – *story-boards* – which are known in other software development processes like the Rational Unified Process [29] as well.

2.2 Software Development Specification Language

This section summarizes the part of SCL called *Software Development Specification Language (SDSL)*. As detailed in [27], SDSL is used for modelling the Architectural and Detailed Design parts of a software case which have rough counterparts in the *Model Driven Architecture* (MDA), namely the *Platform Independent Model (PIM)* and the *Platform Specific Model (PSM)*, respectively.

Other than the name might suggest, SDSL is not an entirely new language but rather "a set of guidelines for using the existing modelling language" [9], namely the *Unified Modeling Language (UML)*. UML is the most prevalent modelling language in software development and has been standardised by the Object Management Group (OMG). ReDSeeDS mainly uses version 2.1.1, which is specified in [12]. It offers a variety of thirteen diagrams which represent two different views of a system model: the static (or structural) view and the dynamic (or behavioural) view [13]. It "emphasizes the idea that complex systems are best described through a number of different views, as no single view can capture all aspects of such a system completely" [36]. However, making use of all parts of UML is usually neither necessary nor useful, as explained in [12]:

> UML is a language with a very broad scope that covers a large and diverse set of application domains. Not all of its modeling capabilities are necessarily useful in all domains or applications. This suggests that the language should be structured modularly, with the ability to select only those parts of the language that are of direct interest.

Following this advice and having the intention of enabling software case reuse in mind, SDSL has been designed as a subset of UML plus some extensions which have been realised with UML profiles. Besides the formal description of SDSL, guidelines on how to build models for ReDSeeDS are provided. This solution satisfies the three main requirements for the ReDSeeDS modelling language, namely "to be well harmonised with the formal requirements specified in RSL", to provide "the possibility to define in an easy way the traceability links to these requirements" and to facilitate "the use of automatic transformations" [27]. Additionally, the restricted use of UML allows developers to quickly comprehend the needed elements.

SDSL provides two so-called *styles*: the *Basic Style* and the *Keyword-based Style*. These styles include "the chosen system and model structure, the related set of design patterns (with indications where they should be used), general design principles and finally, the way model elements are obtained from models preceding in the development chain". While the Basic Style focuses on the "basic" Architecture and Detailed Design models, the Keyword-based Style intends to "extract more behaviour-related information from scenarios in RSL and to 'transport' it as much as possible through to the code level" [27]. The two styles require different transformation rules in order to generate models from requirements. Information about the tool support for these transformations can be found in Chapter 3.

2.2.1 Basic Style

Software cases in the Basic Style comprise the following entities:

- Requirements Model
- Architecture Model
- Detailed Design Model
- Code
- Model Transformations

Components therein are arranged in a "four-layer architecture" consisting of the *Data Access*, *Business Logic*, *Application Logic* and *User Interface* layers. Functional requirements are fulfilled by the Application Logic layer and should not be realised in the User Interface layer. Data exchange between the four layers is offered by *Data Transfer Objects* (DTOs) which serve as data containers.

Architecture Model

The Architecture Model features a static and a dynamic part. While the static part specifies the architecture's elements and their relationships towards each other, the dynamic part specifies their behaviour and interactions.

The static part is specified using the UML *component diagram* and the following selection of elements: components (which represent parts of a logic structure on any of the four layers), interfaces (of components), dependencies (between components), classes (which represent data transfer objects) and packages (for grouping elements). An example for such a component diagram can be seen in Figure 2.5.

For the dynamic part, the UML *sequence diagram* is used along with the following elements: lifelines (which represent either a lifeline of an actor, of a component or an interface) and messages between the lifelines. For an example, see Figure 2.6.

Detailed Design Model

Detailed Design is the "lowest level of project specification" and, therefore, "the basis of implementation in the concrete programming language" [27]. It consists of all logic elements of a software system and depicts their static properties in a UML *class diagram* and their dynamic behaviour in a UML *sequence diagram*. The class diagram uses the class, interface and `DataType` elements along with a number of relationships between them, like association and

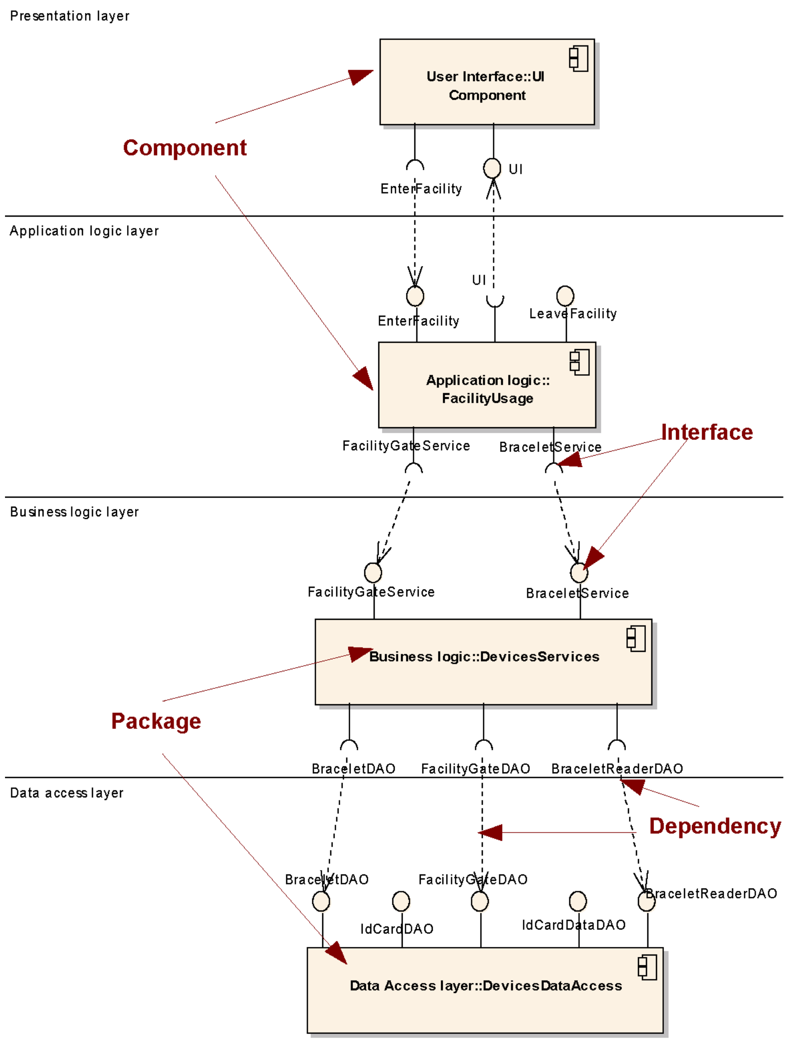


Figure 2.5: 4-layer Basic Style Static Architecture Model example, adapted from [27]

aggregation. An example can be seen in Figure 2.7. It shows the class diagram of an interface implementation and its associated elements, e.g. data transfer objects.

The sequence diagram, similar to the one used for the Architecture Model, features lifelines for actors, components, classes and interfaces, messages between the lifelines and combined fragments for expressing loops.

2.2.2 Keyword-based Style

The "main goal for developing the Keyword-based architecture style is to provide significantly more powerful capabilities for transformations in generating models from requirements" and thus "requires more detailed requirements to be specified in RSL in order to get richer code gen-

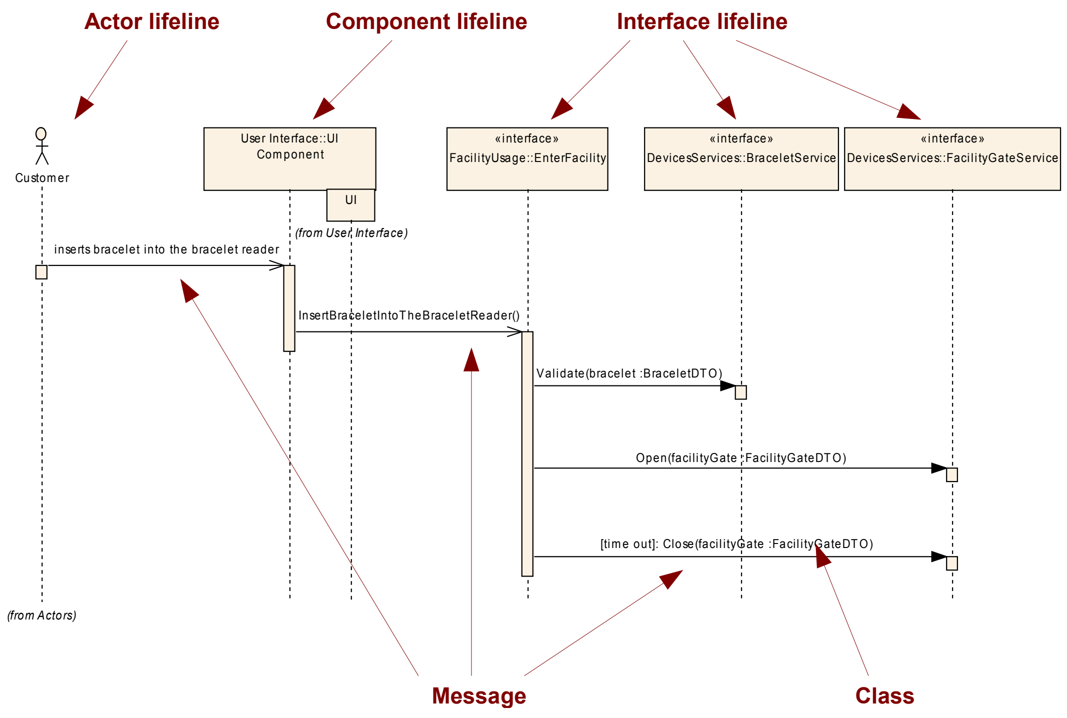

Actor lifeline
Component lifeline
Interface lifeline
Customer
User Interface::UI Component
UI
(from User Interface)
«interface» FacilityUsage::EnterFacility
«interface» DevicesServices::BraceletService
«interface» DevicesServices::FacilityGateService
inserts bracelet into the bracelet reader
InsertBraceletIntoTheBraceletReader()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

Figure 2.6: 4-layer Basic Style Dynamic Architecture Model example, from [27]

erated in the result" [27]. When applying the Keyword-based Style, RSL scenarios are analysed by identifying certain SVO sentence elements as *keywords*. They allow gathering non-trivial behaviour information which can then be used to generate a more sophisticated Architecture Model. The result of this process is information about required UI elements and their relationships to domain elements. Together with domain-related information from RSL notions, it is put into a class diagram called the *Analysis Model*.

The following keywords are recognised: *select*, *show*, *build*, *remove*, *add*, *enter*, *form*, *list*, *empty*, *select* and *link*. Quoting [27]:

> The meaning of keywords is natural and in most cases self-explanatory. Thus the keyword *show* means that the system must display a form which is defined by the direct object of this sentence. This object, in turn, must correspond to a notion, whose complex name ends with the noun keyword form. For example, the SVO sentence: "System shows reservable facility list form" specifies that the form "reservable facility list form" must be displayed at this point.

While the Keyword-based Style still uses the same four-layer architecture as the Basic Style, the Architecture Model and Detailed Design Model were changed to resemble classic MDA's

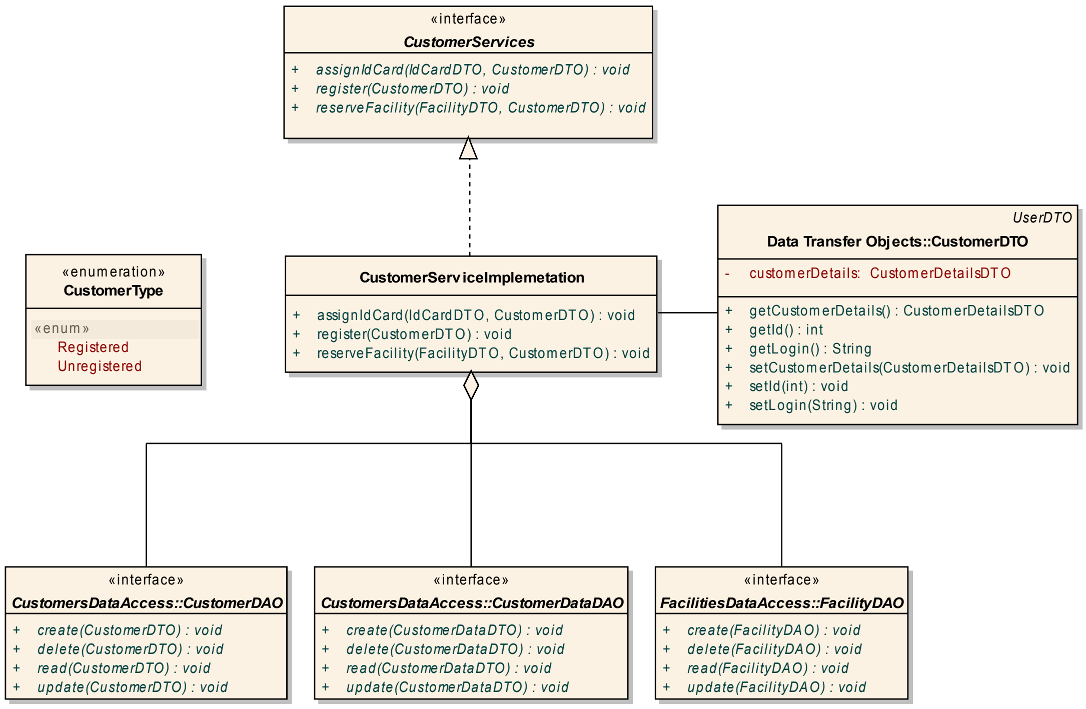

«interface»
CustomerServices
+ assignIdCard(IdCardDTO, CustomerDTO) : void
+ register(CustomerDTO) : void
+ reserveFacility(FacilityDTO, CustomerDTO) : void
«enumeration»
CustomerType
«enum»
Registered
Unregistered
CustomerServiceImplemetation
+ assignIdCard(IdCardDTO, CustomerDTO) : void
+ register(CustomerDTO)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

Figure 2.7: Basic Style Static Detailed Design Model example, from [27]

PIM and PSM more closely: The Architecture Model's static information is now contained in a hierarchy of packages which allows building more flexible and powerful sequence diagrams and the Detailed Design Model now contains platform-specific information for a software framework using Java[3] with the Spring[4] and Hibernate[5] frameworks.

2.3 Transformation Language

This section gives an overview over the transformation language used within SCL, namely *MOLA*. The reasons for choosing this language are given in [27]:

> Though there is a growing interest in such [model transformation] languages, no widely accepted candidate exists. The efforts by the OMG to propose a standard in this area have not succeeded yet. Therefore, we decided to use the transformation language MOLA, being developed by one of the ReDSeeDS partners – University of Latvia IMCS.

[3] http://www.java.com/
[4] http://www.springsource.org/
[5] http://www.hibernate.org/

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

Source and target metamodels are specified in the *MOLA metamodelling language* [34] (which is closely related to the OMG MOF specification [11]) by means of class diagrams, packages (to group the classes) and traceability associations (between corresponding source and target classes). The metamodels may be identical, which is "the case for in-place model update transformations" [27].

MOLA transformations are actually procedural programs in a graphical form. There can be an arbitrary number of *procedures*, as long as there is exactly one main procedure which is the entry point for executing the transformation of a source model. As soon as the main procedure exits, the transformation is finished. In order to determine the control flow, arrows are used to indicate the next statement and *call* statements can be used to invoke sub-procedures. The most important concepts in MOLA are, however, *rules* and *foreach loops* based on rules. A rule tries to match a pattern defined in its *pattern* part. If a match was successful, the *action* part is executed. The two parts are described in [27] as follows:

> Both are defined by means of class elements and links. A *class element* is a metamodel class, prefixed by the element ("role") name.
>
> ...
>
> An *association link* connecting two class elements corresponds to an association linking the respective classes in the metamodel. A pattern is a set of class elements and links, which are compatible to the metamodel for this transformation.
>
> ...
>
> A pattern element may contain also a *constraint* – a Boolean expression in a simplified subset of OCL.

Identifying a match is done by searching for an instance set "where an instance of the appropriate class is allocated to each pattern element so that all required links are present in this set and all constraints evaluate to true" [27]. A successful match then handles the class elements and links in the rule's action part, which represent either elements to be created, modified or deleted, or assignments of attribute values within the target model. The desired operation is executed only once, even if more than one matches have been found. If there is reason to suspect more than one successful matches for a rule, the foreach loop mechanism has to be used. If there has been no match, nothing happens (unless there is a special *ELSE-exit* directing the control flow onto a different path).

Foreach loops' main element is the *loop head* containing the *loop variable*, which iterates over instances of a specific class which satisfy the given pattern. It then executes the given action on each of them. Since the loop head is a rule itself, simple transformations can be done within it. For more complex operations, the *loop body* may contain other rules, control flow statements and even nested loops.

There are a few other less often used constructs in MOLA, which are described in detail in [27].

2.3.1 Example

The following simple example is explained in detail in [27] and only roughly sketched here in order to demonstrate MOLA's main features graphically. The combined source and target metamodel is depicted in Figure 2.8. The imaginary task is to transform a class diagram into a simple database schema definition featuring tables and columns. The class diagram metamodel has been taken from the UML *Classes* package and simplified. Classes have a number of *Property* elements associated, which have to be analysed during the transformation process. Additionally, only persistent classes shall be transformed – this is done by checking the `isPersistent` attribute. The target metamodel consists of tables and columns in the *SQL* package, which are linked by either the `cols` association for regular columns, or the `pkey` association for the primary key.

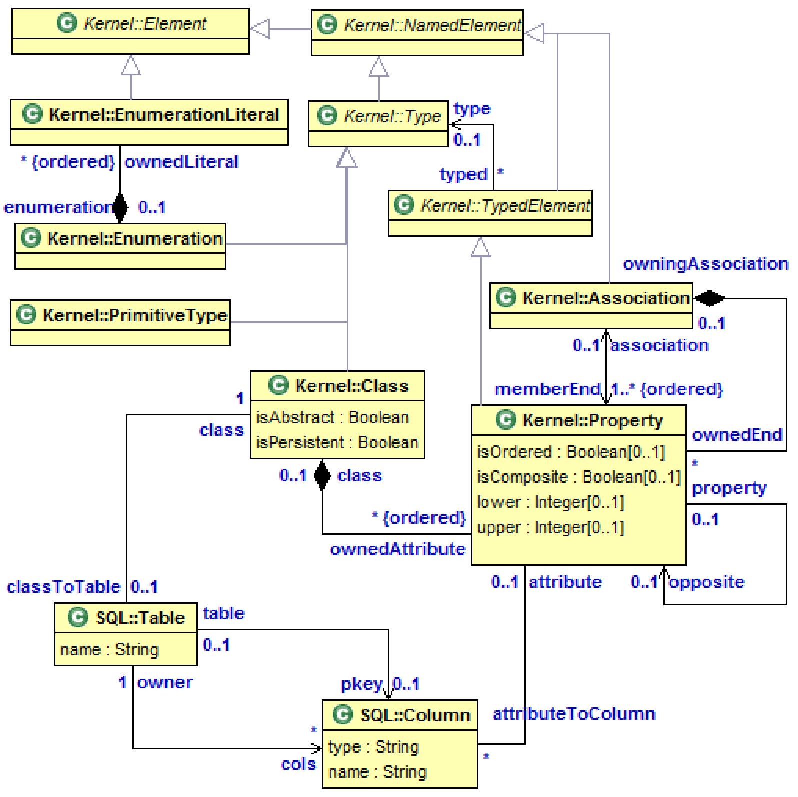


Figure 2.8: Source and target metamodel for the MOLA transformation example, from [27]

The exact transformation to be implemented is the following: "for each persistent class (i.e., `Class` instance) we have to build a `Table` and its primary key `Column` (with a specifically defined name and type `String`). For each attribute of such a class, whose type is a primitive one, we have to build a `Column` in the corresponding `Table` with the same type, but for an attribute with an `Enumeration` type - a `Column` with type `String`. The `Column` name coincides with the attribute name." [27]

The transformation has been implemented using a main procedure called `Main` (depicted in Figure 2.9) and a sub-procedure called `ProcessAttribute` (depicted in Figure 2.10). For easy understanding of the diagrams, here are the most important non-self-explanatory elements in MOLA and what they look like:

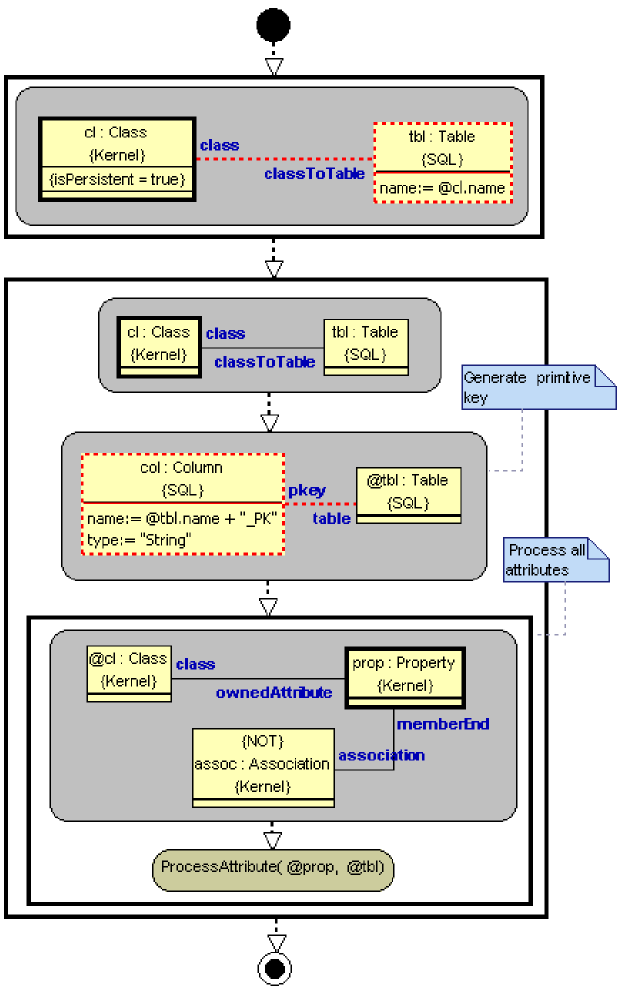


Figure 2.9: MOLA example procedure Main, from [27]

- foreach loop: a rectangle with bold lines
- rule: a grey rounded rectangle

  - pattern part: element with bold black borders
  - action part: associated to the pattern part with a specially coloured/dotted line
    - create action: red dashed borders

- reference to an element ("role") name: @-sign

The association between pattern and action part within a rule is colour- and line-coded like the action part itself. For example, a link that is to be created is represented by a red dashed line. These explanations make the main procedure rather easy to read:

- The foreach loop creates a `Table` element for every `Class` element which has its `is-Persistent` set to true. The `Table`'s `name` property is assigned the value of the `Class`' `name` attribute and the `Class` instance is linked to the new `Table` instance by the `classToTable` link.

- The next element to be executed is another foreach loop. It iterates over all `Class` instances which now already have a `classToTable` relationship to a `Table`. This first rectangle is only the loop head and does not contain an action. Therefore, execution continues with the loop body – the next rule. This rule takes care of the primary key by creating a `Column` element linked to the `Table` instance which has been found in the loop head, assigning appropriate `name` and `type` values.

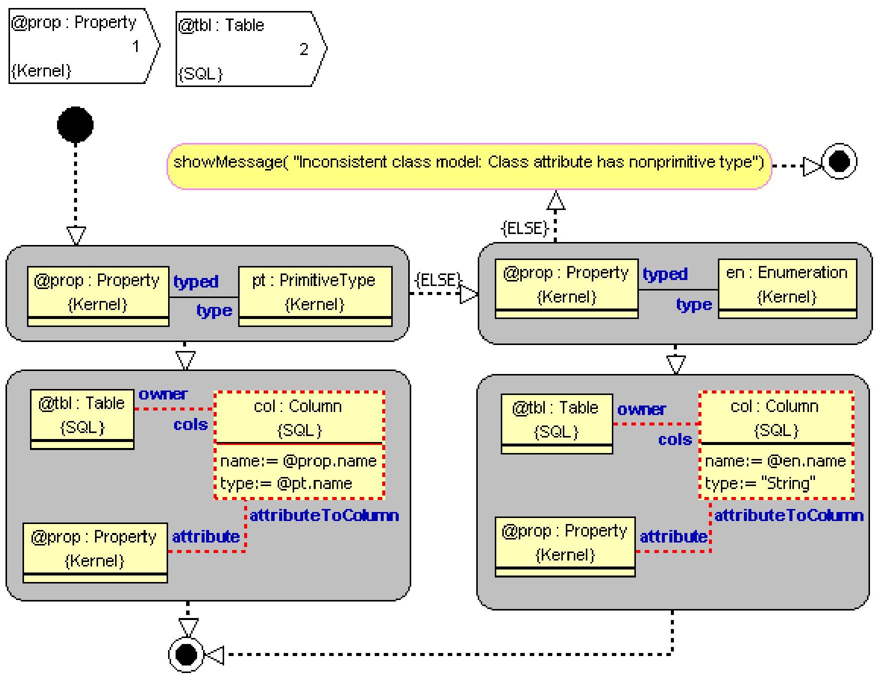

@prop : Property 1 {Kernel}
@tbl : Table 2 {SQL}
showMessage("Inconsistent class model: Class attribute has nonprimitive type")
{ELSE}
@prop : Property {Kernel}
typed
type
pt : PrimitiveType {Kernel}
{ELSE}
@prop : Property {Kernel}
typed
type
en : Enumeration {Kernel}
@tbl : Table {SQL}
owner
cols
col : Column {SQL}
name:= @prop.name
type:= @pt.name
attributeToColumn
@prop : Property {Kernel}
attribute
@tbl : Table {SQL}
owner
cols
col : Column {SQL}
name:= @en.name
type:= "String"
attributeToColumn
@prop : Property {Kernel}
attribute

Figure 2.10: MOLA example procedure ProcessAttribute, from [27]

- The final construct in the `Main` procedure is a nested foreach loop which iterates over the attributes of the current `Class` instance, using the cardinality restraint `NOT` to filter `Property` instances which are association ends. For each identified `Property`, the loop body then calls the sub-procedure `ProcessAttribute` with the current `Property` and `Table` as parameters.

The sub-procedure can be described as follows:

- The two top symbols determine name and number of the parameters given to the procedure.
- The first rule represents an *if*-condition, separating primitive from enumeration attribute types and changing the path of control flow accordingly. If, in the top-right rule, the `Property` is found to be neither of primitive nor of enumeration type, an error message is displayed through the built-in `showMessage` procedure.
- If the `Property` type has been found to be valid, the two bottom rules deal with the creation of the required `Column` elements according to the requirements which have been described above.

A detailed formal description of MOLA can be found in [27].

CHAPTER 3

ReDSeeDS Engine

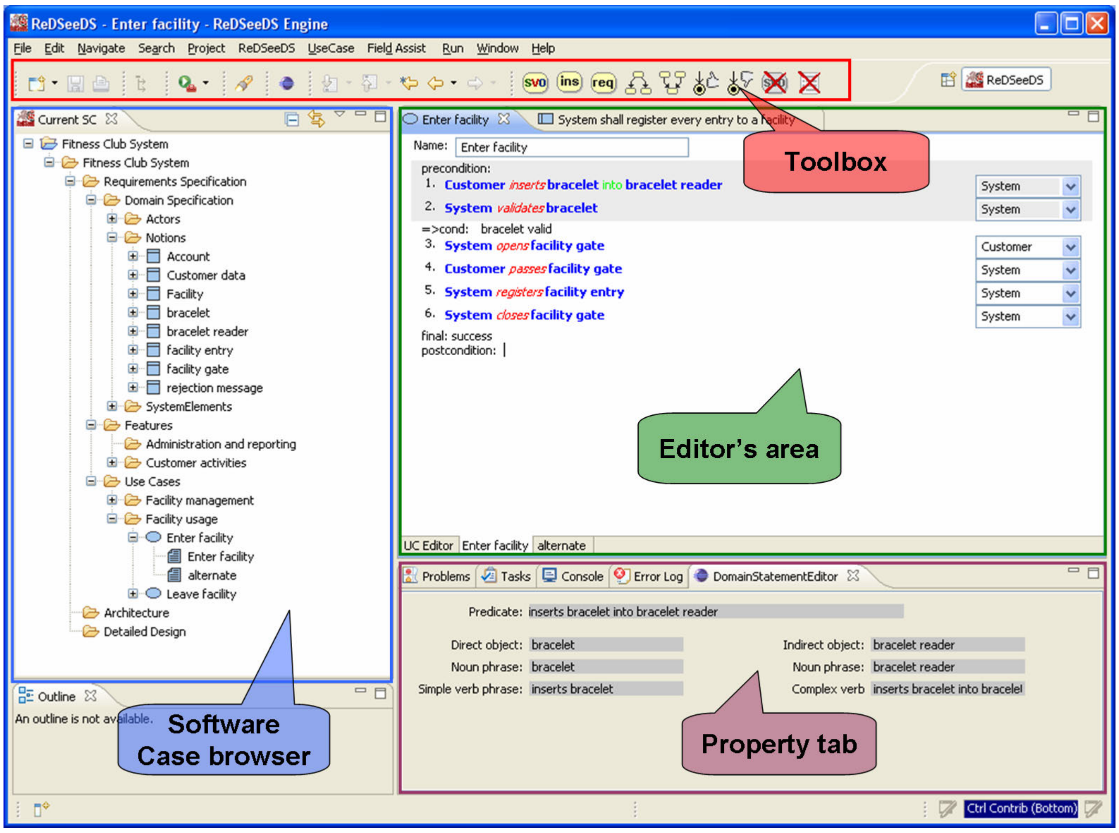
The ReDSeeDS Engine supports requirements-driven, reuse-oriented software development using the ReDSeeDS Software Case Language, which has been summarised in Chapter 2. Through this tool, the ReDSeeDS Software Development Methodology – as described in Chapter 4 – can be applied conveniently. The Engine can be used to create, store, compare and reuse software cases, to specify requirements and terminology and to perform automatic MDD-style transformations between the different stages of development. Additionally, it interfaces with external tools in order for developers to specify the user interface, to view and work on generated architecture and detailed design models and to generate actual code from these models.

The ReDSeeDS Engine is available for download as a fully operational prototype[1] and can be used under open source LGPL terms[2]. It is built on the Eclipse Platform[3] in version 3.3, which is "an open source development platform providing components for creation of standalone applications" [33], in the form of plug-ins which contribute to the user interface of the Eclipse Platform as well as to the underlying business and application logic.

The main screen elements of the ReDSeeDS Engine are annotated in Figure 3.1. On top of the Engine window, a context-sensitive toolbox gives convenient access to important actions. The *Editor Area* is able to show different kinds of editors, for example the *Use Case Editor* as seen in the screenshot. Similarly, the *Property Tab* may show different properties depending on the elements selected in the Editor Area. The *Software Case Browser* on the left is explained in further detail in Figure 3.2. It is used to "browse and manage the structure of current software cases, especially RSL elements like Requirements, Use Cases, Scenarios and SDSL elements" [33].

This chapter further gives an overview over the main parts of the ReDSeeDS Engine: Section 3.1 covers the RSL Editor which is used to specify requirements. Specifying the user interface is described in Section 3.2, which is followed by Section 3.3 and its introduction to the

[1]http://sourceforge.net/projects/redseeds/files/
[2]http://www.gnu.org/licenses/lgpl.html
[3]http://www.eclipse.org/platform/



Name: Enter facility

precondition:
1. Customer *inserts* bracelet into bracelet reader — System
2. System *validates* bracelet — System
=>cond: bracelet valid
3. System *opens* facility gate — Customer
4. Customer *passes* facility gate — System
5. System *registers* facility entry — System
6. System *closes* facility gate — System
final: success
postcondition:

Predicate: inserts bracelet into bracelet reader
Direct object: bracelet
Noun phrase: bracelet
Simple verb phrase: inserts bracelet
Indirect object: bracelet reader
Noun phrase: bracelet reader
Complex verb inserts bracelet into bracelel

Figure 3.1: Main screen elements of the ReDSeeDS Engine, from [33]

SDSL Editor. Transformations from requirements onwards up to code are outlined in Section 3.4 and finally the mechanism for storing and retrieving software cases is detailed in Section 3.5.

3.1 RSL Editor

The RSL Editor actually encompasses five different editor windows, which, however, share similar layout and functionality and enable users to edit a requirements specification element's "description, name and relations to other requirements specification elements" [33]:

- Requirement Editor
- Use Case Editor
- Notion Editor
- Actor Editor
- System Element Editor

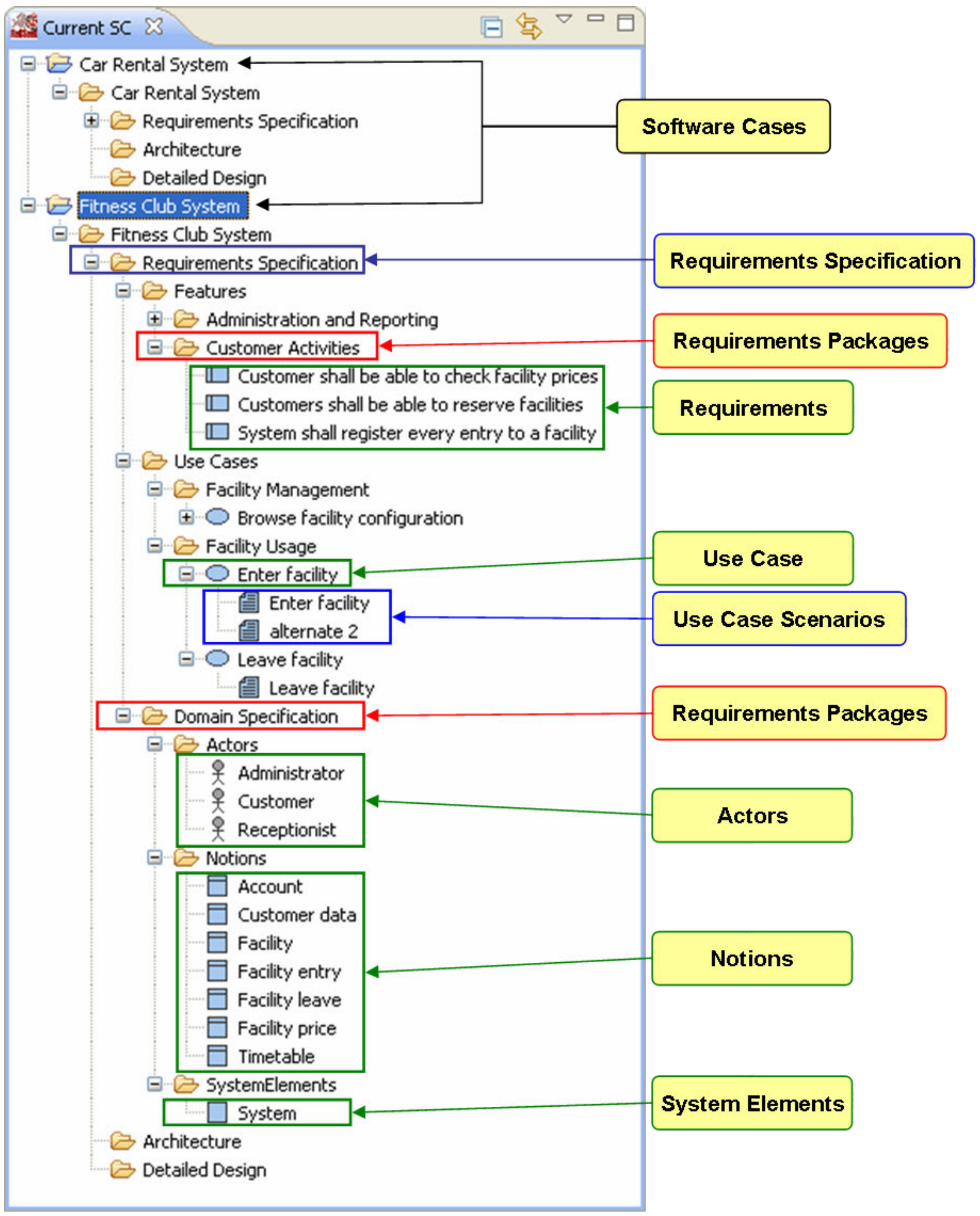
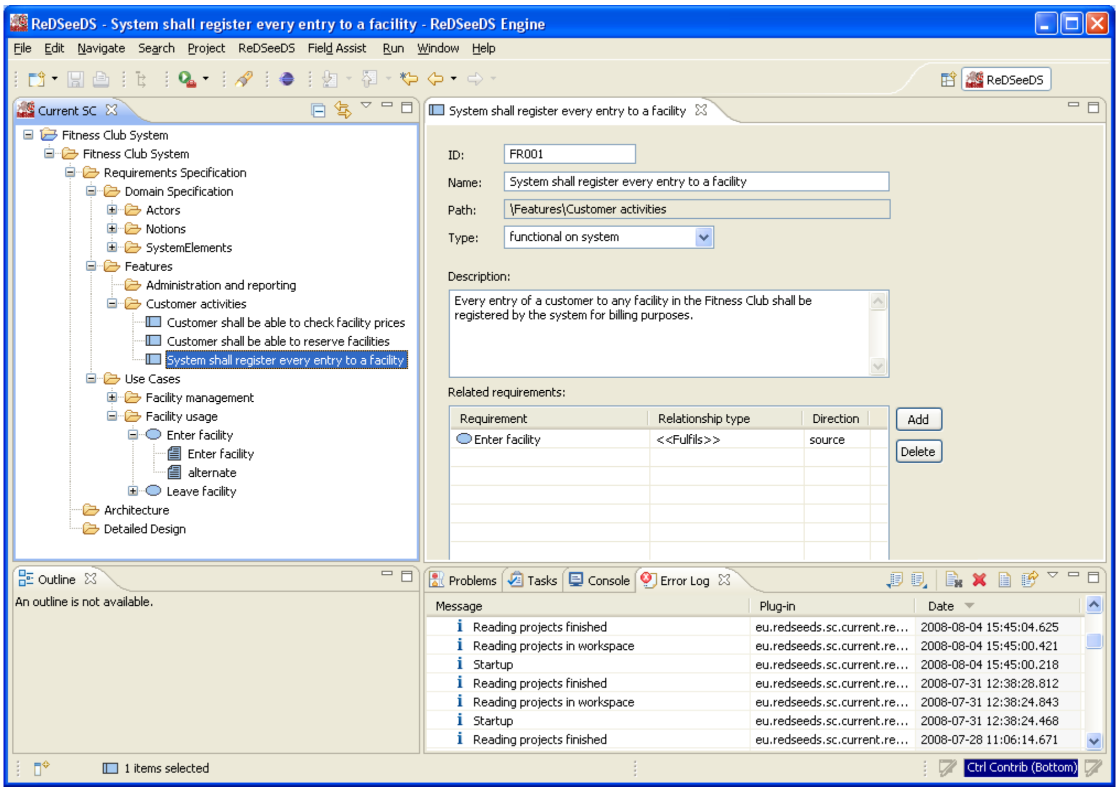


Figure 3.2: Details of the Software Case Browser, from [33]

These editors operate directly on the RSL model within the ReDSeeDS Engine and can be opened by selecting elements from the Software Case Browser described above. It is not possible to go into details for all the different editors and features in this thesis. Thus, the following explanations are only meant as an introduction to the Engine and its "look & feel".

One example screenshot of the *Requirement Editor* can be seen in Figure 3.3. The element which has been selected in the Software Case Browser ("System shall register every entry to a facility") is shown in the Editor Area and can be modified and set into relationship with other elements there.


ReDSeeDS - System shall register every entry to a facility - ReDSeeDS Engine
File Edit Navigate Search Project ReDSeeDS Field Assist Run Window Help
ReDSeeDS
Current SC
Outline
An outline is not available.
1 items selected
Problems Tasks Console Error Log
Add
Delete
Ctrl Contrib (Bottom)

- Fitness Club System
  - Fitness Club System
    - Requirements Specification
      - Domain Specification
        - Actors
        - Notions
        - SystemElements
      - Features
        - Administration and reporting
        - Customer activities
          - Customer shall be able to check facility prices
          - Customer shall be able to reserve facilities
          - System shall register every entry to a facility
      - Use Cases
        - Facility management
        - Facility usage
          - Enter facility
            - Enter facility
            - alternate
          - Leave facility
    - Architecture
    - Detailed Design

System shall register every entry to a facility

ID:	FR001
Name:	System shall register every entry to a facility
Path:	\Features\Customer activities
Type:	functional on system

Description:

Every entry of a customer to any facility in the Fitness Club shall be registered by the system for billing purposes.

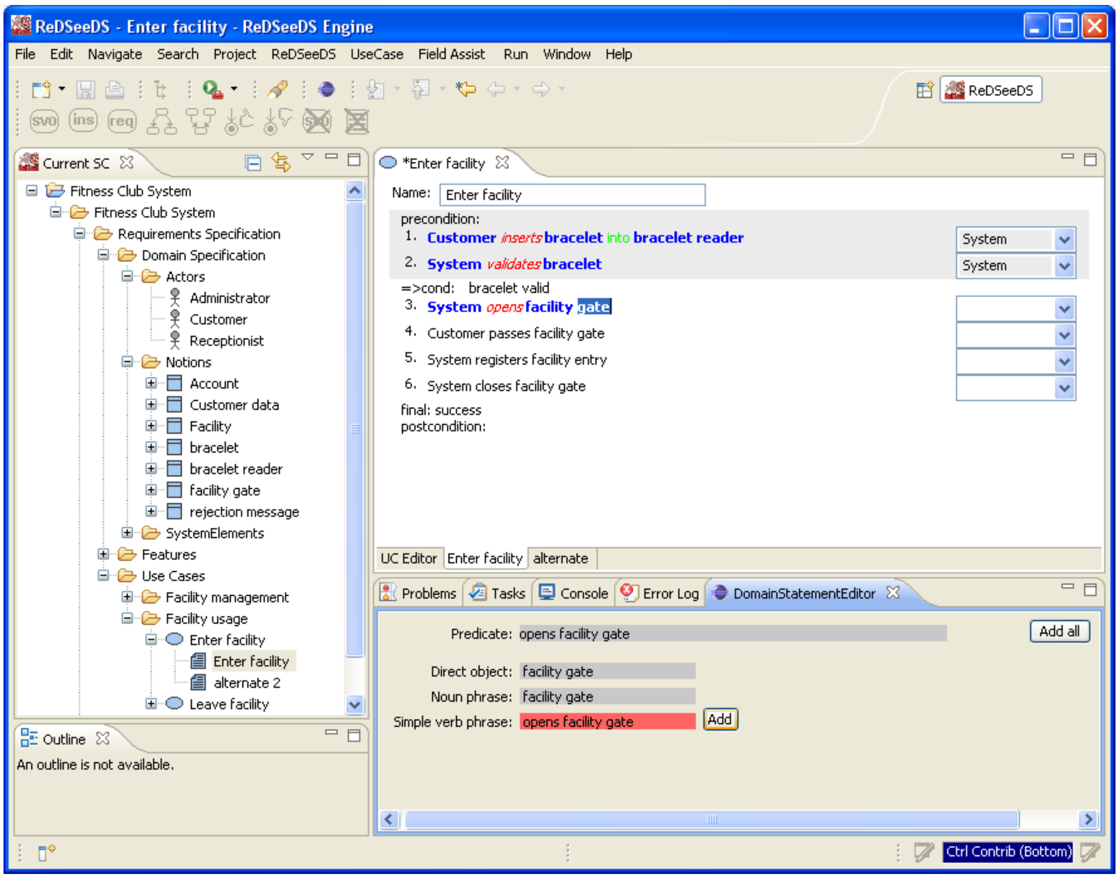
Related requirements:

Requirement	Relationship type	Direction
Enter facility	<<Fulfils>>	source

Message	Plug-in	Date
Reading projects finished	eu.redseeds.sc.current.re...	2008-08-04 15:45:04.625
Reading projects in workspace	eu.redseeds.sc.current.re...	2008-08-04 15:45:00.421
Startup	eu.redseeds.sc.current.re...	2008-08-04 15:45:00.218
Reading projects finished	eu.redseeds.sc.current.re...	2008-07-31 12:38:28.812
Reading projects in workspace	eu.redseeds.sc.current.re...	2008-07-31 12:38:24.843
Startup	eu.redseeds.sc.current.re...	2008-07-31 12:38:24.468
Reading projects finished	eu.redseeds.sc.current.re...	2008-07-28 11:06:14.671

Figure 3.3: Requirement Editor, from [33]

The most sophisticated editor in the ReDSeeDS Engine is the *Use Case Editor* depicted in Figure 3.4. This editor facilitates the easy creation of use cases, in spite of the complex nature of the underlying formal languages, scenarios and relationships. In this example, the use case "Enter facility" is described using a scenario with two pre-conditions and a series of SVO-sentences describing the sequence of actions within this use case. Different colours are used to mark different parts of the sentences (nouns, verbs, modifiers, etc.) and the domain statement tab at the bottom allows creating and editing phrases. Meaning for terms can, however, mostly be determined semi-automatically by consulting the common terminology which is explained in Section 3.5.



An outline is not available.

Name: Enter facility

precondition:
1. Customer inserts bracelet into bracelet reader — System
2. System validates bracelet — System

=>cond: bracelet valid
3. System opens facility gate
4. Customer passes facility gate
5. System registers facility entry
6. System closes facility gate

final: success
postcondition:

Predicate: opens facility gate

Direct object: facility gate
Noun phrase: facility gate
Simple verb phrase: opens facility gate

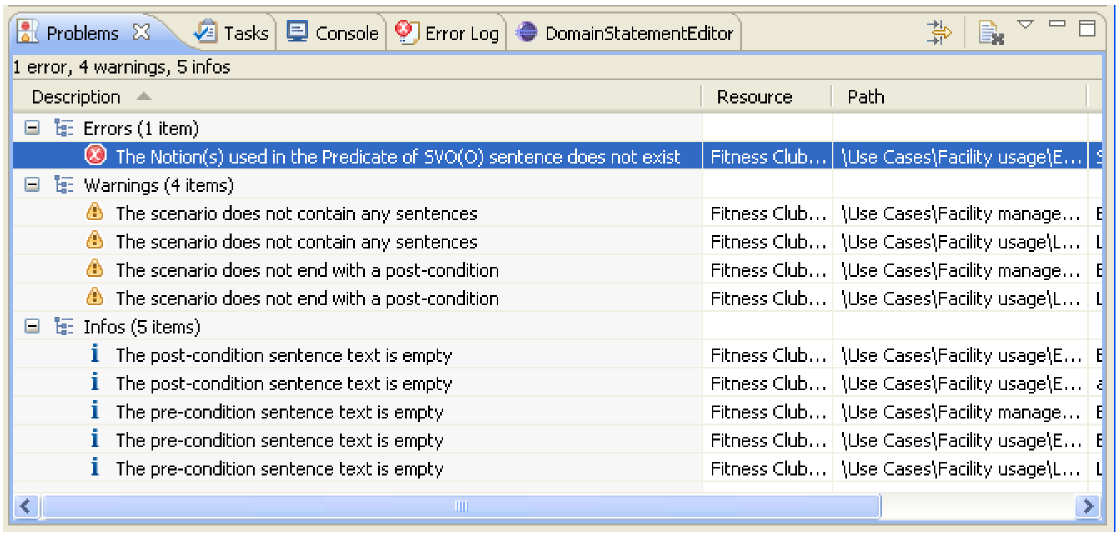
Figure 3.4: Use Case Editor, from [33]

Still, it is possible to write requirements specifications which do not comply with the RSL standard by being inconsistent or incomplete. For this reason, the ReDSeeDS Engine provides mechanisms for verifying any part of the specification by simply selecting a sub-tree within the Software Case Browser and starting the verification process. In case of problems, a list of errors, warnings and additional information is shown as depicted in Figure 3.5.

3.2 UIStoryboard Editor

As mentioned in Section 2.1.3, RSL's Constrained Language Scenarios may be represented as a series of user interface prototypes, called a *UIStoryboard*. This is done with *AMUSEd (Model-based UIStoryboard Editor)*, an application provided by the Fraunhofer research partner. The tool allows easily creating user interfaces and modifying the screen elements' properties. Figure 3.6 shows an example of modelling the relation between two dialogue windows. While the editor can be used for all kinds of projects, it has two special features important for ReDSeeDS:

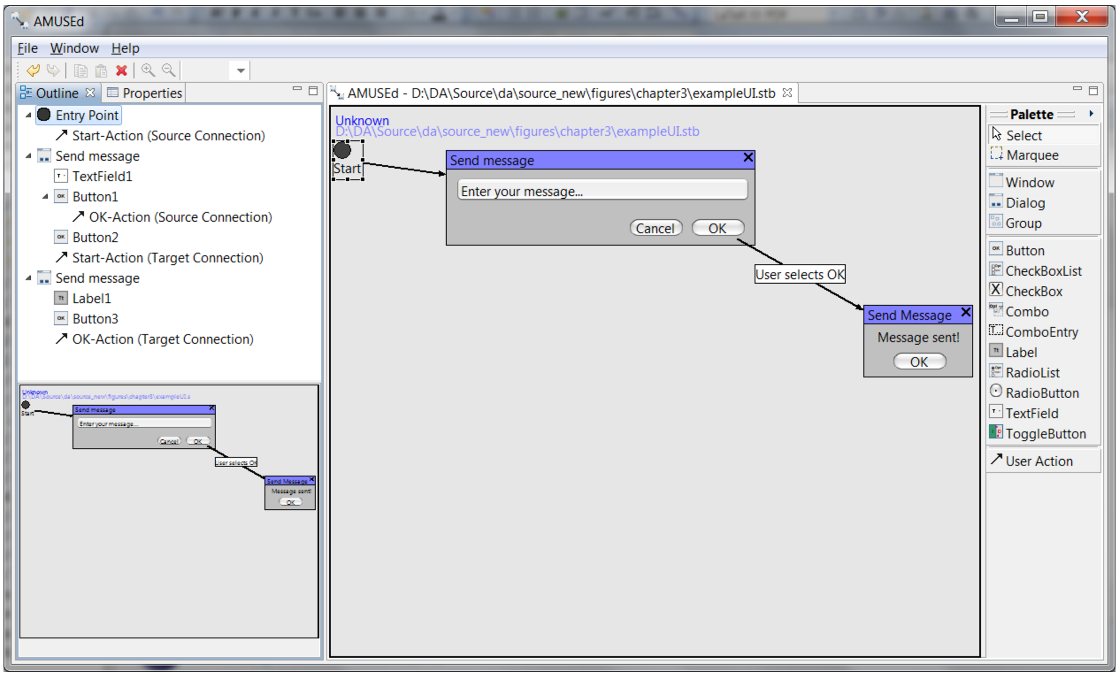
The first special feature is the ability to link UIStoryboards to Constrained Language Scenar-


Problems ☒ | Tasks | Console | Error Log | DomainStatementEditor

1 error, 4 warnings, 5 infos

Description	Resource	Path
⊟ Errors (1 item)		
The Notion(s) used in the Predicate of SVO(O) sentence does not exist	Fitness Club…	\Use Cases\Facility usage\E…
⊟ Warnings (4 items)		
The scenario does not contain any sentences	Fitness Club…	\Use Cases\Facility manage…
The scenario does not contain any sentences	Fitness Club…	\Use Cases\Facility usage\L…
The scenario does not end with a post-condition	Fitness Club…	\Use Cases\Facility manage…
The scenario does not end with a post-condition	Fitness Club…	\Use Cases\Facility usage\L…
⊟ Infos (5 items)		
The post-condition sentence text is empty	Fitness Club…	\Use Cases\Facility usage\E…
The post-condition sentence text is empty	Fitness Club…	\Use Cases\Facility usage\E…
The pre-condition sentence text is empty	Fitness Club…	\Use Cases\Facility manage…
The pre-condition sentence text is empty	Fitness Club…	\Use Cases\Facility usage\E…
The pre-condition sentence text is empty	Fitness Club…	\Use Cases\Facility usage\L…

Figure 3.5: Verification of requirement specifications, from [33]



Entry Point
- Start-Action (Source Connection)

Send message
- TextField1
- Button1
  - OK-Action (Source Connection)
- Button2
- Start-Action (Target Connection)

Send message
- Label1
- Button3
- OK-Action (Target Connection)

Figure 3.6: UIStoryboard Editor

ios. This is necessary in order to define the association formally and let the ReDSeeDS Engine know about it, so that this knowledge can later be used in the model transformations and more sophisticated results are achieved. The second ReDSeeDS-specific feature is automatically generating a Constrained Language Scenario from a UIStoryboard. Using these features does not require any additional effort from the user.

3.3 SDSL Editor

As discussed in Section 2.2, the Software Development Specification Language (SDSL) is a subset of UML. Since there is a number of powerful tools supporting UML, no SDSL Editor has been implemented within the ReDSeeDS Project. Instead, *Enterprise Architect 7.1*[4] is used. This tool, which has been developed by Sparx Systems, "embraces the full product development lifecycle, with high-performance visual tools for business modeling, systems engineering, enterprise architecture, requirements management, software design, code generation, testing and much more", as stated on the company homepage. Despite its powerful features, viewing and editing diagrams is quite easy, even for developers who are not experienced with this specific tool. Figure 3.7 shows an example SDSL interaction diagram opened in Enterprise Architect. The main editor allows the user to model SDSL diagrams and the Project Browser on the right gives an hierarchical overview of all elements within the models.

Architecture and detailed design models can be viewed and modified in Enterprise Architect and then imported back into the current software case. To do this, conversions between the ReDSeeDS TGraph model (see Section 3.5) and the internal Enterprise Architect UML model have been implemented, partly with the model transformation language MOLA.

3.4 Transformations

The ReDSeeDS Engine contains model transformations for the two styles described in Section 2.2. The transformations are specified as a set of MOLA procedures (see Section 2.3) which operate on the SCL metamodel and are compiled to Java code within the Engine. They offer automated steps between the different phases of a software development project and aim to reduce the amount of manual work necessary to progress a software case from requirements to architecture and detailed design.

In the Basic Style, the first transformation to be applied creates an initial Architecture Model in SDSL from the requirements specified in RSL. After possibly modifying the model, the transformation to Detailed Design can be executed.

The Keyword-based Style adds the Analysis Model, which is created by studying notions and scenarios in RSL and represents a conceptual domain model. It has to be created through a MOLA procedure before the Architecture Model can be generated. In this style, analysis of keywords in scenario sentences allows generating a more sophisticated description of system behaviour, provided that the corresponding features of RSL have been used correctly and vastly. The automatically generated Detailed Design Model builds on a Java/Spring/Hibernate

[4]http://www.sparxsystems.com/

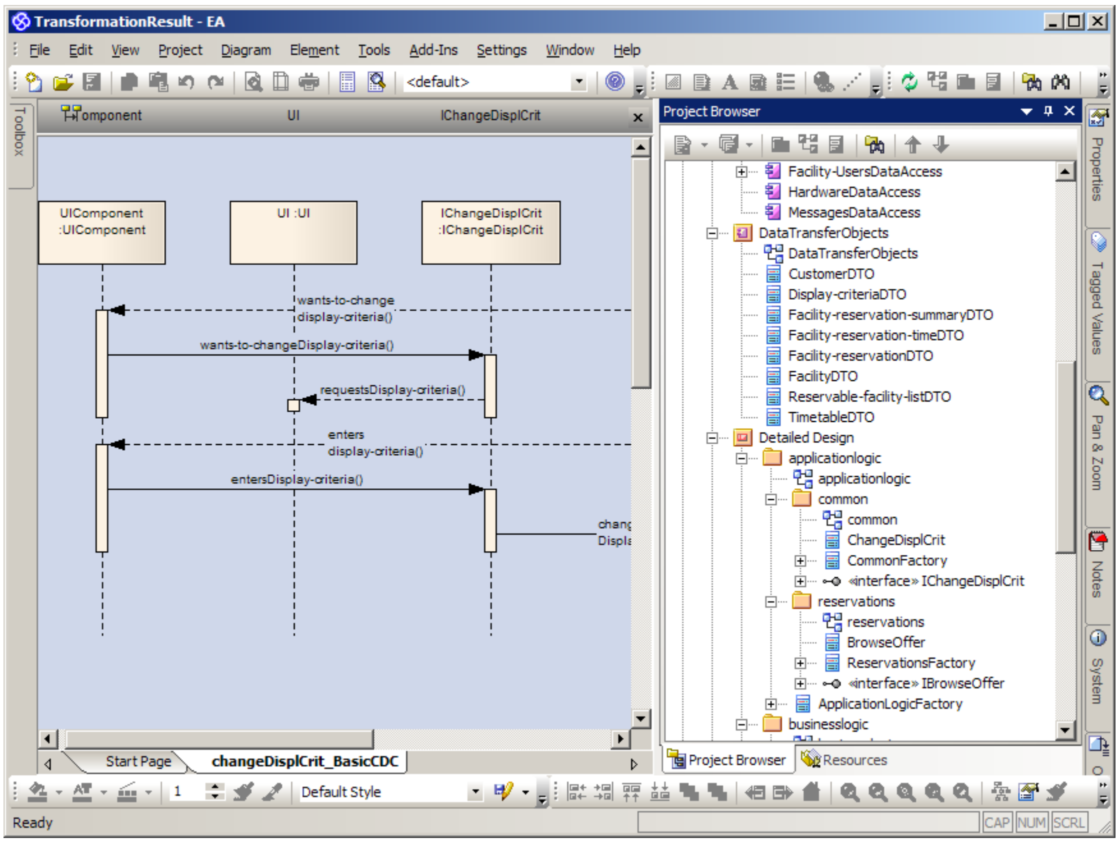


Figure 3.7: Enterprise Architect showing an SDSL diagram, from [33]

framework and allows rich generation of code. This is done via Enterprise Architect's code generation features, which have been extended in order to support all UML stereotypes used in the Keyword-based Style.

An example for another transformation which has been implemented is shown in Figure 3.8, which depicts the visualisation of an RSL use case scenario through a UML activity diagram. It leads from an initial state to two final states and features alternative control flow paths depending on a the fulfilment of a specific condition.

3.5 Software Case Repository and Retrieval

As described in [2], the artefacts of software cases defined using the Software Case Language (SCL, see Chapter 2) are stored as abstract syntax graphs in the ReDSeeDS Software Case Repository. It is based on JGraLab (see [15]) and the SCL metamodel serves as the schema for the underlying TGraph repository (see [10]). Since only abstract syntax graphs are stored, "all parts of SCL that describe only concrete syntax are not related to the fact repository" and "other tools have to be used for creation of requirements, models and code". While the RSL

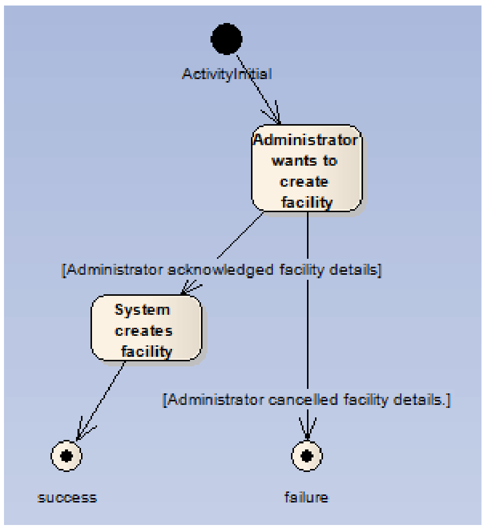

ActivityInitial
Administrator wants to create facility
[Administrator acknowledged facility details]
System creates facility
[Administrator cancelled facility details.]
success
failure

Figure 3.8: Use Case visualisation, from [33]

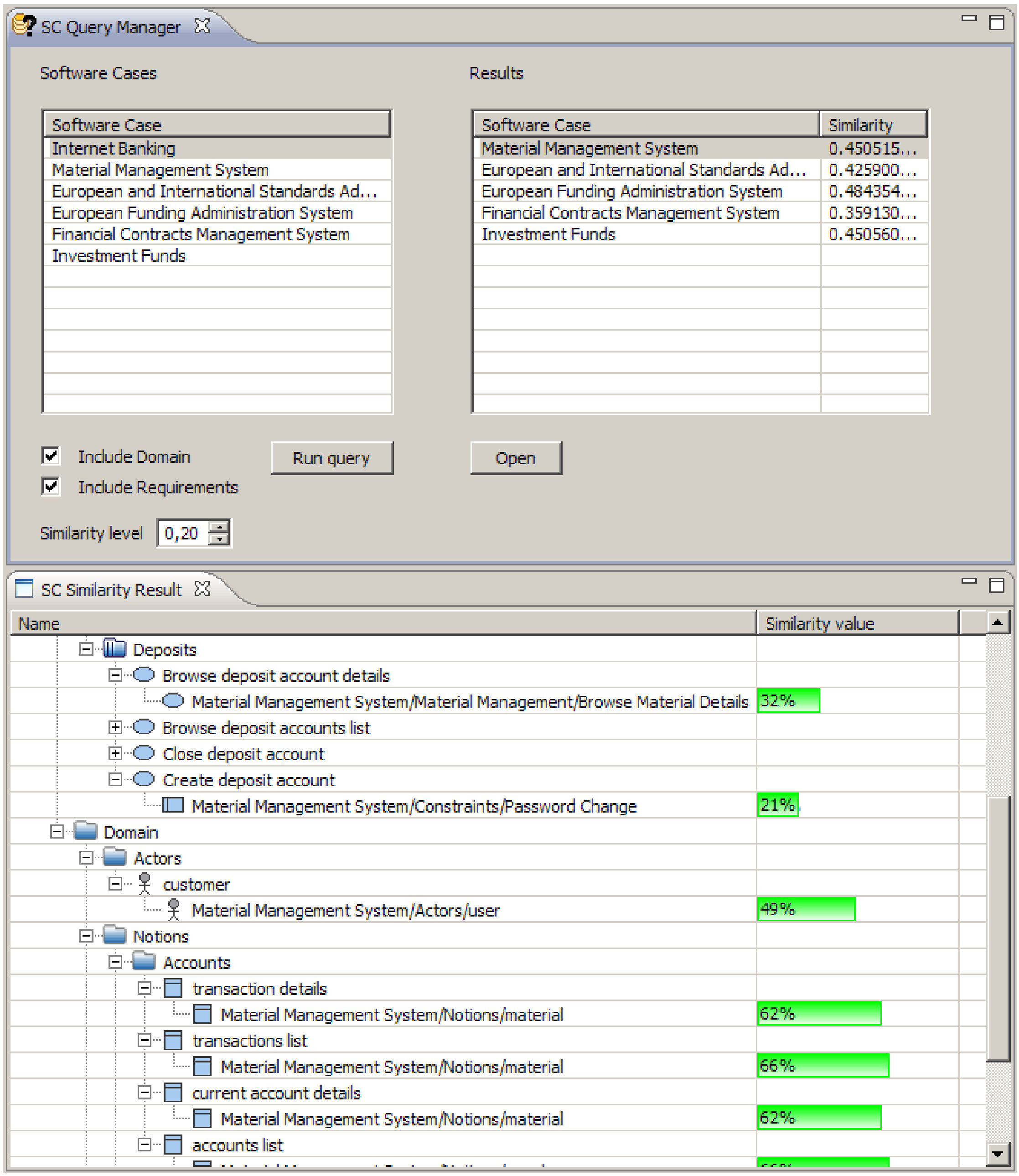
Editor operates directly on the Software Case Repository, an interface between Enterprise Architect's internal UML metamodel and the TGraph-based fact repository has to be used and these "export and import operations have been implemented in ReDSeeDS engine using model transformations in MOLA and function invocations in Enterprise Architect API" [27], as mentioned in Section 3.3. The big advantage of storing complete software cases in one repository is the possibility of "defining traceability links between all layers of a Software Case" [33].

Querying for software cases in the repository is based on similarity calculations and can be done in various ways:

- Comparison of complete cases
- Restriction to the domain part
- Restriction to the requirements part

The Retrieval Engine is accessible through a graphical interface in the ReDSeeDS Engine, as seen in the upper part of Figure 3.9. After selecting the current software case, the parts which are to be compared and a similarity threshold, the query can be run and the results are shown in the upper right list with the calculated grades of similarity between 0.0 and 1.0. Detailed results are displayed in the "SC Similarity Result" tab below in an overview of requirements and domain elements and their similarity values.

The basis of the similarity calculations is the Requirements Model, which is specified using the Requirements Specification Language (RSL, see Section 2.1). The reason for only taking into account this model is that "if two software cases have similar requirements, their other artefacts are similar too and thus, the reuse potential is high" [2]. Since RSL allows developers to specify requirements in different forms, different similarity measures have been implemented and the single similarity results are combined into one final result:


SC Query Manager

Software Cases

Software Case
Internet Banking
Material Management System
European and International Standards Ad...
European Funding Administration System
Financial Contracts Management System
Investment Funds

Results

Software Case	Similarity
Material Management System	0.450515...
European and International Standards Ad...	0.425900...
European Funding Administration System	0.484354...
Financial Contracts Management System	0.359130...
Investment Funds	0.450560...

☑ Include Domain
☑ Include Requirements

Similarity level 0,20

Run query

Open

SC Similarity Result

Name	Similarity value
Deposits	
Browse deposit account details	
Material Management System/Material Management/Browse Material Details	32%
Browse deposit accounts list	
Close deposit account	
Create deposit account	
Material Management System/Constraints/Password Change	21%
Domain	
Actors	
customer	
Material Management System/Actors/user	49%
Notions	
Accounts	
transaction details	
Material Management System/Notions/material	62%
transactions list	
Material Management System/Notions/material	66%
current account details	
Material Management System/Notions/material	62%
accounts list	

Figure 3.9: Software Case Query Manager and visualisation of similarities, from [33]

- Similarity of words based on a thesaurus
- Similarity of natural language based on information retrieval
- Similarity of structured elements and complete cases based on graphs
- Similarity of whole cases based on the ontology language OWL[5]

The “thesaurus” has been implemented in the form of a global terminology generated from the WordNet 3.0 model[6] which contains “the most used words of the English language together with their different senses and relations between them” [33]. The server process for this terminology is delivered along with the ReDSeeDS Engine and can be run locally as well as on a remote machine. For a detailed discussion of the similarity measurement and its implementation, see [2].

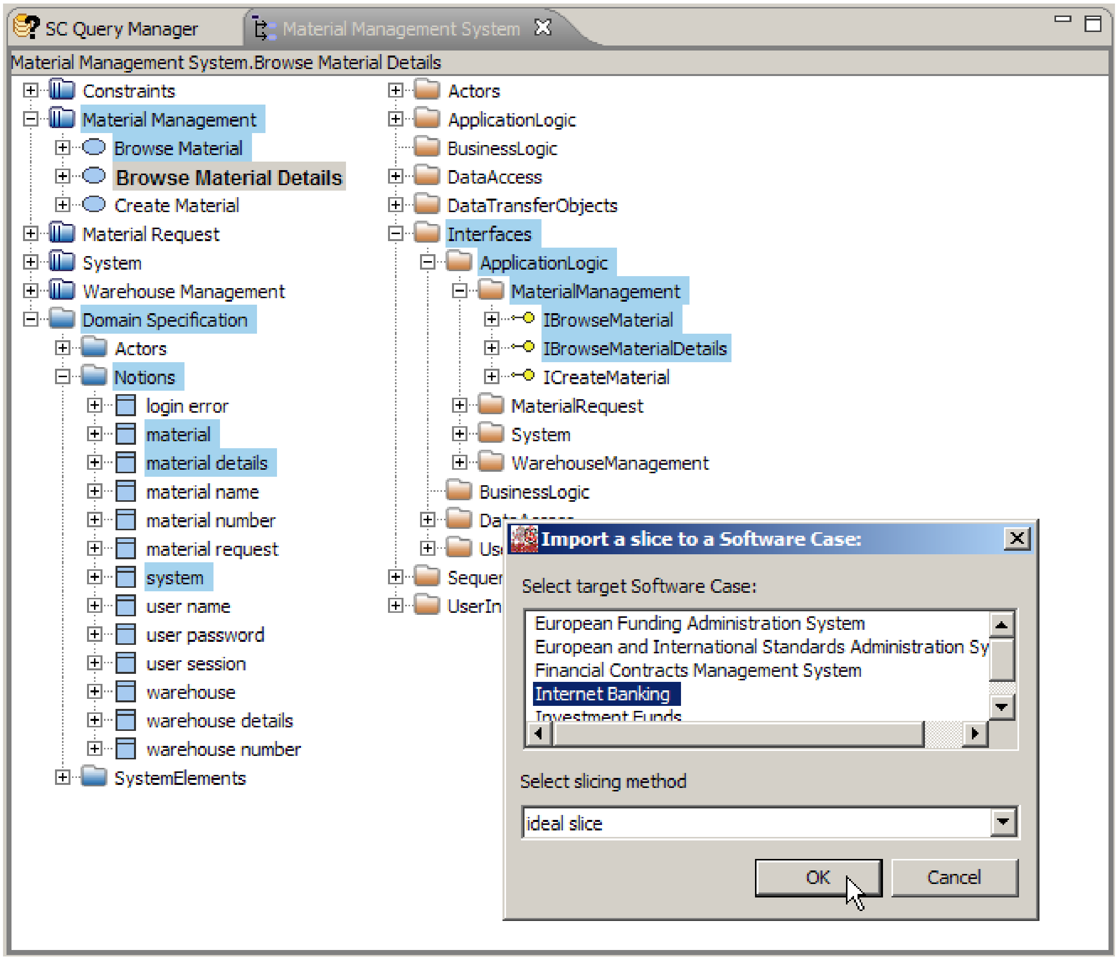
If the user decides to reuse parts of the found software case, the “Open” button will bring up a detailed view of the software case as shown in Figure 3.10. While the ReDSeeDS Engine preselects a set of requirements and domain elements with significant similarity to elements in the current case, manual adjustments are possible. If the developer decides to reuse parts of a former software case, the Engine offers different “slicing methods” to choose from. The underlying idea is that a software case is composed of several possibly overlapping *partial software cases*, which can be computed “on the base of some selected requirements and domain elements, using a mechanism called *slicing*”. The resulting *slices* are constructed by following “the traceability links connecting the different elements of a case” [2]. According to [2], there are four possible types of slices available:

- Minimal Slice: Including only elements which are directly related to the selected requirements and domain elements.
- Maximal Slice: Including all elements in the case which are indirectly related to the selected requirements and domain elements.
- Ideal Slice: Including all elements directly related to the selected requirements and domain elements and elements needed to achieve their proper functionality.
- Domain Including Slice: Including only the domain elements used or referred to by the selected requirement.

After a slice has been calculated, it can be imported into the current software case and semi-automatically merged with the existing elements.

[5]http://www.w3.org/TR/owl-features

[6]http://wordnet.princeton.edu/



Select target Software Case:

European Funding Administration System
European and International Standards Administration Sy
Financial Contracts Management System
Internet Banking
Investment Funds

Select slicing method

ideal slice

Figure 3.10: Selection of elements for reuse, from [2]

CHAPTER 4

ReDSeeDS Software Development Methodology

While the *Software Case Language (SCL)* provides powerful means for specifying software case artefacts and the *ReDSeeDS Engine* and Enterprise Architect allow developers to conveniently create, specify, store and reuse such artefacts, there is still a strong need for "guidance, especially for using the ReDSeeDS language, for model-driven development, for case-driven reuse, or for combinations of these" [39]. This was the main reason for developing the *ReDSeeDS Software Development Methodology* (or short: *ReDSeeDS Methodology*). The author of this thesis was part of the team specifying the Architectural Design Phase.

The ReDSeeDS Methodology consists of three main parts [39]:

- A precise notation – the Software Case Language (SCL),
- A particular software development process and its pretailored variants and
- Techniques with tool support provided by the ReDSeeDS Engine.

This chapter gives a broad overview over the process dimension of the methodology, which explains not only *what* to do with the *Requirements Specification Language (RSL)*, the *Software Development Specification Language (SDSL)*, the transformation language *MOLA* and the *ReDSeeDS Engine*, but also *how* to perform the necessary tasks in order to make full use of ReDSeeD's reuse potential.[1] The "techniques" appear as descriptions of activities which are performed by specific roles within the processes. Due to the methodology's complexity and comprehensiveness, only a superficial summary can be given in this chapter. The full methodology is, however, available for online browsing.[2]

[1]The Software Case Language and the ReDSeeDS tool framework are summarised in Chapters 2 and 3, respectively, and will not be further detailed here.

[2]http://redseeds.sourceforge.net/methodology/

After introducing the processes in Section 4.1, their phases in Section 4.2 and after explaining how SPEM can be visualised in Section 4.3, some details from one of the processes are shown in Section 4.4 in order to convey how the representation actually "looks and feels" like.

4.1 Processes

Figure 4.1 shows the ReDSeeDS software development processes with the arrows pointing towards the more elaborate process variants. The "full" variant includes all ReDSeeDS technologies and artefacts, thus merging the elements of the three "smaller", pre-tailored variants. All processes use the *Slim Process* as their fundament. It is a process template that introduces a certain structure and is used for organisations to reorganise the current practices of their *Organisation-specific Process* in order to be able to adapt the ReDSeeDS processes.

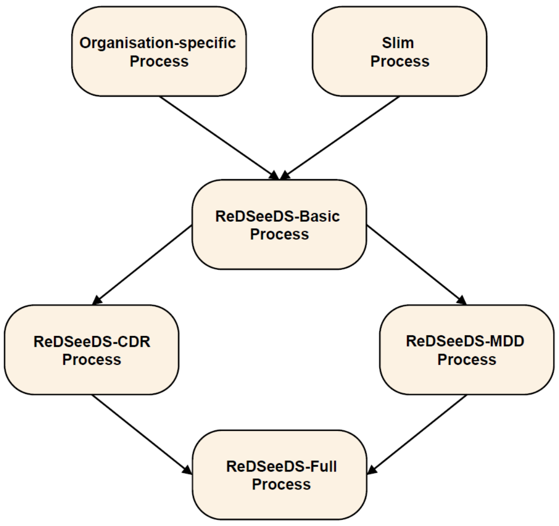


Figure 4.1: Relation between ReDSeeDS software development processes, from [39]

The *ReDSeeDS-Basic Process* lays the foundations for the more complex process variants by introducing RSL (see Section 2.1) for specifying consistent and unambiguous requirements, and SDSL (see Section 2.2) for representing the architecture and detailed design models. The ReDSeeDS Engine assists users in writing the requirements and defining words and phrases used throughout the requirements specification. Additionally, the Basic Process already defines most of the *roles* of the methodology.

Organisations that want to adapt the ReDSeeDS Methodology are "advised to choose carefully one of the four processes and gradually adapt them to reach the ReDSeeDS-Full Process" [39]. Therefore, there are two intermediate process variants. The *ReDSeeDS-CDR Process* adds *Case-driven Reuse* by enabling the (partial) reuse of former software cases with similar requirements specifications, based on precise RSL specifications. The *ReDSeeDS-MDD Process*, on the other hand, focuses on *Model-driven Development* practices by providing transformations from requirements to architecture and on to detailed design, based on RSL and SDSL specifications.

Finally, the *ReDSeeDS-Full Process* “integrates the ReDSeeDS-CDR Process and the ReDSeeDS-MDD Process into one coherent software development process” [39].

4.2 Phases

All the ReDSeeDS processes are “centered around building and reusing software cases, where requirements form the fundament for all activities” [39] and are composed of the following six phases:

1. Requirements Engineering with User Interface Specification
2. Architectural Design
3. Detailed Design
4. Implementation
5. Testing
6. Deployment

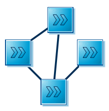
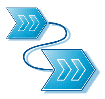
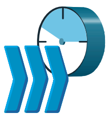
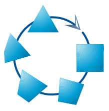
It is up to the developer to decide whether these phases should be executed sequentially or iteratively. It is important to note that the ReDSeeDS Methodology is not concerned with the Implementation, Testing and Deployment phases. For instructions and information about best practices in these phases, developers are referred to existing software development methodologies such as [22] or [29].

4.3 Representation

The ReDSeeDS processes are defined using the *Software Process Engineering Metamodel (SPEM)*, which has been defined by the Object Management Group in [31]. It is a “metamodel that can be used to describe specific software development processes or families of related software development processes” [39] and offers “an object oriented approach using the UML notation” [6]. SPEM focuses on modelling processes by specifying *activities* that are performed on *work products* by certain *process roles* (i.e. requirements engineers, architects, developers, etc.).

For each process, diagrams show the development phases and their activities, work products and so on. Phases are depicted in two views: The *Software Developer’s View* shows information for regular users like requirements engineers and developers, while the *Methodology Manager’s View* contains “diagrams highlighting the differences between the processes for implementing a process within an organisation” [39]. From the initial overview diagrams down to the specific roles, activities and work products, more and more detailed information is added, for example by attaching textual descriptions which precisely explain what to do and how to do it. For a detailed summary of SPEM symbols, see Table 4.1.

Table 4.1: Important SPEM elements, slightly adapted from [39]

	A *work product* is anything created, used or modified by a *process*. Examples for *work products* are *documents*, *models* and source code. A *work product* can be associated with a responsible *role*.
	A *document* is a specific type of work product. *Documents* can be anything that is stored, i.e. ASCII text, Word documents, mind maps, etc.
	A *model* is also a specific type of work product. *Models* are based on a specific meta model, e.g. a UML model or more specifically RSL- or SDSL-based models.
	A *work definition* is a kind of operation that describes the work performed in the *process*. The most important subclass of *work definition* is *activity*. Others are *phase* and *iteration*. A *work definition* can require *work products* as input and can create or modify *work products* as output. Workflow between work definitions is shown by red <preceeds>-arrows: $\longrightarrow$.
	A *process component* is a chunk of process description that is internally consistent and may be reused with other *process components* to assemble a complete *process*.
	A *process package* is basically taken over from the UML *package* element (and thus, does not have a specific SPEM icon). *Process packages* are used to semantically group related elements.
	A *phase* is a specialisation of *work definition* such that its precondition defines the phase entry criteria and its goal defines the phase exit criteria. *Phases* are defined with the additional constraint of sequentiality; that is, their enactments are executed with a series of milestone dates spread over time and often assume minimal (or no) overlap of their activities in time. Workflow between phases is shown by red <preceeds>-arrows: $\longrightarrow$.
	A *process* is an entity that is distinguished from other process entities by the fact that it is not intended to be composed with other entities.

While SPEM models can be viewed in UML editors like Enterprise Architect, there is also the possibility of easily exporting the whole methodology specification to a sequential view of diagrams and textual representations and, more importantly, to an HTML representation, which gives a high level of comfort for reading and following the instructions. Furthermore, activities that have to be executed within the ReDSeeDS Engine can be directly linked from the hypertext representation to the tool, where actions (like opening certain windows) are automatically performed or where the user receives further step-by-step guidelines on how to proceed. This feature is discussed thoroughly in Chapter 5.

4.4 The ReDSeeDS-Full Process

This section gives insight into the ReDSeeDS-Full Process, which is also known as the *Case-driven Software Development (CDSD)* Process, with a focus on the Architectural Design Phase. It is the most elaborate and complete process and, therefore, the most important one. Its life cycle is depicted in Figure 4.2. It consists of the *Requirements Engineering & UI Specification Phase*, the *Architectural Design Phase* and the *Detailed Design Phase*. Other phases are *Implementation*, *Testing* and *Deployment*, but as mentioned in Section 4.2, they are not covered in any further detail by the ReDSeeDS Methodology. As mentioned above, these phases can be executed sequentially or iteratively. The process makes full use of SCL, model-based development and case-driven reuse and, therefore, greatly reduces the amount of manual work necessary in its phases.

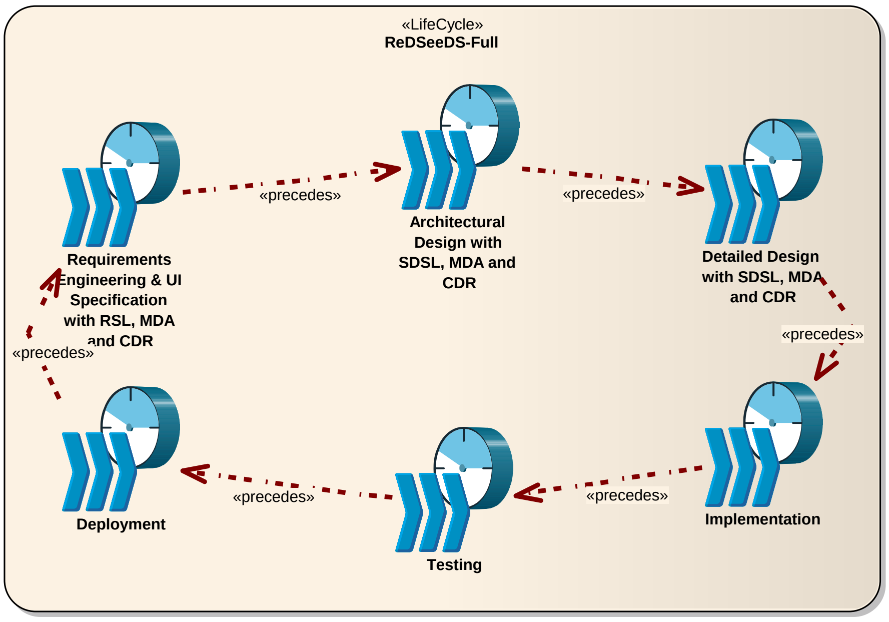

«LifeCycle»
ReDSeeDS-Full
Requirements Engineering & UI Specification with RSL, MDA and CDR
«precedes»
Architectural Design with SDSL, MDA and CDR
«precedes»
Detailed Design with SDSL, MDA and CDR
«precedes»
Implementation
«precedes»
Testing
«precedes»
Deployment
«precedes»

Figure 4.2: Phases of the ReDSeeDS-Full Process, from [39]

The **Requirements Engineering & UI Specification Phase**'s main output is the *Require-*

ments Model, which consists of the domain specification, the requirements specification and the goal specification defined in RSL. There are two ways of identifying requirements: specifying requirements from scratch and reusing existing requirements from former software cases' requirements models (i.e. employing Case-driven Reuse; see [23]). The latter alternative offers the possibility to reuse parts of existing software cases after specifying a query, selecting appropriate elements and modifying and enhancing them. The resulting merged software case allows a refined search for former cases, and so on – existing parts can again be reused and merged into the current Requirements Model. In between these steps, it is important to check the quality of requirements, which includes ensuring their validity and completeness.

The task of specifying the user interface can be performed from scratch and/or by using Case-driven Reuse. In any case, the main idea of the UI prototype is "not to meet high usability standards, but to demonstrate and clarify requirements" [39].

Figure 4.3 shows the overview diagram of the **Architectural Design Phase**. The *Architect* prepares the architectural design by analysing the requirements and creating an architectural vision. Next, he creates a query for retrieving similar software cases based on the requirements model and selects the most appropriate software case of the resulting list for reuse. The selected software case can then be sliced and merged into the current model (see Section 3.5 for an explanation of slicing operations). Afterwards, it is possible to generate a draft architectural model by choosing a transformation specification (i.e. a MOLA program which has been created earlier by the *Transformation Designer*) and executing it on the requirements model. The resulting model can then be modified manually by defining the system components and the system dynamics.

Examining the overview diagram in Figure 4.3, it can be seen that the diagram clearly conveys the main characteristics of the Architectural Design Phase to the audience and is intended to be easily understood and navigated. All elements are linked to more detailed views – work products and roles are associated with descriptions and work definitions lead to sequences of activities which in turn link to detailed instructions on how to execute specific actions. Most of these textual descriptions are interlinked to other elements of the methodology as well. This structure can be exemplified by exploring the work definition which is to be performed by the role *Transformation Designer*, namely *Define Requirements to Architecture Definition*. It links to the diagram shown in Figure 4.4.

This diagram shows which roles are involved in which activities: The *Transformation Designer* performs the three activities – *Specify requirements to architecture transformation rules*, *Write requirements to architecture transformation* and *Test requirements to architecture transformation* – while the Architect assists in the first one. The activities use a temporary artefact called *Requirements to Architecture Transformation Rules* to indicate the knowledge transfer between them and finally result in the *Requirements to Architecture Transformation Specification*, which will later be used in the work definition *Create Architectural Model Using MDA and CDR* (as shown above in Figure 4.3). Following an activity's hyperlink leads the user to the textual description, which potentially links to other methodology elements and might even link directly into the ReDSeeDS Engine (see Chapter 5). For example, clicking on the activity *Test requirements to architecture transformation* will display the information depicted in Figure 4.5.

The **Detailed Design Phase**'s main result is the *Component Design Document* which includes the Detailed Design models written in SDSL. The phase starts with the specification of

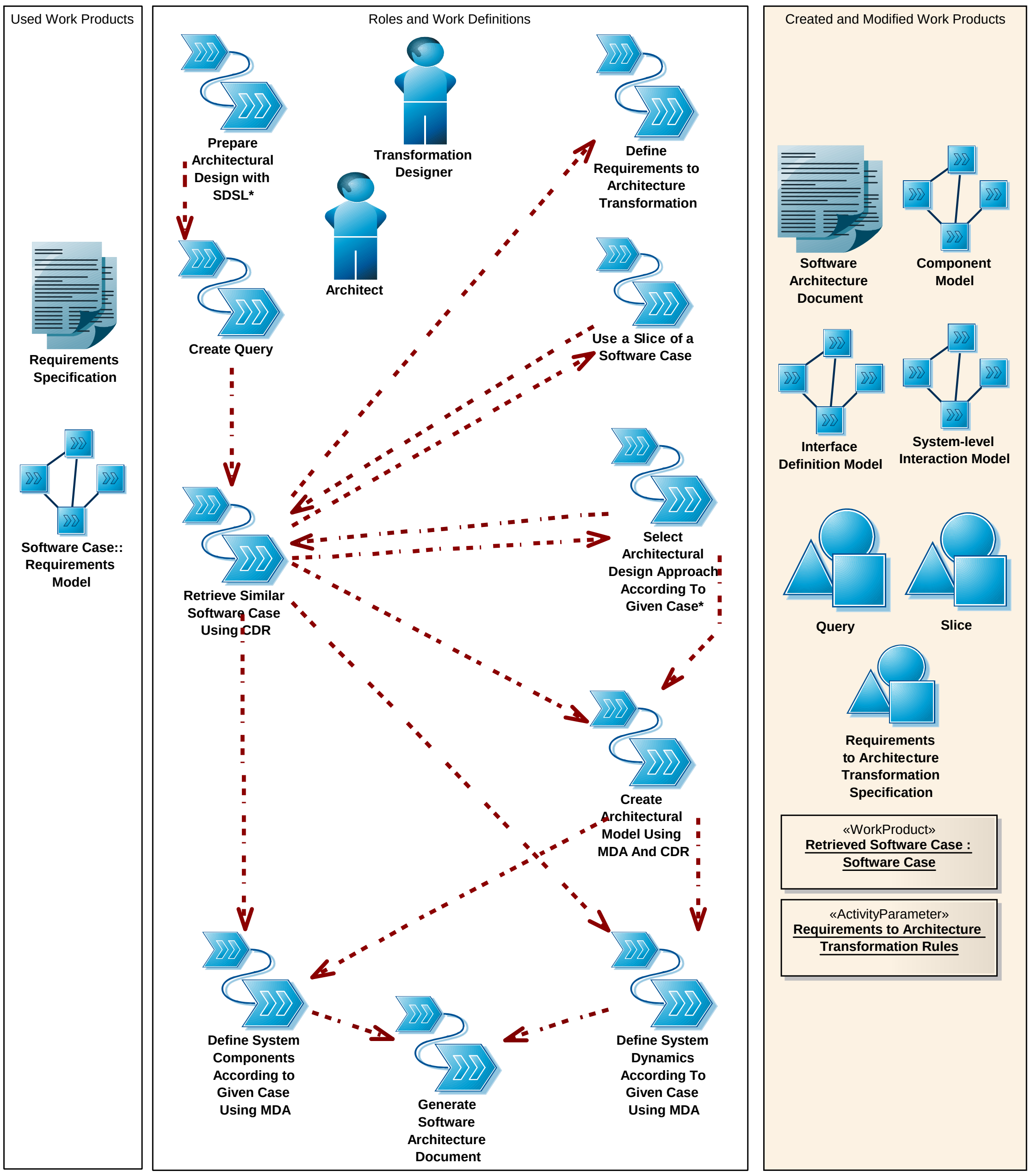
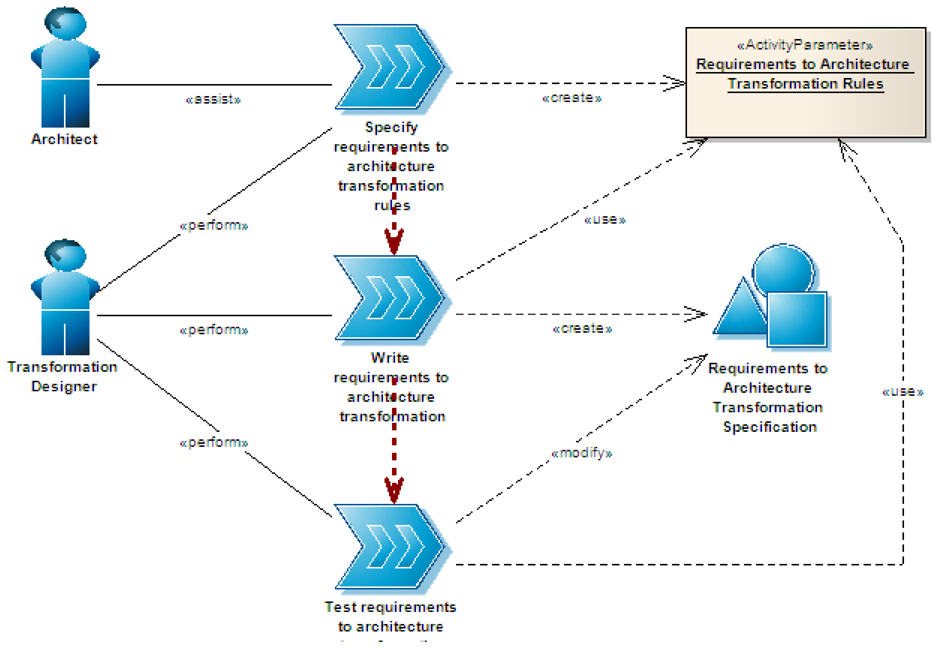
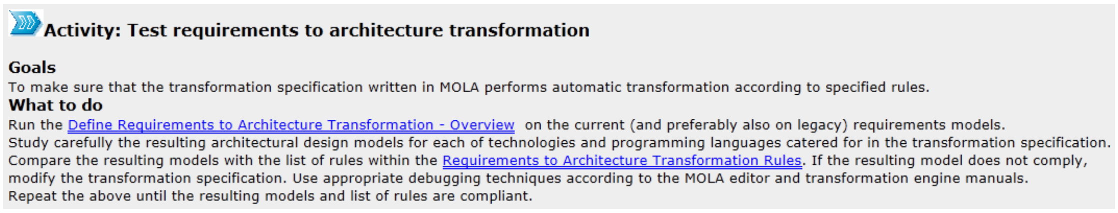

Used Work Products
Roles and Work Definitions
Created and Modified Work Products
Requirements Specification
Software Case:: Requirements Model
Prepare Architectural Design with SDSL*
Transformation Designer
Define Requirements to Architecture Transformation
Architect
Create Query
Use a Slice of a Software Case
Retrieve Similar Software Case Using CDR
Select Architectural Design Approach According To Given Case*
Create Architectural Model Using MDA And CDR
Define System Components According to Given Case Using MDA
Generate Software Architecture Document
Define System Dynamics According To Given Case Using MDA
Software Architecture Document
Component Model
Interface Definition Model
System-level Interaction Model
Query
Slice
Requirements to Architecture Transformation Specification
«WorkProduct» Retrieved Software Case : Software Case
«ActivityParameter» Requirements to Architecture Transformation Rules

Figure 4.3: Roles, work definitions and work products in Architectural Design of the ReDSeeDS-Full Process



Architect
«assist»
Specify requirements to architecture transformation rules
«create»
«ActivityParameter» Requirements to Architecture Transformation Rules
«perform»
«use»
Transformation Designer
«perform»
Write requirements to architecture transformation
«create»
Requirements to Architecture Transformation Specification
«use»
«perform»
«modify»
Test requirements to architecture

Figure 4.4: Work definition *Define Requirements to Architecture Definition*, from the ReDSeeDS Methodology



Activity: Test requirements to architecture transformation

Goals
To make sure that the transformation specification written in MOLA performs automatic transformation according to specified rules.
What to do
Run the Define Requirements to Architecture Transformation - Overview on the current (and preferably also on legacy) requirements models.
Study carefully the resulting architectural design models for each of technologies and programming languages catered for in the transformation specification.
Compare the resulting models with the list of rules within the Requirements to Architecture Transformation Rules. If the resulting model does not comply, modify the transformation specification. Use appropriate debugging techniques according to the MOLA editor and transformation engine manuals.
Repeat the above until the resulting models and list of rules are compliant.

Figure 4.5: Textual description of activity *Test requirements to architecture transformation*, from the ReDSeeDS Methodology

transformation rules in MOLA (see Section 2.3), which are then executed to transform the Architectural Design models into an initial class model and a component-level interaction model. These models are defined with SDSL. Afterwards, the artefacts may be merged with reused parts of a retrieved software case and modified and extended to fulfil “the realisation of these elements of the architectural model that did not get generated automatically (mainly associated with non-functional requirements)” [39].

CHAPTER 5

Coupling Methodology and Engine

The existing gap between the ReDSeeDS Methodology and the ReDSeeDS Engine can be paraphrased by the difference between *primary tasks* and *secondary tasks*, notions which have been coined in [32]. An introduction to this topic is given in [20]:

> [...] a *primary task* is a task that someone wants to (or is supposed to) carry out in the first place, and a *secondary task* is a task of mastering enough of a tool to accomplish a primary task. Since the process only prescribes the primary tasks, a user knowing them still has to figure out how to perform all the required secondary tasks.

With all the information available about the ReDSeeDS Engine and the ReDSeeDS Methodology, as summarised in Chapters 3 and 4, it is still difficult for the end-user to actually apply the methodology with the tool. While the tool at least "hides" the underlying Software Case Language's complexity to the user (see Chapter 2), there has been no clear link between the methodology and the tool.

In the approach to coupling methodology and tool as proposed in this thesis, Web pages contain and convey the primary tasks which are to be done, while so-called *cheat sheets* within the tool present step-to-step instructions on how to perform the secondary tasks necessary to follow the methodology. These two facilities are directly linked to each other by a technical set-up which is described in the following sections. The relationship between primary and secondary tasks is further discussed in [20]:

> None of these secondary tasks would be important per se from the methodological point of view. However, without them the activities prescribed in the method would not become operational with the given tool. Vice versa, without the method descriptions on the higher level, these secondary tasks may lead to artifacts represented in the tool, but not necessarily in the way foreseen by the method.

With the ReDSeeDS Engine developed on top of the Eclipse Platform, it is obvious to make use of the already built-in cheat sheet functionality by integrating it into the Engine. As put in [38], "[t]his will speed up the training curve and make the users feel more comfortable learning about it [...] because the tutorial becomes a part of [the] product." The final aim is that "the Requirements Engineer can work according to the method and its supporting tool, even when not having had experience with either of them" [20].

The workflow for enabling a tight coupling between methodology and tool is depicted in Figure 5.1 and referred to as the *Coupling Workflow*. It distinguishes between the *Development Perspective* which deals with what the developer of a methodology has to do, and the *Runtime Perspective* which shows the processes that are running when an end-user wants to make use of the methodology, the tool and the advantages offered by its coupling.

This chapter first discusses the Development Perspective, i.e. how cheat sheets are designed (Section 5.1.1) and how the methodology can be specified (Section 5.1.2) and prepared (Section 5.1.3). Following that, the Runtime Perspective of the Coupling Workflow is described, i.e. how the Eclipse Platform can be extended (Section 5.2.1), how the cheat sheets are made available to it (Section 5.2.2) and how the Engine can be linked to from within the methodology (Section 5.2.3). Finally, a "real-life example" for this process is given in Section 5.3.

The author of this thesis has contributed to all aspects of the coupling process described in this chapter.

5.1 Development Perspective

5.1.1 Specification of Cheat Sheets

The Eclipse Platform offers a technology called *Cheat Sheets*, distinguishing between *Simple Cheat Sheets* and *Composite Cheat Sheets*. The main idea of Simple Cheat Sheets is to present an interactive list of steps and explanations which are supposed to guide the user through a certain workflow. This is commonly known as a *tutorial*. Composite Cheat Sheets allow cheat sheets to be grouped in order to build complex, modular tutorials. They will, however, not be discussed in detail here because they are not that important for the basic idea of coupling an activity within a methodology with an external tool.

According to [8], cheat sheets are an ideal way to realise tutorials which lead the user step by step through a series of work items. In the context of coupling a methodology and a tool as it is discussed in this chapter, cheat sheets are used to convey *secondary tasks* of an activity to the users – that is, "*how* to do what the method instructs them to" [20].

It is the Developer's task to design cheat sheets, as depicted in Figure 5.2.

Cheat sheets are defined as XML documents. While this allows developers to create cheat sheets with any text editor they like, Eclipse 3.3 also contains a graphical *Cheat Sheet Editor* which makes the specification of cheat sheets a lot easier and more convenient. Therefore, although Eclipse is not necessary for this workflow and not depicted in Figure 5.2, the next paragraphs will give an overview over the creation of cheat sheets with Eclipse.

- To create a new cheat sheet, the user has to navigate to the *File > New > Other...* menu item and then select the appropriate template, as shown in Figure 5.3.

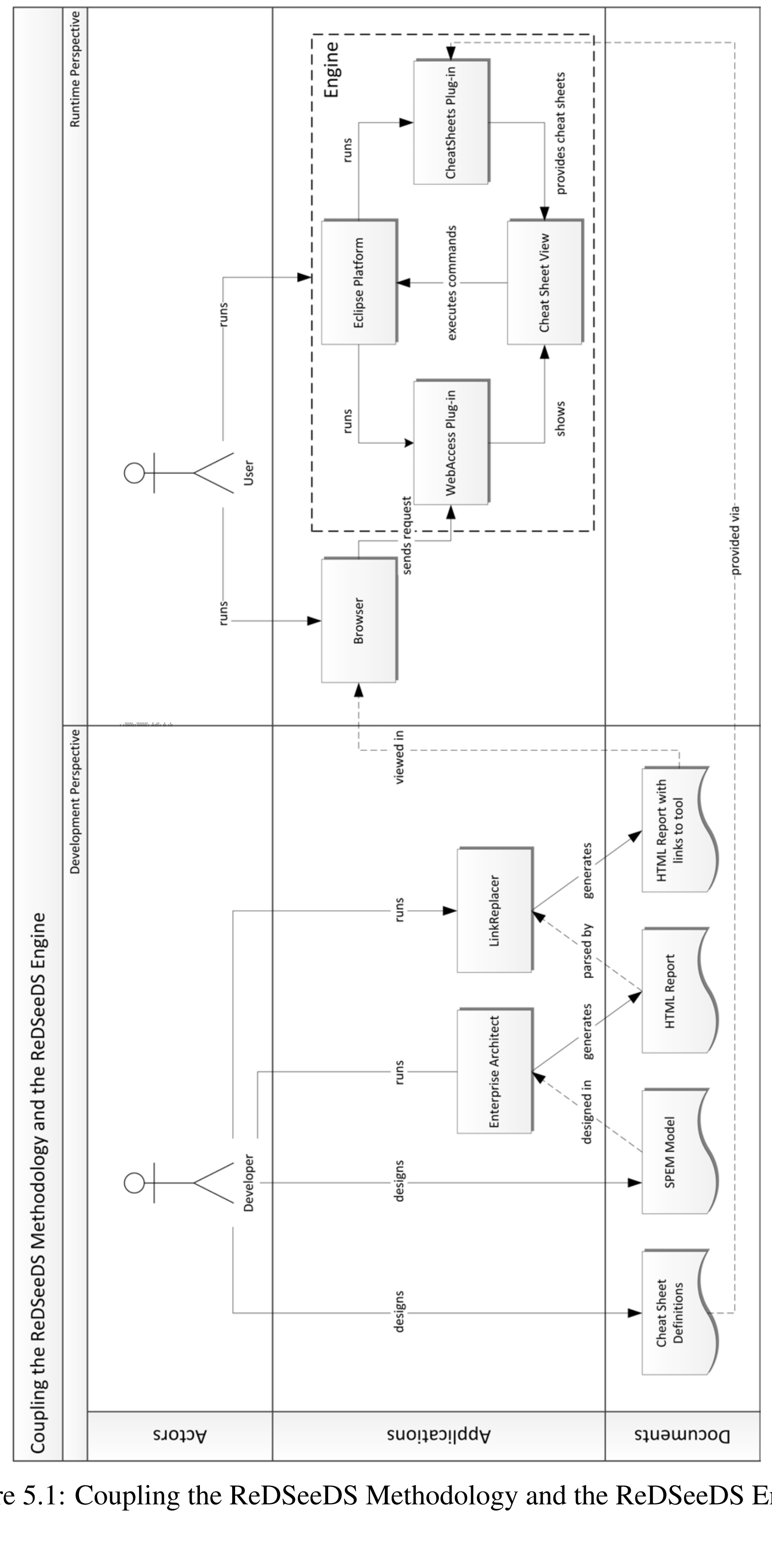


Figure 5.1: Coupling the ReDSeeDS Methodology and the ReDSeeDS Engine

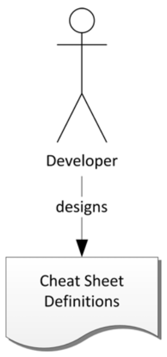


Figure 5.2: Workflow *Specification of Cheat Sheets*

- After the user has entered a filename for the new cheat sheet, Eclipse opens the Cheat Sheet Editor with some example contents as seen in Figure 5.4.
- Every Simple Cheat Sheet consists of an introduction text and a number of items ("steps") which can have sub-elements. Adding and modifying steps is simple but the Eclipse help pages provide additional information if necessary.

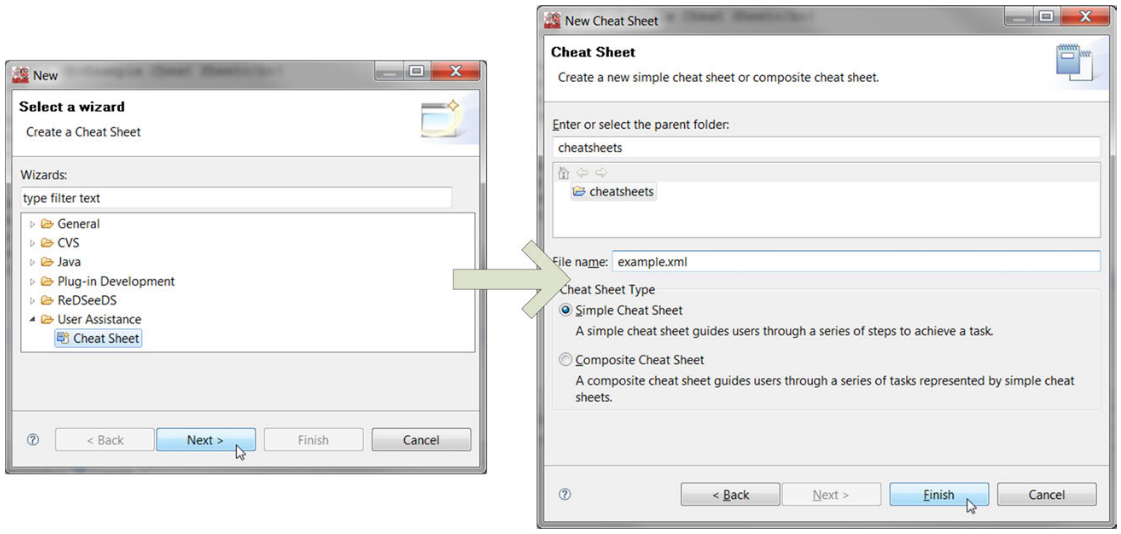
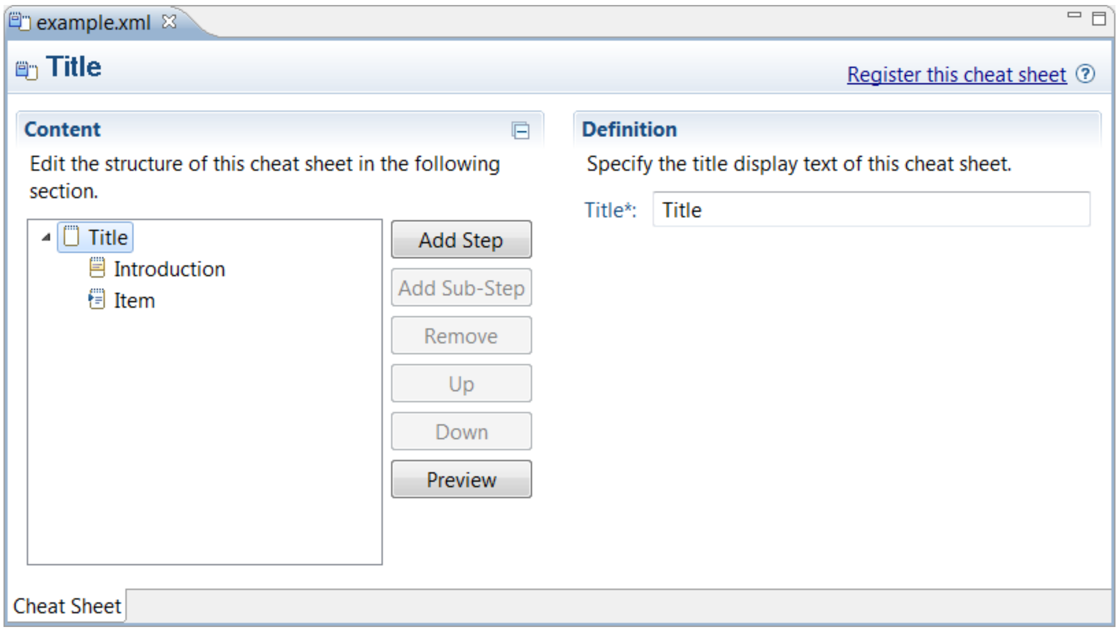


Figure 5.3: Creating a new cheat sheet

Cheat sheets, or more specifically the XML documents in which they are specified, are processed by Eclipse and graphically displayed as a series of steps which can be completed one by one. Providing a set of cheat sheets along with a tool built on the Eclipse Platform requires the XML files to be packaged as described in Section 5.2.2. For testing purposes, however, cheat sheets can also be opened and viewed individually via the preview functionality of the Cheat Sheet Editor or through the menu item *Cheat Sheets...* in the *Help* menu, as shown in Figure 5.5.


example.xml

Title

Register this cheat sheet

Content

Edit the structure of this cheat sheet in the following section.

- Title
  - Introduction
  - Item

Add Step | Add Sub-Step | Remove | Up | Down | Preview

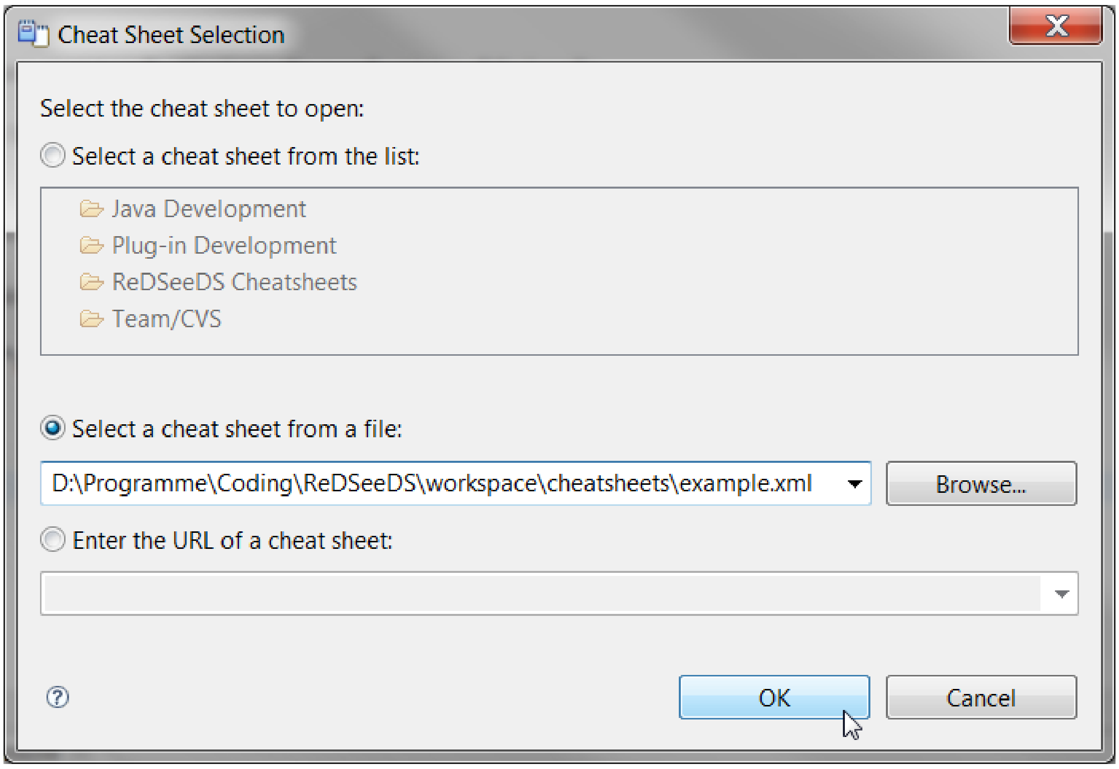
Definition

Specify the title display text of this cheat sheet.

Title*: Title

Cheat Sheet

Figure 5.4: The Eclipse Cheat Sheet Editor after creating a new cheat sheet


Cheat Sheet Selection

Select the cheat sheet to open:

( ) Select a cheat sheet from the list:

- Java Development
- Plug-in Development
- ReDSeeDS Cheatsheets
- Team/CVS

(•) Select a cheat sheet from a file:

D:\Programme\Coding\ReDSeeDS\workspace\cheatsheets\example.xml

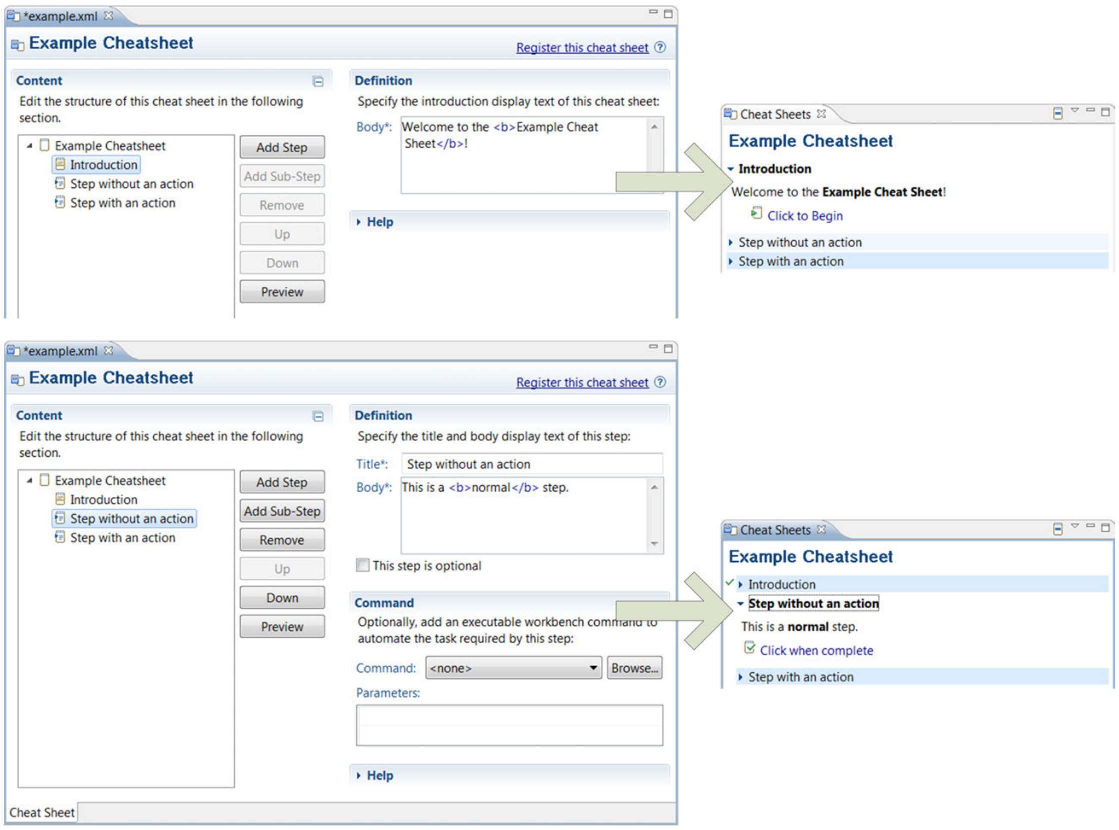
Browse...

( ) Enter the URL of a cheat sheet:

OK | Cancel

Figure 5.5: Opening a cheat sheet directly

Figure 5.6 shows what cheat sheets look like when they are shown to the end-user. Navigation through cheat sheets is intended to be self-explanatory and intuitive to users. Additionally, the figure shows that the *body* field may contain simple HTML tags like <b> for rendering text bold.


Specify the introduction display text of this cheat sheet:
Body*: Welcome to the <b>Example Cheat Sheet</b>!

Welcome to the **Example Cheat Sheet**!

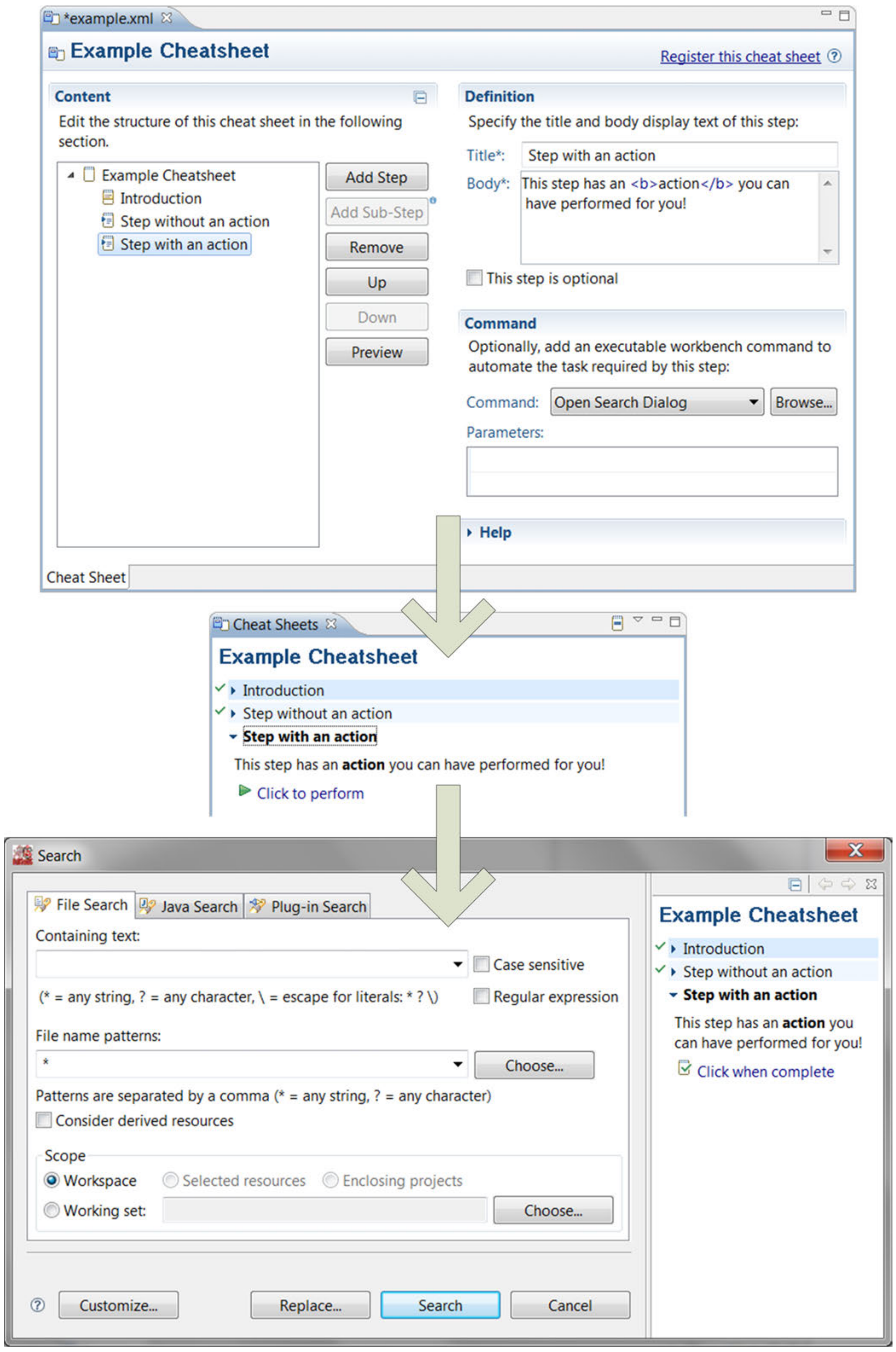
Specify the title and body display text of this step:
Title*: Step without an action
Body*: This is a <b>normal</b> step.
This step is optional
Optionally, add an executable workbench command to automate the task required by this step:
Command: <none>

This is a **normal** step.

Figure 5.6: Cheat sheets as viewed in the Cheat Sheet Editor and in the Cheat Sheet View

A powerful feature of cheat sheets is the possibility to invoke actions of the tool, i.e. to have actions performed automatically for the user. This is required because "even with such advice as given in the cheat sheet, it is not always easy to find everything in the tool and to do it right in order to operationalize this advice" [20]. If a step is connected to a specific action, the user is shown a "Click to perform" link within the cheat sheet which, after being clicked, executes the action. As stated in [21], it is a good idea to have the tool perform some steps on behalf of the user. This feature reduces users' workload and allows them to focus on the important aspects of the tutorial. And seen in the context of coupling a methodology with a tool, it enables even tighter coupling, not only from the methodology to the surface of the tool (i.e. the cheat sheets), but actually even *into* the functionality of the tool itself.

Figure 5.7 demonstrates a cheat sheet item which has been linked to a command which shows the built-in Search dialogue. As soon as the user clicks the "Click to perform" link, the dialogue is opened automatically and the Cheat Sheet View is integrated into the new window.


*example.xml

Example Cheatsheet

Register this cheat sheet

Content

Edit the structure of this cheat sheet in the following section.

- Example Cheatsheet
  - Introduction
  - Step without an action
  - Step with an action

Definition

Specify the title and body display text of this step:

Title*: Step with an action

Body*: This step has an <b>action</b> you can have performed for you!

This step is optional

Command

Optionally, add an executable workbench command to automate the task required by this step:

Command: Open Search Dialog

Parameters:

Cheat Sheets

Example Cheatsheet

- ✓ Introduction
- ✓ Step without an action
- Step with an action

This step has an **action** you can have performed for you!

Click to perform

Search

Containing text:

(* = any string, ? = any character, \ = escape for literals: * ? \)

File name patterns: *

Patterns are separated by a comma (* = any string, ? = any character)

Consider derived resources

Scope: [✓ Workspace] Selected resources Enclosing projects Working set:

Example Cheatsheet

- ✓ Introduction
- ✓ Step without an action
- Step with an action

This step has an **action** you can have performed for you!

Click when complete

Figure 5.7: A cheat sheet item featuring a command to be performed automatically

Cheat sheet steps can be linked to all commands which provide so-called "Action" classes, as it is the case with most menu and toolbar items, for example. Extra work by the cheat sheet designer is only necessary for "cheat-sheet specific actions and actions that have to be aware of the cheat sheet's state". In this case, "the method-tool coupling developer has to program corresponding Action classes that orchestrate all necessary tool-specific actions or method calls" [20].

As mentioned above, cheat sheets are actually XML documents which can in theory be written textually as well. Eclipse does not save any meta-information along with cheat sheets, so that Listing 5.1 is fully equivalent to the graphically designed cheat sheet as shown in Figures 5.6 and 5.7.

Listing 5.1: Example cheat sheet XML source

```
<?xml version="1.0" encoding="UTF-8"?>
<cheatsheet title="Example Cheatsheet">
   <intro>
      <description>
         Welcome to the <b>Example Cheat Sheet</b>!
      </description>
   </intro>
   <item title="Step without an action" dialog="true" skip="false">
      <description>
         This is a <b>normal</b> step.
      </description>
   </item>
   <item title="Step with an action" dialog="true" skip="false">
      <description>
         This step has an <b>action</b> you can have performed for you!
      </description>
      <command serialization="org.eclipse.search.ui.openSearchDialog" confirm="
          false">
      </command>
   </item>
</cheatsheet>
```

For more information on cheat sheets, authoring guidelines and advanced features like conditional control flow, see the Eclipse Help and [38].

5.1.2 Creation and Representation of the Methodology Model

While ReDSeeDS invokes Enterprise Architect as its SDSL Editor (see Section 3.3), the developers of the ReDSeeDS Methodology (see Chapter 4) also used it to model the methodology and its processes and to generate a hypertext representation of the model. This part of the Coupling Workflow, which focuses on the developer's work in Enterprise Architect, is shown in Figure 5.8.

Specifying the Methodology

As described in Section 4.3, the *Software Process Engineering Metamodel (SPEM)* has been used to model the methodology. Enterprise Architect does, however, not support SPEM by itself. Instead, since SPEM is an extension to UML, the tool allows importing a SPEM Profile which

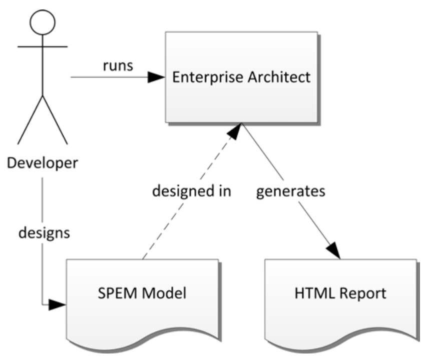
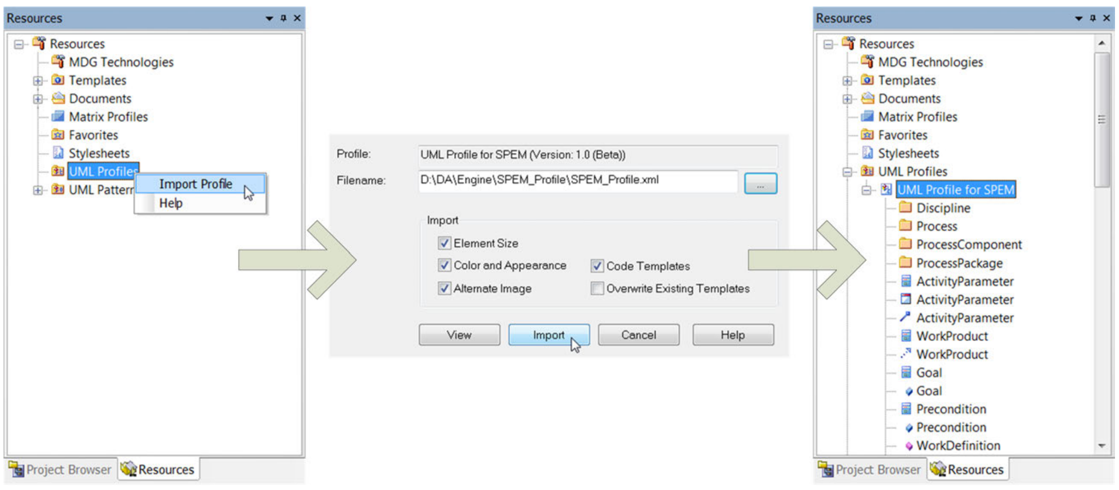


Figure 5.8: Workflow *Creation and Representation of the Methodology Model*

is available for download from the Sparx Systems homepage.[1] This profile defines a number of UML stereotypes which allow developers to conveniently and graphically design software development processes within Enterprise Architect.

After downloading and extracting `SPEM_Profile.zip`, the profile can be made available by importing `SPEM_Profile.xml` via the *Resources* tab, as shown in Figure 5.9. The various SPEM elements can then be shown in the *Toolbox* frame and simply be dragged and dropped onto the model design area. For more information about SPEM itself, see [31].



Profile: UML Profile for SPEM (Version: 1.0 (Beta))
Filename: D:\DA\Engine\SPEM_Profile\SPEM_Profile.xml

Import
- [✓] Element Size
- [✓] Color and Appearance
- [✓] Code Templates
- [✓] Alternate Image
- [ ] Overwrite Existing Templates

Figure 5.9: Importing the SPEM Profile into Enterprise Architect

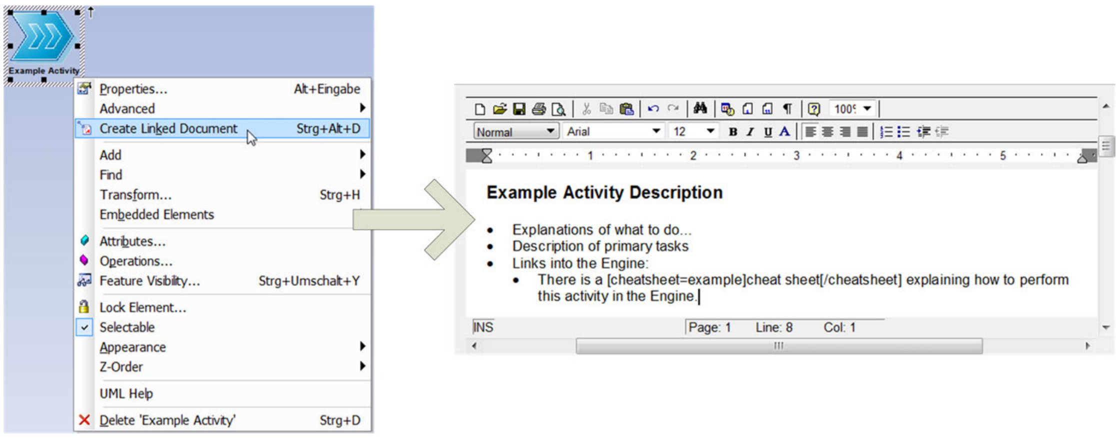
Adding Textual Descriptions

At this point in the presented workflow, activities in the designed software engineering process are described by their name and their relation to activities, actors, work definitions and other

[1]http://www.sparxsystems.com/resources/developers/spem_profile.html

elements within the model. In order to list the *primary tasks* necessary for performing an activity, however, a textual description has to be added which can then be read by the user of the methodology. This is done by attaching *Linked Documents* to elements within the model. While, usually, these documents will be linked to activities, they can also be attached to other elements, for example to give detailed information about a role or a work product.

Linked Documents can be edited directly within Enterprise Architect. Since the contents are specified in *Rich Text Format (RTF)*, other tools like Microsoft Word can be used to edit and format them as well. Figure 5.10 shows how to add a textual description to an example SPEM activity.



Example Activity Description

- Explanations of what to do...
- Description of primary tasks
- Links into the Engine:
  - There is a [cheatsheet=example]cheat sheet[/cheatsheet] explaining how to perform this activity in the Engine.

Figure 5.10: Attaching a *Linked Document* to a SPEM activity

This screenshot also shows one of the main aspects of the coupling between the ReDSeeDS Methodology and the Engine, namely putting *symbolic cheat sheet links* into linked documents as it has been done in this sentence:

```
There is a [cheatsheet=example]cheat sheet[/cheatsheet]
explaining how to perform this activity in the Engine.
```

This creates a symbolic link between the activity description within the methodology and a cheat sheet within the Engine, which has to be defined according to Section 5.1.1. These links can be inserted anywhere within Linked Documents and their syntax is defined in the following way:

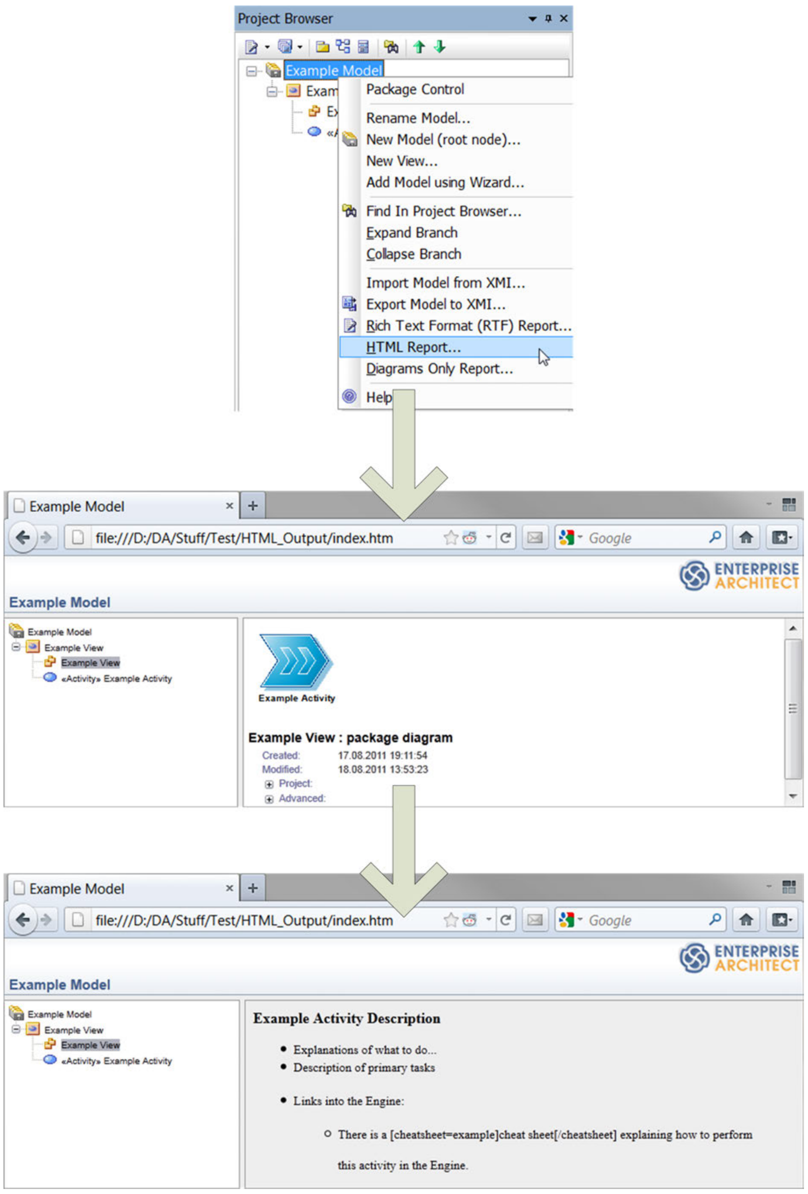
`[cheatsheet=`*`CHEATSHEET_ID`*`]`*`TEXT`*`[/cheatsheet]`

The *`CHEATSHEET_ID`* and *`TEXT`* placeholders have to be replaced by the unique cheat sheet ID of a certain cheat sheet and an arbitrary link caption, respectively. The cheat sheet identifiers are defined by the *CheatSheets plug-in*, which is discussed in Section 5.2.2.

This facility is crucial for allowing descriptions of primary tasks to be linked to and set in relation with detailed instructions on secondary tasks. While the symbolic links will not be working after being exported to a hypertext representation, as described in the next paragraphs, they will get converted to functional links later on – see Section 5.1.3.

Exporting the Methodology Model

Making the ReDSeeDS Methodology available to users solely as an Enterprise Architect model is not very convenient because this tool is meant for creating and editing this model, not for viewing and browsing it. Instead, the developers of the methodology have chosen to make use of Enterprise Architect's *HTML Report* functionality, which allows model hierarchies to be exported to a set of HTML files, which can then be viewed comfortably in any common Web browser as seen in Figure 5.11.



Example View : package diagram

Created: 17.08.2011 19:11:54

Modified: 18.08.2011 13:53:23

Example Activity Description

- Explanations of what to do...
- Description of primary tasks
- Links into the Engine:
  - There is a [cheatsheet=example]cheat sheet[/cheatsheet] explaining how to perform this activity in the Engine.

Figure 5.11: Exporting and viewing a methodology model

Not only does the generated output include an additional navigational structure for easier reading; it also entails all the Linked Documents which have automatically been converted from their original Rich Text Format to HTML pages. The figure shows how to export the model through the *Project Browser* context menu, what the start page of the report looks like and how the Linked Document is shown after clicking on the example activity.

5.1.3 Conversion of Hypertext Links

As seen in Figure 5.11, the symbolic links into the Engine are still preserved in their original (i.e. symbolic) form. They have to be converted to valid HTML links; a workflow depicted in Figure 5.12.

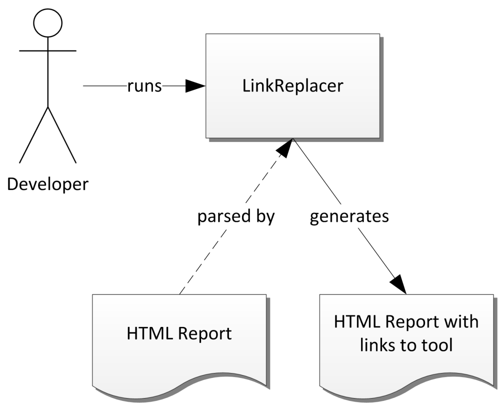


Figure 5.12: Workflow *Conversion of Hypertext Links*

Since Enterprise Architect does not offer the possibility to define post-processing rules for the HTML Report, a short program has been implemented by the author of this thesis in Java which parses the generated HTML files and replaces all symbolic links with valid HTML hyperlinks. These links direct the browser to the Web server implemented by the *WebAccess plug-in* (see Section 5.2.3): Clicking on the inserted links will trigger a reaction within the ReDSeeDS Engine – namely the display of the related cheat sheet.

There are two main reasons why these valid links have not been put into the Linked Documents in the first place: First of all, this is simply not technically possible because Enterprise Architect does not allow including arbitrary links into its model. There is only the possibility to link to other elements within the methodology, which is done in many places in order to facilitate easy navigation for the user, but is of no use for linking into the ReDSeeDS Engine. Secondly, the developers of the methodology should not be bothered with implementation details like how the Engine can be linked to, which TCP port the WebAccess plug-in is listening on, and so on. The better solution is to have an abstract mechanism, i.e. symbolic links, and to post-process the output generated by Enterprise Architect.

The source code for the `CheatsheetLinkReplacer` class is shown in a shortened version in Listing 5.2.

Listing 5.2: CheatsheetLinkReplacer Source Code

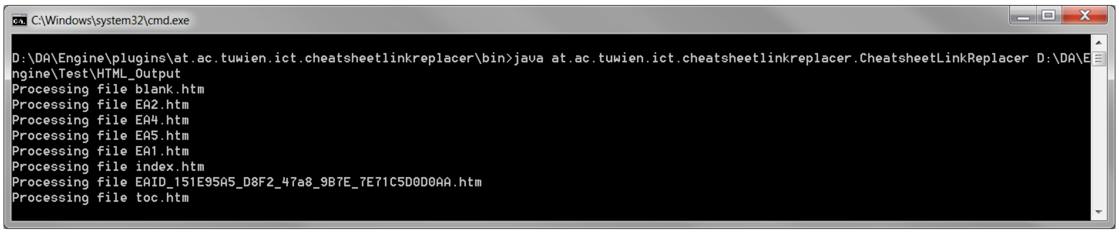
```java
package at.ac.tuwien.ict.cheatsheetlinkreplacer;
// [... import statements ...]

public class CheatsheetLinkReplacer {

  private static final String pattern_cheatsheet =
    "\\[cheatsheet=(.+)\\](.+)\\[/cheatsheet\\]";
  private static final String pattern_cheatsheeturl =
    "\\[cheatsheeturl=(.+)\\]";
  private static final String replacement_cheatsheet =
    "<a href=\"http://localhost:8081/$1\">$2</a>";
  private static final String replacement_cheatsheeturl =
    "http://localhost:8081/$1";

  private static Pattern pattern_cheatsheet_compiled;
  private static Pattern pattern_cheatsheeturl_compiled;

  /**
   * Searches for all HTML files in the folder specified as a command line
   * argument, and its subfolders.
   */
  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("Usage: java CheatsheetLinkReplacer <HTML-FOLDER>");
      System.exit(1);
    }

    File startDir = new File(args[0]);
    if (!startDir.isDirectory()) {
      System.out.println(args[0] + " is not a directory.");
      System.exit(1);
    }

    pattern_cheatsheet_compiled = Pattern.compile(pattern_cheatsheet,
      Pattern.DOTALL);
    pattern_cheatsheeturl_compiled = Pattern.compile(pattern_cheatsheeturl,
      Pattern.DOTALL);

    processRecursively(startDir);
  }

  private static void processRecursively(File dir) {
    // Search for the files and subfolders in this folder.
    File[] files = dir.listFiles();

    // Process the files and subfolders.
    for (File file : files) {
      if (file.isDirectory()) {
        processRecursively(file);
      } else {
        if (file.getName().endsWith(".html") || file.getName().endsWith(".htm")) {
          processFile(file);
        }
      }
    }
  }

  private static void processFile(File file) {
    System.out.println("Processing file " + file.getName());

    BufferedReader reader = null;
    BufferedWriter writer = null;

    try {
      reader = new BufferedReader(new FileReader(file));
      StringBuffer in_buffer = new StringBuffer();

      String line;
      while ((line = reader.readLine()) != null) {
        in_buffer.append(line + System.getProperty("line.separator"));
      }
      reader.close();

      String result = pattern_cheatsheet_compiled.matcher(in_buffer.toString())
        .replaceAll(replacement_cheatsheet);
      result = pattern_cheatsheeturl_compiled.matcher(result)
        .replaceAll(replacement_cheatsheeturl);

      writer = new BufferedWriter(new FileWriter(file));
      writer.write(result);
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (reader != null)
        try {
          reader.close();
        } catch (Exception e) {
        }
      ;
      if (writer != null)
        try {
          writer.close();
        } catch (Exception e) {
        }
      ;
    }
  }
}
```

The program expects a directory path as its command-line argument. This directory and its sub-directories are recursively scanned for HTML files which are then processed individually. The files are read and symbolic links are replaced by using regular expressions. There are two pairs of regular expressions and replacements defined, as shown in Table 5.1. The first pair has already been described in Section 5.1.2. The second pair is similar but intended to be used directly within Enterprise Architect hypertext elements.

Search pattern	Replacement pattern
[cheatsheet=*ID*]*TEXT*[/cheatsheet]	<a href="http://localhost:8081/*ID*">*TEXT*</a>
[cheatsheeturl=*ID*]	http://localhost:8081/*ID*

Table 5.1: Regular epxressions used by `CheatsheetLinkReplacer`

Running the program outputs the names of the single files it is processing, as depicted in Figure 5.13. Viewing the HTML output again after this post-processing, it can be seen that the

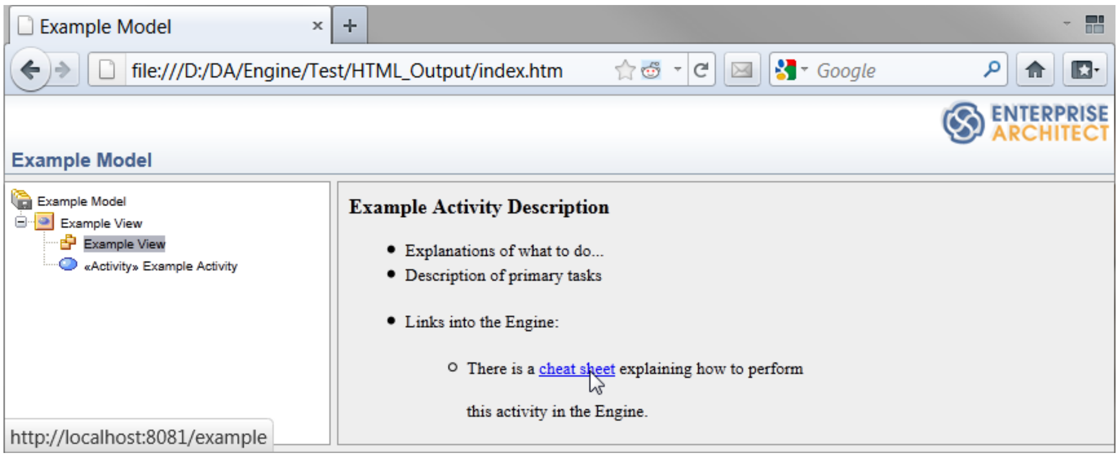


C:\Windows\system32\cmd.exe

```
D:\DA\Engine\plugins\at.ac.tuwien.ict.cheatsheetlinkreplacer\bin>java at.ac.tuwien.ict.cheatsheetlinkreplacer.CheatsheetLinkReplacer D:\DA\E
ngine\Test\HTML_Output
Processing file blank.htm
Processing file EA2.htm
Processing file EA4.htm
Processing file EA5.htm
Processing file EA1.htm
Processing file index.htm
Processing file EAID_151E95A5_D8F2_47a8_9B7E_7E71C5D0D0AA.htm
Processing file toc.htm
```

Figure 5.13: Running the `CheatsheetLinkReplacer` program

formerly symbolic hyperlink in the example activity description has been replaced by a valid hyperlink, which can be clicked by the user of the methodology (see Figure 5.14). Following the link is supposed to open the appropriate cheat sheet within the ReDSeeDS Engine. The underlying technology is discussed in Section 5.2.



Example Model

file:///D:/DA/Engine/Test/HTML_Output/index.htm

Example Model

Example Model
Example View
Example View
«Activity» Example Activity

Example Activity Description

- Explanations of what to do...
- Description of primary tasks

- Links into the Engine:
  - There is a cheat sheet explaining how to perform

    this activity in the Engine.

http://localhost:8081/example

Figure 5.14: Viewing the Methodology after running `CheatsheetLinkReplacer`

5.2 Runtime Perspective

5.2.1 Extending the Eclipse Platform

The techniques for coupling the ReDSeeDS Methodology and the ReDSeeDS Engine discussed in this chapter have not been included in the original ReDSeeDS Engine. Instead, cheat sheets and the WebAccess plug-in have to be provided in the form of *Eclipse Plug-ins*, as detailed in Sections 5.2.2 and 5.2.3. This section gives an overview over the Eclipse infrastructure which has to be used in order to write, compile and make available these plug-ins.

The Eclipse infrastructure is summarised in [4] as follows:

> Eclipse isn't a single monolithic program, but rather a small kernel containing a plug-in loader surrounded by hundreds (and potentially thousands) of plugins [...].

> This small kernel is an implementation of the OSGi R4 specification and provides the environment in which plug-ins execute. Each plug-in contributes to the whole in a structured manner, may rely on services provided by another plug-in, and each may in turn provide services on which yet other plug-ins may rely.

Starting with version 3 of the Eclipse Platform, the developers have introduced a new runtime layer based on standards by the OSGi Alliance[2], which has replaced the former custom plug-in mechanism in Eclipse. Plug-ins, or *bundles* as they are called in OSGi[3] terms, are now supported by a framework which provides a powerful dynamic component model and allows components to be remotely installed, started, stopped, updated and uninstalled during runtime. The Eclipse implementation of the framework specification is called *Equinox* and is maintained as an Eclipse project of its own[4]. For detailed information about the OSGi platform specification, see [1].

Plug-ins are usually packaged into JAR archives which mainly contain Java source code and the plug-in description files `MANIFEST.MF` and `plugin.xml`. These files describe the plug-ins' dependencies and services. As it is the case with cheat sheets, these files can be either edited manually or by using the *Manifest Editor* provided by Eclipse. Among the various settings which can be controlled through these files, like giving plug-ins unique identifiers, the concept of extensions and extension points is particularly important. Plug-ins may declare extension points "so that other plug-ins can extend the functionality of the original plug-in in a controlled manner" [4]. Regarding the coupling of methodology and tool, this is necessary in order to extend the cheat sheet facility and provide additional cheat sheets, as described in Section 5.2.2, and in order to extend the *Startup* functionality to make sure that the WebAccess plug-in is started when Eclipse launches, as described in Section 5.2.3.

Plug-in behaviour is specified in the source code files. Additionally to custom classes, each plug-in may provide an *Activator Class*, which controls its lifecycle during runtime, for example by saving and restoring state information when its `start()` and `stop()` methods are called by the Eclipse Platform.

Eclipse provides a "New Plug-in Project" wizard which helps developers to create a new plug-in along with its basic directory and file structure from scratch. For detailed instructions on how to do this, see the Eclipse Help or the tutorial provided by IBM [38]. In order to install a plug-in, the source code classes have to be compiled and then packaged into a JAR archive together with the two description XML documents and any other resource files. The JAR archive then has to be put into the `plugins` directory within the Eclipse installation directory and will be loaded after (re)starting the application. If the plug-in version number has not changed, restarting Eclipse with the `-clean` parameter will ensure that no older, cached version of the plug-in is loaded.

5.2.2 Providing Cheat Sheets

As shown in Figure 5.15, cheat sheets designed according to Section 5.1.1 have to be made available to the Eclipse Platform via a plug-in called the *CheatSheets plug-in*.

[2] http://www.osgi.org/

[3] Open Services Gateway initiative

[4] http://www.eclipse.org/equinox/

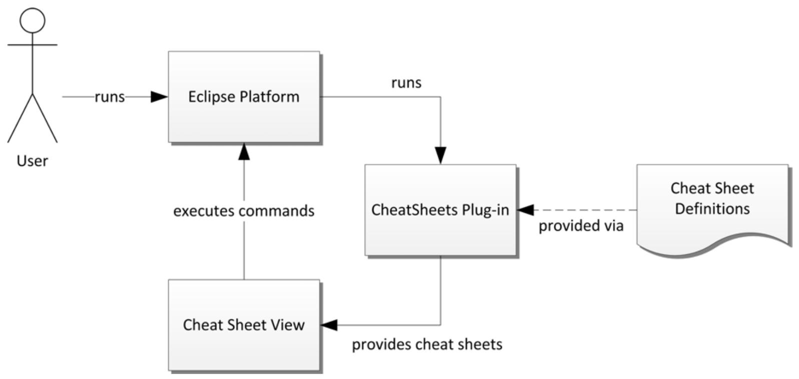
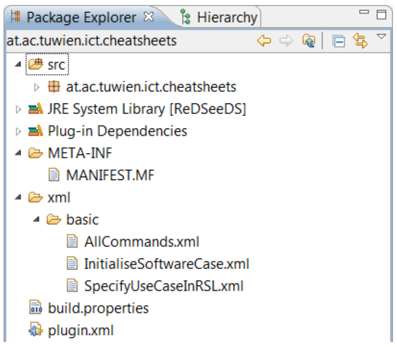


Figure 5.15: Workflow *Providing Cheat Sheets*

This plug-in does not provide any special behaviour information, as can be seen from its source code in Listing 5.3. The Activator Class has been implemented, however, to show where future behaviour can be integrated. The main point of the CheatSheets plug-in is to provide cheat sheets for the Engine. This is done by including a number of cheat sheet definitions (i.e. XML files) within the JAR package, as can be seen in the Package Explorer's hierarchical view of the `at.ac.tuwien.ict.cheatsheets` package (see Figure 5.16).

Listing 5.3: Cheat Sheets Plug-in: `CheatsheetsPlugin.java`

```java
package at.ac.tuwien.ict.cheatsheets;

import org.eclipse.ui.plugin.AbstractUIPlugin;
import org.osgi.framework.BundleContext;

public class CheatsheetsPlugin extends AbstractUIPlugin {

  public static final String PLUGIN_ID = "at.ac.tuwien.ict.cheatsheets";
  private static CheatsheetsPlugin plugin;

  public CheatsheetsPlugin() {
  }

  public void start(BundleContext context) throws Exception {
    super.start(context);
    plugin = this;
  }

  public void stop(BundleContext context) throws Exception {
    plugin = null;
    super.stop(context);
  }

  public static CheatsheetsPlugin getDefault() {
    return plugin;
  }

}
```

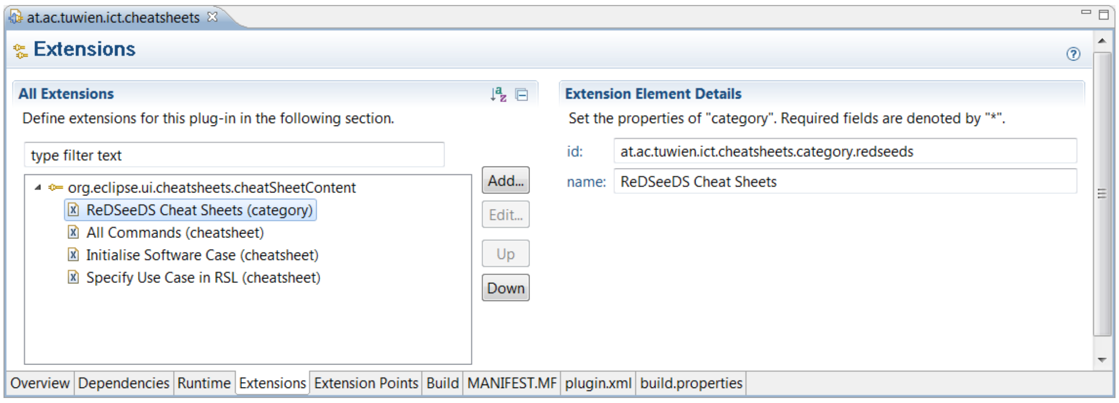

Package Explorer | Hierarchy

at.ac.tuwien.ict.cheatsheets

```
src
  at.ac.tuwien.ict.cheatsheets
JRE System Library [ReDSeeDS]
Plug-in Dependencies
META-INF
  MANIFEST.MF
xml
  basic
    AllCommands.xml
    InitialiseSoftwareCase.xml
    SpecifyUseCaseInRSL.xml
build.properties
plugin.xml
```

Figure 5.16: The CheatSheets plug-in's package hierarchy

The cheat sheet definitions are then referenced by the plug-in configuration, which defines extensions to the `org.eclipse.ui.cheatsheets.cheatSheetContent` extension point already present within the Eclipse Platform. This is conveniently done via Eclipse's *Manifest Editor*'s *Extensions* tab. After creating the extension, sub-elements may be added for cheat sheet categories and actual cheat sheets. In the example shown in Figure 5.17, one category named "ReDSeeDS Cheat Sheets" has been added. This category will be shown in the cheat sheet overview dialogue, as explained at the end of this section.


at.ac.tuwien.ict.cheatsheets

Extensions

All Extensions

Define extensions for this plug-in in the following section.

type filter text

```
org.eclipse.ui.cheatsheets.cheatSheetContent
  ReDSeeDS Cheat Sheets (category)
  All Commands (cheatsheet)
  Initialise Software Case (cheatsheet)
  Specify Use Case in RSL (cheatsheet)
```

Extension Element Details

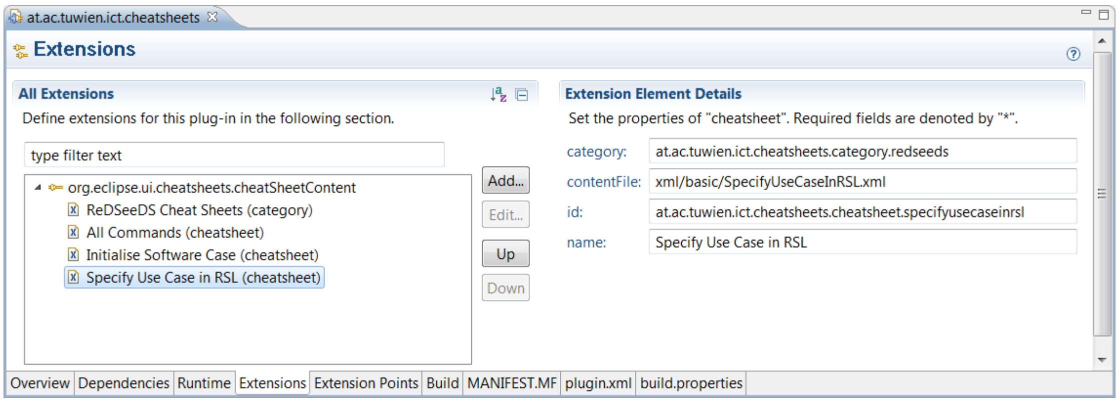
Set the properties of "category". Required fields are denoted by "*".

id: at.ac.tuwien.ict.cheatsheets.category.redseeds

name: ReDSeeDS Cheat Sheets

Figure 5.17: Extension details for a cheat sheet category shown in the Manifest Editor

As mentioned, the Manifest Editor may also be used to reference the actual cheat sheet definitions. An example is depicted in Figure 5.18, which shows details for a cheat sheet called "Specify Use Case in RSL". It belongs to the category which has been defined before and gets its content from the XML document with the relative path `xml/basic/SpecifyUseCaseInRSL.xml`. This path reflects the package hierarchy shown above in Figure 5.16. It is particularly important to set the cheat sheet's unique identifier here, which is the identifier referenced by the symbolic cheat sheet links in Linked Documents (as described in Section 5.1.2).



Extensions

All Extensions

Define extensions for this plug-in in the following section.

type filter text

- org.eclipse.ui.cheatsheets.cheatSheetContent
  - ReDSeeDS Cheat Sheets (category)
  - All Commands (cheatsheet)
  - Initialise Software Case (cheatsheet)
  - Specify Use Case in RSL (cheatsheet)

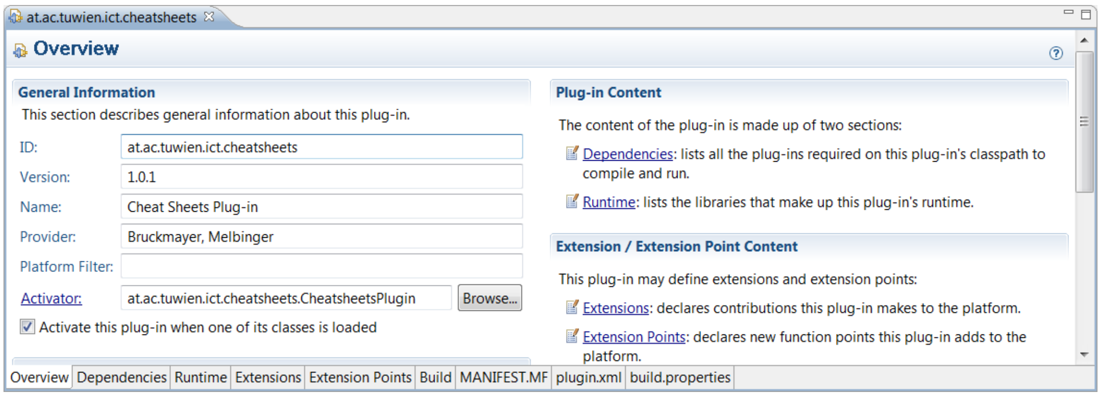
Extension Element Details

Set the properties of "cheatsheet". Required fields are denoted by "*".

category:	at.ac.tuwien.ict.cheatsheets.category.redseeds
contentFile:	xml/basic/SpecifyUseCaseInRSL.xml
id:	at.ac.tuwien.ict.cheatsheets.cheatsheet.specifyusecaseinrsl
name:	Specify Use Case in RSL

Figure 5.18: Extension details for a cheat sheet shown in the Manifest Editor

The plug-ins's main settings can easily be edited on the Manifest Editor's *Overview* tab, as shown in Figure 5.19. Apart from providing the plug-in ID and version information, it is important to specify the Activator class for this plug-in.



Overview

General Information

This section describes general information about this plug-in.

ID:	at.ac.tuwien.ict.cheatsheets
Version:	1.0.1
Name:	Cheat Sheets Plug-in
Provider:	Bruckmayer, Melbinger
Platform Filter:	
Activator:	at.ac.tuwien.ict.cheatsheets.CheatsheetsPlugin

[✓] Activate this plug-in when one of its classes is loaded

Plug-in Content

The content of the plug-in is made up of two sections:

- Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.
- Runtime: lists the libraries that make up this plug-in's runtime.

Extension / Extension Point Content

This plug-in may define extensions and extension points:

- Extensions: declares contributions this plug-in makes to the platform.
- Extension Points: declares new function points this plug-in adds to the platform.

Figure 5.19: The Manifest Editor showing the CheatSheets plug-in's general configuration

As pointed out in Section 5.2.1, Eclipse plug-ins define their runtime services and extensions through the `MANIFEST.MF` and the `plugin.xml` files. The Manifest Editor provides a convenient way to edit them, they could however be created manually as well. Listings 5.4 and 5.5 show the source code for these files which are, together, equivalent to the settings made in the Manifest Editor's tabs.

Listing 5.4: CheatSheets Plug-in: `MANIFEST.MF`

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 10.0-b23 (Sun Microsystems Inc.)
Bundle-ManifestVersion: 2
Bundle-Name: Cheat Sheets Plug-in
Bundle-SymbolicName: at.ac.tuwien.ict.cheatsheets;singleton:=true
Bundle-Version: 1.0.1
```

```
Bundle-Activator: at.ac.tuwien.ict.cheatsheets.CheatsheetsPlugin
Bundle-Vendor: Bruckmayer, Melbinger
Require-Bundle: org.eclipse.ui,org.eclipse.core.runtime,org.eclipse.ui
 .cheatsheets
Eclipse-LazyStart: true
```

Listing 5.5: CheatSheets Plug-in: `plugin.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
         point="org.eclipse.ui.cheatsheets.cheatSheetContent">
      <category
            id="at.ac.tuwien.ict.cheatsheets.category.redseeds"
            name="ReDSeeDS Cheatsheets">
      </category>
      <cheatsheet
            category="at.ac.tuwien.ict.cheatsheets.category.redseeds"
            contentFile="xml/basic/AllCommands.xml"
            id="at.ac.tuwien.ict.cheatsheets.cheatsheet.allcommands"
            name="All Commands">
      </cheatsheet>
      <cheatsheet
            category="at.ac.tuwien.ict.cheatsheets.category.redseeds"
            contentFile="xml/basic/InitialiseSoftwareCase.xml"
            id="at.ac.tuwien.ict.cheatsheets.cheatsheet.initialisesoftwarecase"
            name="Initialise Software Case">
      </cheatsheet>
      <cheatsheet
            category="at.ac.tuwien.ict.cheatsheets.category.redseeds"
            contentFile="xml/basic/SpecifyUseCaseInRSL.xml"
            id="at.ac.tuwien.ict.cheatsheets.cheatsheet.specifyusecaseinrsl"
            name="Specify Use Case in RSL">
      </cheatsheet>
   </extension>
</plugin>
```

After compiling and installing the plug-in (i.e. after putting the JAR archive into the `plugins` folder and restarting the Engine), the cheat sheets can be viewed in the *Cheat Sheet View* provided by the Eclipse Platform. Additionally to being opened by the WebAccess plug-in, they can also be viewed by selecting them in the *Help* > *Cheat Sheets...* dialogue as shown in Figure 5.20. Note that by running the plug-in, the cheat sheets have been registered to extend the cheat sheets under a specific category and are, therefore, visible in this dialogue now.

5.2.3 Enabling Linking into the ReDSeeDS Engine

The previous sections describe how to prepare the methodology for the user to read and how to prepare the tool to provide cheat sheets which will guide the user through secondary tasks. The methodology even provides hyperlinks at this point, but they are still not working because there is no server listening for incoming requests from these hyperlinks. Is it the *WebAccess plug-in*'s job to do this and to ensure that appropriate actions are performed within the Engine.

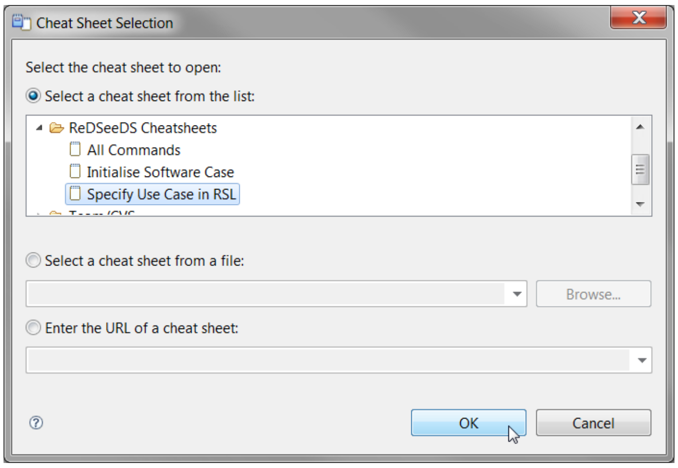

Cheat Sheet Selection
Select the cheat sheet to open:
Select a cheat sheet from the list:
ReDSeeDS Cheatsheets
All Commands
Initialise Software Case
Specify Use Case in RSL
Select a cheat sheet from a file:
Browse...
Enter the URL of a cheat sheet:
OK
Cancel

Figure 5.20: Opening registered cheat sheets directly

The basic idea is to have the WebAccess plug-in emulate a simple Web server. The user is reading the methodology in a Web browser which, upon activation of a link, sends an HTTP[5] request to the address specified by the hyperlink. The originally symbolic links have been replaced by links to the local machine as described in Section 5.1.3 already, so that the only component missing is a Web server listening for incoming requests as long as the ReDSeeDS Engine is running. Figure 5.21 shows this part of the Coupling Workflow.

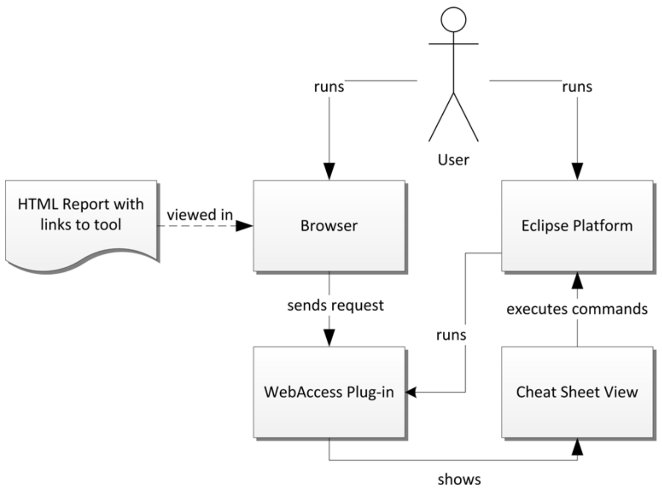


Figure 5.21: Workflow *Enabling Linking into the ReDSeeDS Engine*

The user is running the browser in order to study the methodology and is also running the Engine in order to work according to the methodology. When a link into the Engine is activated within the browser, it sends an HTTP request to the WebAccess plug-in within the Engine,

[5]Hypertext Transfer Protocol

which in turn displays the cheat sheet defined for the specific activity the user is reading about in Eclipse's built-in Cheat Sheet View. This cheat sheet may then even execute commands within the Engine, as described in Section 5.1.1. The plug-in knows which cheat sheet to show because its unique identifier is sent along with the request by the browser.

The plug-in's `MANIFEST.MF` file, which defines `at.ac.tuwien.ict.webaccess.WebAccess` as its Activator class, is shown in Listing 5.6.

Listing 5.6: WebAccess Plug-in: `MANIFEST.MF`

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 10.0-b23 (Sun Microsystems Inc.)
Bundle-ManifestVersion: 2
Bundle-Name: WebAccess Plug-in
Bundle-SymbolicName: at.ac.tuwien.ict.webaccess;singleton:=true
Bundle-Version: 1.0.1
Bundle-Activator: at.ac.tuwien.ict.webaccess.WebAccess
Require-Bundle: org.eclipse.ui,org.eclipse.core.runtime,org.eclipse.jd
 t.ui,org.eclipse.ui.cheatsheets,org.eclipse.help.ui
Eclipse-LazyStart: false
Bundle-Vendor: Bruckmayer, Melbinger
```

It should be noted that the `Eclipse-LazyStart` attribute is set to `false`, which means that there will be no automatic start-up but that the developer is responsible for starting the plug-in himself. The background is explained in [37]:

> Plug-ins are normally set to load lazily, so that the code isn't loaded into memory until it is required. This is normally a good thing as you don't want to affect the startup time of Eclipse. If you do require your plug-in to start up and load when Eclipse launches, you can use the `org.eclipse.ui.startup` extension point.

In the case of a Web server which needs to be running during all the time that the Engine is running, non-lazy startup has to be ensured and this is reflected in the `plugin.xml` file shown in Listing 5.7.

Listing 5.7: WebAccess Plug-in: `plugin.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
         point="org.eclipse.ui.startup">
      <startup></startup>
   </extension>
</plugin>
```

The WebAccess plug-in extends the `org.eclipse.ui.startup` extension point where plug-ins that are to be activated on startup can register. All extensions to this point are guaranteed to be loaded as soon as the Eclipse Platform is starting. WebAccess only has to implement the `org.eclipse.ui.IStartup` interface, which introduces the `earlyStartup()` method. This can be seen in the Activator class' source code in Listing 5.8.

Listing 5.8: WebAccess Plug-in: `WebAccess.java`

```
package at.ac.tuwien.ict.webaccess;
// [... import statements ...]

public class WebAccess extends AbstractUIPlugin implements IStartup {

  public static final String PLUGIN_ID = "WebAccess";
  private static WebAccess plugin;
  private ServerSocket server_socket;

  class CheatSheetShow implements Runnable {

    private String id;

    public CheatSheetShow(String id) {
      this.id = id;
    }

    public void run() {
      CheatSheetView view = ViewUtilities.showCheatSheetView();
      view.setInput(id);
      IWorkbenchPage page = view.getSite().getWorkbenchWindow().getActivePage();
      page.bringToTop(view);
    }

  }

  public static void showCheatSheet(String id) {
    Display.getDefault().asyncExec(getDefault().new CheatSheetShow(id));
  }

  public void start(BundleContext context) throws Exception {
    super.start(context);
    plugin = this;
    System.out.println("WebAccess started. Creating server.");

    try {
      server_socket = new ServerSocket(8081);
      new ServerThread(server_socket).start();
    } catch (IOException e) {
      System.out.println("Could not create server (" + e.getMessage() + ")");
      return;
    }
  }

  public void stop(BundleContext context) throws Exception {
    plugin = null;
    super.stop(context);

    server_socket.close();
    System.out.println("WebAccess stopped.");
  }

  public static WebAccess getDefault() {
    return plugin;
  }

  public void earlyStartup() { }
}
```

The `WebAccess` class starts its operations when the `start()` method is called by the Eclipse Platform (line 34). This method opens a TCP[6] server socket on port 8081, creates a new instance of `ServerThread` and starts its execution as a Java thread. This thread is listening for incoming requests from this moment on. When the Eclipse Platform shuts down, the `stop()` method is called which then closes the server socket (line 48).

`WebAccess` also implements a facility to conveniently display cheat sheets to the user. The inner class `CheatSheetShow`, which implements the `Runnable` interface and can, therefore, be started as a thread, expects a cheat sheet identifier in its constructor. When its `run()` method is called (line 21), it creates a cheat sheet view into which it loads the requested cheat sheet and then shows this view within the current workbench window. In order to access this functionality, `WebAccess`' static method `showCheatSheet()` may be called, which takes a cheat sheet identifier, creates a new `CheatSheetShow` thread with it and then tells the Eclipse Platform's default display to asynchronously execute this thread (line 30). Through this rather complicated set-up, incoming requests and the display of cheat sheets do not block each other.

Listing 5.9 shows the source code for the `ServerThread` class.

Listing 5.9: WebAccess Plug-in: `ServerThread.java`

```java
package at.ac.tuwien.ict.webaccess;
// [... import statements ...]

public class ServerThread extends Thread {

  private ServerSocket server;

  public ServerThread(ServerSocket server) {
    this.server = server;
  }

  public void run() {
    while (true) {
      Socket client_socket = null;
      try {
        client_socket = server.accept();
        System.out.println("Got a connection (" + client_socket.toString() + ")");

        BufferedReader in = new BufferedReader(new InputStreamReader(client_socket.
            getInputStream()));
        Pattern p = Pattern.compile("GET (.*) HTTP/.\\..");

        String line;
        while ((line = in.readLine()) != null) {
          Matcher m = p.matcher(line);
          if (m.find()) {
            System.out.println("Found " + m.group(1) + " in the GET request.");
            WebAccess.showCheatSheet("at.ac.tuwien.ict.cheatsheets.cheatsheet." + m
                .group(1).substring(1));
            break;
          }
        }

        // Send an answer.
        String reply = "<html><head><title>WebAccess</title><script type=\"text/
            javascript\">history.back();</script></head><body><font face=\"Verdana
```

[6]Transmission Control Protocol

```
              \" size=\"2\"><span style=\"font-size:10pt\">The cheatsheet was opened
              .<br /><br /><a href=\"#\" onclick=\"history.back();\">Back</a></span
              ></font></body></html>";
34
35          BufferedWriter out = new BufferedWriter(new OutputStreamWriter(
              client_socket.getOutputStream()));
36          out.write("HTTP/1.1 200 OK\n");
37          out.write("Content-Length: " + reply.length() + "\n");
38          out.write("Content-Type: text/html\n");
39          out.write("Connection: close\n");
40          out.write("\n");
41          out.write(reply);
42          out.close();
43        } catch (SocketException e) {
44          // The ServerSocket was closed when stop() was called.
45          break;
46        } catch (IOException e) {
47          e.printStackTrace();
48        } finally {
49          if (client_socket != null) try { client_socket.close(); } catch (
              IOException e) { }
50        }
51      }
52    }
53
54  }
```

As explained above, `ServerThread` is started as a thread as soon as the WebAccess plug-in is started. It is listening on the server socket created by `WebAccess` until there is an exception – which usually means that the socket was closed by `WebAccess` because the Engine is shutting down. Incoming requests are accepted and its payload is read into a `BufferedReader` (line 19). This input is then scanned line by line until a regular expression resembling an HTTP request is matched (line 25). By scanning for the following regular expression, HTTP/1.0 requests are accepted as well as HTTP/1.1 requests, which are usually sent by modern browsers:

```
GET (.*) HTTP/.\..
```

As soon as a request has been identified, the request path (i.e. the requested URL) is treated as a cheat sheet identifier and forwarded to the `WebAccess.showCheatSheet()` method, which in turn shows the specified cheat sheet (line 27). All that is left for `ServerThread` to do at this point is to send an HTTP response to the client (line 35). This message will then be displayed by the end-user's browser and, therefore, provides a short explanation that the cheat sheet was opened and a "Back" link, which takes the user back to the page he was at before, which is typically an activity description. This is done with a JavaScript `history.back()` call. Finally, the TCP connection between client and server is closed and `ServerThread` stops.

The debug messages output by `WebAccess` and `ServerThread` can be read if the ReDSeeDS Engine is started with the `-consoleLog` parameter.

5.3 Practical Usage

This section gives an example of the usage of method-tool coupling which shows how this technology can be used in "real life" while working with the ReDSeeDS Methodology and the ReDSeeDS Tool. The methodology could be exchanged for any other methodology that has been defined according to the principles discussed above. Also, another tool built upon the Eclipse Platform might make use of the plug-ins and provide coupling for the end-user. In order to demonstrate the process, this section shows how a Requirements Engineer specifies requirements with the ReDSeeDS framework. Figure 5.22 shows an overview of the work definition *Specify Requirements in RSL*.

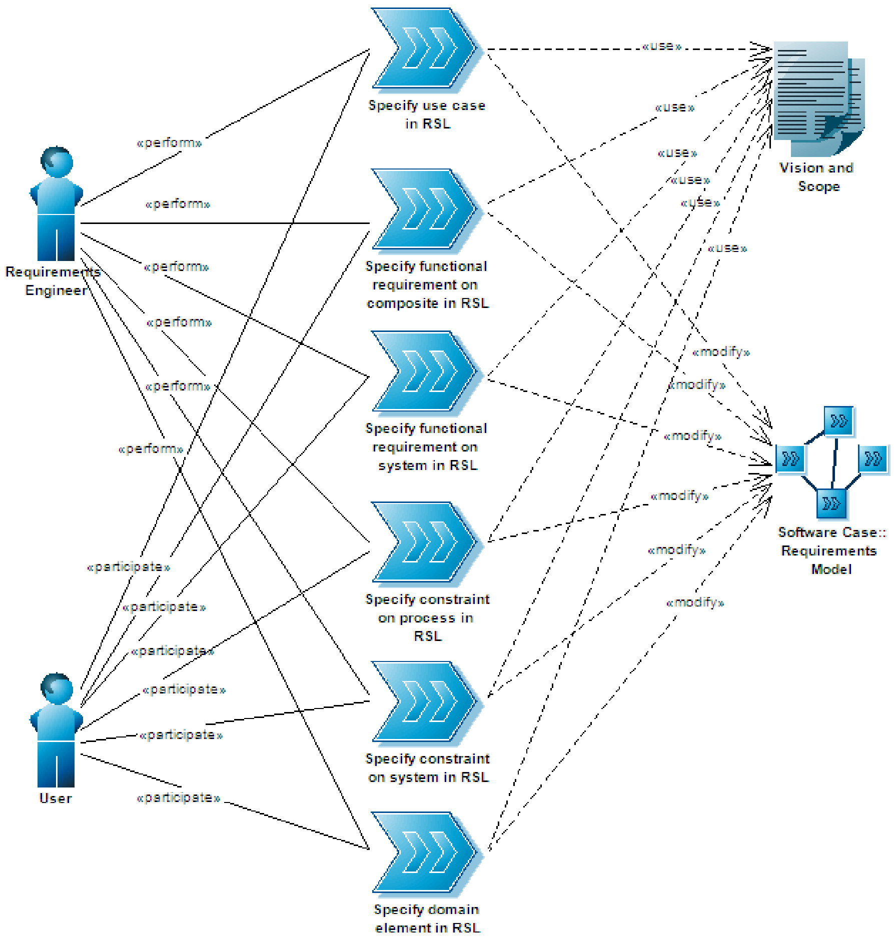
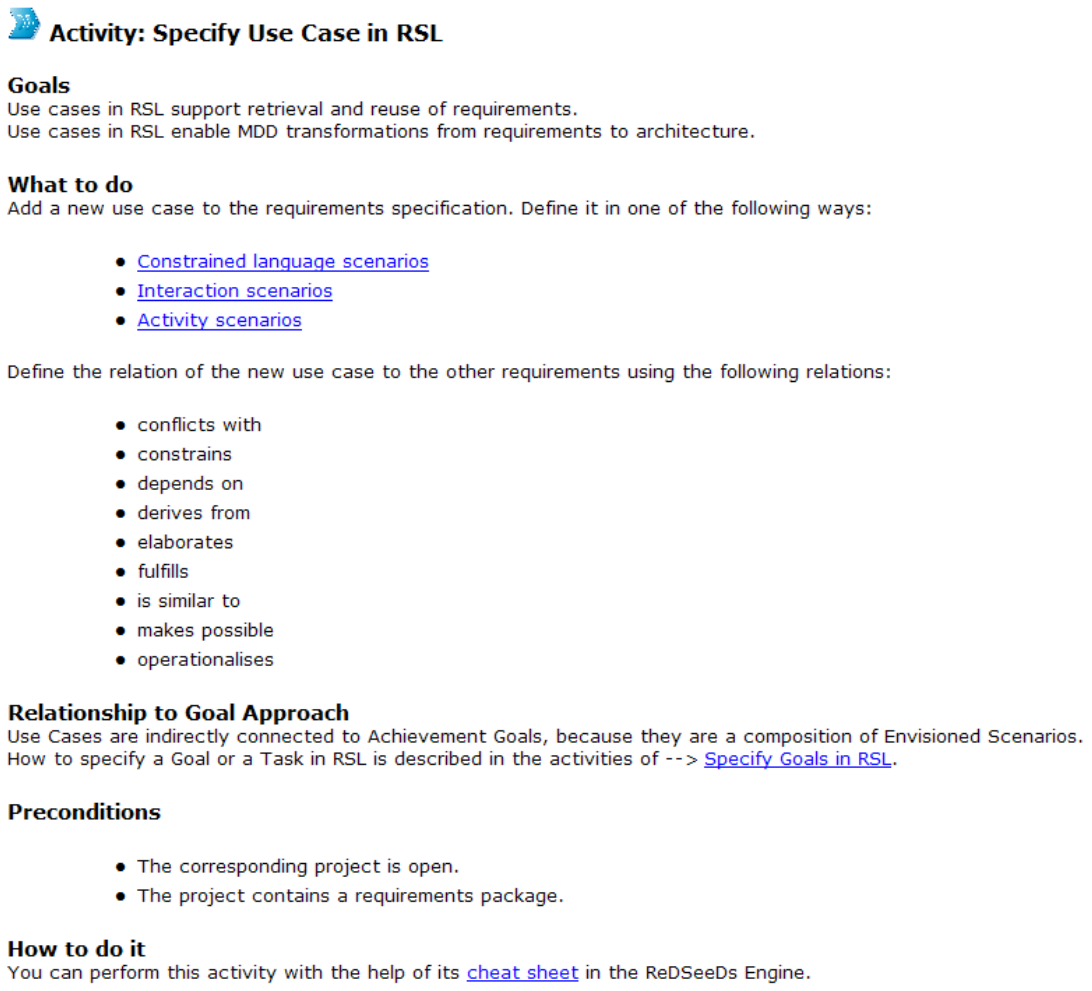

Requirements Engineer
User
«perform»
«participate»
Specify use case in RSL
Specify functional requirement on composite in RSL
Specify functional requirement on system in RSL
Specify constraint on process in RSL
Specify constraint on system in RSL
Specify domain element in RSL
«use»
«modify»
Vision and Scope
Software Case:: Requirements Model

Figure 5.22: Work definition *Specify Requirements in RSL*, from [20]

We assume that the user is viewing it in a Web browser. This means that the methodology has already been defined in SPEM using Enterprise Architect and equipped with Linked Documents. Into these documents, symbolic links have been inserted which have then been replaced by actual links using the `CheatsheetLinkReplacer` program.

The Requirements Engineer is now interested in the activity named *Specify use case in RSL*. By clicking on it, he is taken to a Linked Document which details the primary tasks of this activity in a textual description and also provides a link into the ReDSeeDS Engine, as shown in Figure 5.23. The primary tasks explain why this activity exists, what its meaning is within the particular methodology the user is following and *what* to do. The link into the Engine, on the other hand, is supposed to lead the user to a cheat sheet explaining step-by-step *how* to perform all that by showing the secondary tasks and how exactly this can be done with the Engine. This way, the high-level goals presented above can be achieved.



Activity: Specify Use Case in RSL

Goals
Use cases in RSL support retrieval and reuse of requirements.
Use cases in RSL enable MDD transformations from requirements to architecture.

What to do
Add a new use case to the requirements specification. Define it in one of the following ways:

- Constrained language scenarios
- Interaction scenarios
- Activity scenarios

Define the relation of the new use case to the other requirements using the following relations:

- conflicts with
- constrains
- depends on
- derives from
- elaborates
- fulfills
- is similar to
- makes possible
- operationalises

Relationship to Goal Approach
Use Cases are indirectly connected to Achievement Goals, because they are a composition of Envisioned Scenarios.
How to specify a Goal or a Task in RSL is described in the activities of --> Specify Goals in RSL.

Preconditions

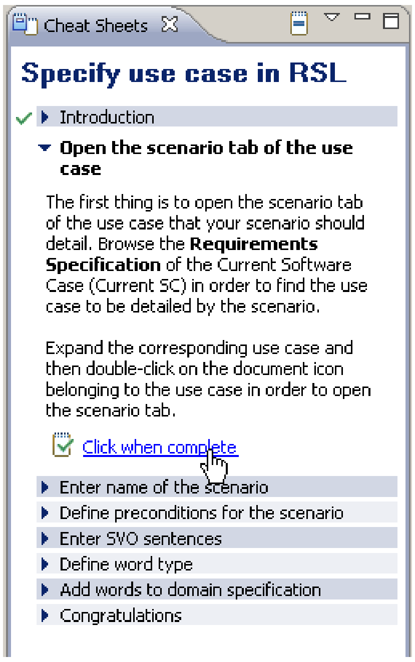
- The corresponding project is open.
- The project contains a requirements package.

How to do it
You can perform this activity with the help of its cheat sheet in the ReDSeeDs Engine.

Figure 5.23: Activity description *Specify use case in RSL*, from [20]

Clicking onto the link into the Engine shows the explicitly linked cheat sheet and the user can continue his work within the tool, as seen in Figure 5.24.

As shown in Section 5.1.1, the cheat sheet may now even perform certain actions automatically on behalf of the user. These "click-to-perform" links improve the user-experience even


Cheat Sheets

Specify use case in RSL

✓ Introduction

Open the scenario tab of the use case

The first thing is to open the scenario tab of the use case that your scenario should detail. Browse the **Requirements Specification** of the Current Software Case (Current SC) in order to find the use case to be detailed by the scenario.

Expand the corresponding use case and then double-click on the document icon belonging to the use case in order to open the scenario tab.

Click when complete

Enter name of the scenario

Define preconditions for the scenario

Enter SVO sentences

Define word type

Add words to domain specification

Congratulations

Figure 5.24: Cheat sheet *Specify use case in RSL*, from [20]

more and make learning how to apply a certain methodology within a specific tool a more pleasant undertaking.

As soon as he is finished with one specific cheat sheet, the Requirements Engineer will return to the methodology and perform the next activities building on the ones he has already executed. If appropriate, these other activities will again include links to their own, specific cheat sheets which will accompany the user along his journey.

CHAPTER 6

Conclusion

While there are many tools supporting developers and many methodologies explaining what to do in which order when building software, there usually is a remarkable gap between the tools and the methodologies. Work on the EU-funded ReDSeeDS Project has shown that the ReDSeeDS Engine and the ReDSeeDS Software Development Methodology deliver much higher value to the user when they are coupled tightly. In other words, primary tasks which someone wants to perform have to be set in close relation to secondary tasks which prescribe how to use a tool in order to accomplish the primary tasks.

This thesis describes the context of the actual study domain in order to give a broad overview of where the problem statement comes from and which context has to be regarded in order to fully understand it. The ReDSeeDS technologies provide powerful capabilities to their users but they only unleash their full potential when used in accordance with a strong methodology, when a tool supports them wherever possible and, especially, when there is tight and direct coupling between the two of them.

The methodology is presented to the user in the form of Web pages and the tool is primarily built on the Eclipse Platform. Therefore, an approach involving HTML and Eclipse plug-ins was chosen. The prototypical implementation presented in this thesis gives a proof-of-concept solution to the question of how to couple a tool and a methodology. However, the ideas presented here can easily be transposed to other methodologies and other tools which do not necessarily have to be related to software development.

Bibliography

[1] OSGi Alliance. *OSGi Service Platform Core Specification*. OSGi Alliance, 2011.

[2] Daniel Bildhauer, Jürgen Ebert, Tassilo Horn, Lothar Hotz, Stephanie Knab, Volker Riediger, Sören Schneickert, and Katharina Wolter. Definition of a software case query language - second iteration. Technical Report D4.1.2, ReDSeeDS Project, September 2009.

[3] Ming-Feng Chen, Bin-Shiang Liang, and Jian W. Wang. A process-centered software engineering environment with network centric computing. In *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 234–239, October 1997.

[4] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. Addison-Wesley Professional, third edition, December 2008.

[5] Alistair Cockburn. Structuring Use Cases with Goals. *Journal of Object-Oriented Programming*, 5(10):56–62, 1997.

[6] Benoit Combemale, Xavier Crégut, Alain Caplain, and Bernard Coulette. Towards a rigorous process modeling with SPEM. In Y. Manolopoulos, J. Filipe, P. Constantopoulos, and J. Cordeiro, editors, *8th International Conference on Enterprise Information Systems: Databases and Information Systems Integration*. ICEIS, 2006.

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

[8] Berthold Daum. *Java-Entwicklung mit Eclipse 3.2*. dpunkt.verlag GmbH, Heidelberg, 2006.

[9] Szymon Drejewicz. Final Activity Report. Technical Report D1.2, ReDSeeDS Project, December 2009.

[10] Jürgen Ebert and Angelika Franzke. A declarative approach to graph based modeling. In Ernst Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 38–50. Springer Berlin / Heidelberg, 1995.

[11] Object Management Group. *Meta Object Facility Core Specification, version 2.0, formal/2006-01-01*. Object Management Group, January 2006.

[12] Object Management Group. *Unified Modeling Language: Superstructure, version 2.1.1, formal/2007-02-03*. Object Management Group, February 2007.

[13] Jon Holt. *UML for Systems Engineering*. Institution of Electrical Engineers, London, 2004.

[14] 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.

[15] Steffen Kahle. *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*. Institute for Software Technology, University of Koblenz-Landau, June 2006.

[16] Hermann Kaindl. Using Hypertext for Semiformal Representation in Requirements Engineering Practice. *The New Review of Hypermedia and Multimedia*, 2:149–173, 1996.

[17] Hermann Kaindl. A practical approach to combining requirements definition and object-oriented analysis. *Annals of Software Engineering*, 3:319–343, 1997.

[18] Hermann Kaindl. A Design Process Based on a Model Combining Scenarios with Goals and Functions. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 30(5):537–551, September 2000.

[19] Hermann Kaindl and Jürgen Falb. Can we transform requirements into architecture? In *The Third International Conference on Software Engineering Advances*, pages 91–96, Sliema, October 2008.

[20] Hermann Kaindl, Jürgen Falb, Stefan Melbinger, and Thomas Bruckmayer. An Approach to Method-Tool Coupling for Software Development. In *Proceedings of the Fifth International Conference on Software Engineering Advances (ICSEA 2010)*, August 2010.

[21] Hermann Kaindl, Stefan Kramer, and Mario Hailing. An Interactive Guide Through a Defined Modelling Process. In *People and Computers XV, Joint Proceedings of HCI 2001 and IHM 2001*, pages 107–124, Lille, France, September 2001. Springer.

[22] Hermann Kaindl, Benedikt Lutz, and Peter Tippold. *Methodik der Softwareentwicklung: Vorgehensmodell und State-of-the-Art der professionellen Praxis*. Vieweg, Braunschweig / Wiesbaden, Germany, 1998.

[23] Hermann Kaindl, Michal Smialek, and Wiktor Nowakowski. Case-based Reuse with Partial Requirements Specifications. In *Proceedings of the 18th IEEE International Requirements Engineering Conference (RE 2010)*, pages 399–400, 2010.

[24] Hermann Kaindl, Michal Smialek, Patrick Wagner, Davor Svetinovic, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Hannes Schwarz, Daniel Bildhauer, Jürgen Falb, John P. Brogan, Kizito S. Mukasa, Katharina Wolter, Sevan Kavaldjian, Alexander Szép, Elina Kalnina, and Audris Kalnins. Requirements Specification Language Definition. Technical Report D2.4.2, ReDSeeDS Project, October 2009.

[25] Hermann Kaindl and Davor Svetinovic. On confusion between requirements and their representations. *Requirements Engineering*, 15:307–311, 2010.

[26] Hermann Kaindl and Patrick Wagner. A Unification of the Essence of Goal-Oriented Requirements Engineering. In *The Fourth International Conference on Software Engineering Advances*, pages 45–50, Porto, September 2009.

[27] 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 Jürgen Falb. Final Reuse-Oriented Modelling and Transformation Language Definition. Technical Report D3.2.2, ReDSeeDS Project, September 2009.

[28] Thorsten Krebs, Wiktor Nowakowski, Audris Kalnins, and Elina Kalnina. Modelling and Transformation Language Validation Report. Technical Report D3.4, ReDSeeDS Project, December 2007.

[29] Philippe Kruchten. *The Rational Unified Process: An Introduction, 3rd ed.* Addison Wesley, 2003.

[30] John Mylopoulos, Lawrence Chung, and Eric Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, 1999.

[31] Object Management Group. *Software Process Engineering Metamodel Specification, version 1.1, formal/05-01-06*, January 2005.

[32] Jenny Preece, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland, and Tom Carey. *Human Computer Interaction*. Addison-Wesley, 1994.

[33] Michal Rein, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Audris Kalnins, Edgars Celms, Elina Kalnina, Daniel Bildhauer, Tomasz Szymanski, Kizito S. Mukasa, and Özgür Ünalan. Final ReDSeeDS Prototype. Technical Report D5.4.3, ReDSeeDS Project, November 2009.

[34] Volker Riediger, Daniel Bildhauer, Hannes Schwarz, Audris Kalnins, Agris Sostaks, Edgars Celms, Michal Smialek, and Albert Ambroziewicz. Software Case Meta-Model Definition. Technical Report D3.1, ReDSeeDS Project, March 2007.

[35] Izzet Safer, Gail C. Murphy, Julie Waterhouse, and Jin Li. A focused learning environment for Eclipse. In *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 75–79, 2006.

[36] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (Second Edition)*. Addison-Wesley Professional, 2 edition, January 2008.

[37] James Sugrue. *Eclipse Plug-in Development*. DZone, 2009.

[38] Philipp Tiedt. Building cheat sheets in Eclipse. `http://www.ibm.com/developerworks/opensource/tutorials/os-cheatsheets/`, December 2005.

[39] Katharina Wolter, Lothar Hotz, Michal Smialek, Hermann Kaindl, Patrick Wagner, Sebastian Weber, Kizito S. Mukasa, and Andreas Jedlitschka. ReDSeeDS Software Development Methodology - 2nd Release. Technical Report D7.1.2, ReDSeeDS Project, November 2009.