Departamento
de Engenharia
Informática

Relatório

TRABALHO FINAL DE CURSO

LICENCIATURA EM ENGENHARIA
INFORMÁTICA E DE COMPUTADORES (LEIC)

Ano Lectivo 2005/2006

N.º da Proposta: 11

Título: ProjectIT-RSL

Professor Orientador:

Alberto Manuel Rodrigues da Silva _____

Professor Co-Orientador:

Carlos Alberto Escaleira Videira _____

Aluno:

N.º 49318, David de Almeida Ferreira _____

ABSTRACT

ProjectIT is an academic collaborative research project that integrates several graduation thesis, MSc and PhD thesis. Its main research goal is to provide a complete software development workbench, with support for project management, requirements engineering, analysis, design and code generation activities, through the implementation of a set of CASE tools to cover all the stages of an IT product's life-cycle.

ProjectIT-Requirements is one of ProjectIT's subprojects and its purpose is to support the early activities of an information system's life-cycle, namely the ones covered by Requirements Engineering, such as requirements specification and validation.

The main goals of this work are: (1) to develop a text editor plug-in for Eclipse.NET that provides features of a specialized text editor, supporting a predefined requirements specification language (PIT-RSL); (2) to offer guidance during the development process, with debugging contextualized information for erroneously inserted specifications; (3) to generate UML2 models that will be interpreted by the ProjectIT-Studio/MDD tools; (4) to supply a RDF/OWL generator to translate from PIT-RSL to this ontology language for further validation with an inference-engine; and (5) to support requirements storage and reuse with the respective import and export operations. Therefore, this project will endow ProjectIT-Studio with a tool that not only supports the capability of assisting the requirements engineers' activity, but that can also be used by non-technical stakeholders during the requirements elicitation and specification activities.

In terms of adopted technology, it's necessary to emphasize the technologies predetermined in this project's proposal: .NET framework, C# programming language, Lex and Yacc like compiler generation tools, XML, and Eclipse.NET platform. Influenced by the research nature of this project and the results gathered during the research phase for Natural Language Processing (NLP) techniques, this project's scope moved towards wider research fields such as AI and Semantic Web topics (ontology XML based technologies, such as RDF/OWL language, associated with knowledge-base and inference-engine components).

KEYWORDS

Requirements Engineering, ProjectIT-Requirements, PIT-RSL, Natural Language Processing.

ACKNOWLEDGEMENTS

I would like to demonstrate my gratitude to all my teachers, especially the ones responsible for the disciplines related with Information Systems, who have provided me with the necessary knowledge and motivation during my degree in IST.

From the previously mentioned teachers, I'm most thankful to my supervisor, Alberto Silva, who proposed me to integrate this project, and to my co-supervisor, Carlos Videira. Both of them have oriented my work and provided me guidance during my research. Thanks to them, it was possible for me to participate and develop this collaborative research project of the GSI at INESC-ID. In addition, I'm most grateful for the given opportunities of being involved in research and scientific research activities, such as elaboration and submitting papers to renowned Iberian and International Conferences, and also the entrustment to represent the ProjectIT team in three conferences of the four papers accepted, of which I'm a co-author.

Besides the role assumed by teachers during the learning process, I think it's important to recognize the importance of the work environment and collaboration of fellow classmates in knowledge sharing and as an engineering skills enhancement driver. Therefore, I also want to show my gratitude to my colleagues and friends, namely João Saraiva (who has naturally assumed the team coach and technological tutor roles from the beginning, due to the authorship and consequent expertise in the Eclipse.NET project) and Rui Silva, for their support and friendship over this last year.

I also would like to show my appreciation for my family's and my girlfriend's support, especially in the most difficult moments. Without their comprehension and patience, it would be difficult to support the heavy weight of the involved work in this project.

I have only reached this stage in my academic life thanks to my parents, who have supported and provided me with all they could, and to my girlfriend, Catarina, that has always motivated and showed me the positive side of things and gave me the courage to face critical stressful situations. To these most loved ones, my regret for the little availability and few time devoted to them over this last year of hard work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACE Attempto Controlled English

API Application Programming Interface

AUP Agile Unified Process

CASE Computer Aided Systems Engineering

CDT Eclipse C/C++ Development Tooling

CIRCE Cooperative Interactive Requirements-Centered Environment

CRUD Create, Read, Update, Delete (typical operations)

DBMS Database Management System

DLL Dynamic Linked Library

DPA Dynamic Programming Algorithm

DSL Domain Specific Language

EXE Executable (file extension)

GSI *Grupo de Sistemas de Informação*

GUI Graphical User Interface

IDE Integrated Development Environment

INESC-ID *Instituto de Eng. de Sistemas e Computadores, Investigação e Desenvolvimento*

JAR Java Application Resource (file extension)

JDT Eclipse Java Development Tools Subproject

JLCA Java Language Conversion Assistant

JVM Java Virtual Machine

LL Left-to-right scan, leftmost derivation, top-down parser

LALR(1) Left scan, rightmost derivation, bottom-up parser, one look ahead token

MDA Model Driven Architecture

MDD Model Driven Development

MVC Model-View-Controller Architecture Pattern

NL-OOPS Natural Language – Object-Oriented Production System

NLP Natural Language Processing

OMG Object Management Group

OWL Web Ontology Language

PDE Plug-in Development Environment

PIT ProjectIT

PIT-MDD ProjectIT-MDD (Model Driven Development)

PIT-RSL ProjectIT-RSL (Requirements Specification Language)

PSP Personal Software Process

RDF Resource Description Framework

RDQL Resource Description Query Language

RSL Requirements Specification Language

SDK Standard Development Kit

SPARQL Simple Protocol and RDF Query Language

SWT Standard Widget Toolkit

UML Unified Modeling Language

VSIP Visual Studio Industry Partner Program SDK

W3C World Wide Web Consortium

XIS eXtreme modeling Interactive Systems

XMI XML Metadata Interchange

XML eXtensible Markup Language

XP eXtreme Programming

XSD XML Schema Definition

XSLT eXtensible Stylesheet Language Transformations

1 INTRODUCTION

1.1 Preamble

Despite the resemblance with conventional engineering disciplines and the efforts made during the last few decades, the software development process of information systems is still an immature activity and suffers from several critical issues that seriously compromise their success, namely in terms of planning and control activities of project management's core dimensions, such as time, cost, and quality of the software product delivered to the final client and end users. The main cause to this situation is related with the fact that the majority of IT projects do not follow a structured, standard and systematic approach like the methodologies and best practices proclaimed by Software Engineering community [Silva2001].

Facing this problem, the ProjectIT initiative [Silva2004] proposes to minimize the negative consequences mentioned above by introducing a set of innovative concepts, being its main objective to provide a common software workbench environment, created by keeping in mind Software Engineering's best practices and simultaneously being methodology-independent. This initiative's main goal is to research new approaches to accelerate the underlying process by automating its repetitive activities, which are prone to introduce errors and unnecessary complexity in the final product, therefore globally improving development process quality.

Bearing in mind that (1) the early problems' detection strongly reduces the inherent costs of only detecting them in further phases of the process, which can represent budget skews several orders of magnitude greater than the value initially planned, and (2) that the software development process most of the times begins with the requirements specification activity, it becomes clear that one of the most important phases on an IT project consists in the Requirement Engineering activities' phase (comprising the requirements specification, verification, and validation activities), since the resulting artifact from this phase (the requirements document) reflects the client's expectations and needs about the resulting system. The correct interpretation of the stakeholders' requirements is critical for the success and quality of the IT project. Although being of extreme importance, this phase continues to be overlooked nowadays and the existent tool support (commercial solutions and open source projects) still lack the ability to interpret and automatically validate the meaning of systems'

requirements, specified with the most common mean of communication used by humans, the natural language.

After identifying this clear disregarding of the inexistence of a suitable tool support for the Requirements Engineering activities, associated with the volatility of the stakeholders' mental conception of the system and the inherent difficulty of establishing an efficient traceability between the artifacts produced in the several phases of the development process, led the Information System Group (GSI) of INESC-ID, designated as GSI for short this point forward, to the idea of developing a common platform for a complete software development workbench that incorporates a series of other CASE tools developed separately, but sharing a common goal: assisting the software engineer during the developing process, independently of the methodology adopted and covering the entire software product life-cycle. Therefore, the main objective of this innovative and collaborative initiative, named ProjectIT (PIT for short), is to aggregate these tools in a common software development workbench, emphasizing their common goal through the creation of synergies between them for a deeper level of integration, such as automating and streamlining the underlying process's workflow (thus avoiding user intervention in non-productive time-consuming activities, which are typically the process's major bottlenecks) and providing a common, normalized, and standard data format (OMG's widely adopted industry standard UML model).

After this brief contextualization, the relevance of this "small" step in the entire ProjectIT initiative (in which there is a process of communication and knowledge transmission between the stakeholders, the requirements engineers, and the designers) becomes clear. Each small glitch in the process of domain representation and business model design corresponds to a cascade of accumulative costs across the product development life-cycle, causing late misinterpretation detection consequences to become several orders of magnitude greater (in terms of cost and impact) than detecting them in the early phases of the development process, such as requirements specification. Therefore, the Requirements Engineering research topics and the approach of the ProjectIT proposal seeks to reduce the negative aspects of the software development process previously mentioned by introducing a controlled natural language for requirements specification, the ProjectIT-RSL (PIT-RSL for short), based on the identification of the most common linguistic patterns of requirements documents, and a

specialized CASE tool whose underlying parsing mechanisms provide the on-the-fly validation required to determine the specification's consistency in terms of syntactic and semantic rules of PIT-RSL language, hence allowing a deeper integration level with MDD tools.

1.2 Theoretical Problem Context

The main problem that this project proposes to solve is the lack of a requirements specification language and a corresponding CASE tool among the available commercial tools. Nowadays, the main objective of requirement management tools is to provide functionalities that assist the users during the requirements specification activity. However, they are essentially focused on tasks such as traceability, check lists validation, and requirements dependencies, ignoring everything related to their contents. These tools consider requirements as "black boxes" disregarding their semantic information, that is to say, they offer a large set of interesting features but make the mistake of leaving out of their plans the core issue of requirements specification and validation activities: the analysis of their meaning.

The main cause to this situation derives from the urgent need of a specification language that encompasses the best of both worlds: (1) the expressive power and familiarity of natural language, and (2) the strict meaning and machine processing abilities of formal language approaches. The latter has its range of application limited mainly to mission-critical projects where the correctness and robustness of formal methods are crucial. Having this premise in mind, software companies have always tried to avoid until the limit the need of taking the required step of adopting natural language processing (NLP) techniques for interpreting the requirements' meaning. Albeit its inherent processing problems, the former is the mainstream language for requirements specification thanks to its familiarity and consequent ease of learning.

This work, as a few others (which will be presented in Chapter 2), attempts to establish new boundaries by creating a requirements specification language, based in natural language, and to develop a specific text editor for requirements specification, that provides the user with means for performing analysis, validation, and verification, based on graphical views generated on-the-fly. The main objective is to provide the possibility for non-technical users

(the domain experts) to specify themselves their own system's requirements, thus minimizing the typical gap between the domain knowledge expert and the system's designer. By lessening the mediation role of the requirements engineer in this error-prone bridging/interpretation activity, it's possible to streamline and accelerate the underlying process of information specification and respective disambiguation. This kind of approach will endow the requirements engineer with the necessary tool support for the activities under his/hers responsibility. The further integration with ProjectIT will represent the achievement of a complete tool that covers all the stages in the product development life-cycle, allowing anyone to develop a system almost automatically, based on the natural language free-form text requirements, created by the non-technical stakeholders.

1.3 Main Goals to Achieve

The two main objectives of this work are: (1) to define, design, and specify a set of linguistic pattern-based rules for a new requirements specification language, called ProjectIT-RSL; and (2) to create a CASE tool, built upon the Eclipse.NET plug-in architecture, to eliminate or to bridge the gap between stakeholders' view of the system and the representation conceived by the requirements engineer. To attain this ambitious goal, it is necessary to build a tool that, by applying the above mentioned linguistic rules, can analyze requirements documents through the usage of parsing techniques that extract computer-understandable information for further validation and implicit knowledge extraction. Therefore, PIT-RSL will introduce a set of features that the majority of commercial tools lack, since their philosophy is to treat requirements as "black boxes" without looking at their content [Ambriola2003]; consequently their features are based on control versioning systems and traceability, but without considering semantic analysis and consistency verification for example.

To achieve this goal, we have to analyze the best and most common medium of message transmission between humans. Several studies reveal that [Mich2002], despite some undesirable characteristics such as ambiguity, imprecision, and incompletion that make it difficult for computer processing, natural language is by far the most common and easy form of communication between the stakeholders and the requirements engineers. Another one of the advantages of natural language for this type of tools is the familiarity and easiness of using and understanding it, avoiding typical long and difficult learning curves of formal

approaches. As a result, it's possible for the stakeholders themselves to specify, reason, and validate the requirements, since the only thing they must do is to write the requirements in English plain text documents. ProjectIT-Studio/Requirements is then responsible for the analysis and interpretation of the free form text and to produce the views and perspectives for helping the user in the verification and validation tasks. Consequently, the tool forces the user to adopt an interactive approach [Ambriola2003] that is the most suited for requirements specification, since it's based on the continued iterative refinement of the actual document, until the description of the system (in terms of requirements) corresponds to the mental representation of the stakeholders.

1.4 Methodology

The methodology adopted for this work was the one proposed by my supervisor, called "TFC Process" (its description is available at http://speedy.inesc.pt/ProjectIT-Enterprise/Processes). This approach is mainly composed by two major phases: (1) the *research phase*, which consists in a thorough analysis of the current state of the art related to the main topics of the work to be developed, thus providing solid theoretical foundations to support the system's design and architecture rationales; and (2) the *implementation phase*, which is focused on the concrete development of the software product based on the knowledge (ideas, concepts, and design/architecture) gathered in the previous phase of this methodology.

During the former phase, besides the exhaustive research undertaken on the most common features of generic text editors and requirements management commercial tools, followed by a deep study of natural language processing (NLP) techniques, algorithms, related projects, and other associated fields belonging to the Artificial Intelligence (AI) scope of research, like knowledge representation by ontology definition and manipulation techniques; the first parsing mechanisms prototypes were developed (one for requirements documents structural analysis and the other for the fuzzy matching mechanism), as well as a simple text editor plug-in for the Eclipse.NET platform.

The latter phase focused mainly on the enhancement of the parsing mechanisms developed during the research phase and the improvement, through GUI features addition, of the already developed simple text editor plug-in. At the same time, during the implementation tasks, all

new ideas and knowledge that appeared were deeply explored to extract new techniques, find related tools, and to guide the development process which was strongly dependent on the associated underlying research characteristic of this work.

Since this project was initially proposed with the intention of serving as a base for a MSc degree thesis, it has been assigned and developed solely by a single student. Therefore, there was no need of planning a cooperative work between two students as usual. The only interaction verified during the project was between the fellow colleagues that have developed previous work and with the other ProjectIT team members (two colleagues) that were also developing components which, in the end of this project, were also integrated in the ProjectIT-Studio tool via its plug-in architecture.

In parallel with the typical software development process adopted, the GSI performed some periodic briefing workshops with the participation of all the students working on GSI department's projects at the moment, as a way to monitor the undergoing projects' progress. An additional level of progress control was achieved via sporadically *ad hoc* meetings and brainstorms (reflection about the work performed and the best orientation to follow given the recent discoveries) between the ProjectIT-Studio development team members and their supervisors (face-to-face conversation is the best form of communication [Jeffries2001]).

Despite being conformant with the previous enunciated methodology, there was the extra concern of complementing the followed software development process (since this was a risky project due to the disruptive approach adopted, which is explained further in this report) by incorporating several of the best practices widely accepted and proclaimed by the Software Engineering community, most of them coming from the emerging agile approaches like eXtreme Programming (XP) [Jeffries2001] and Agile Unified Process (AUP) [Ambler2006] software development processes (also called "lightweight methodologies").

The best practices adopted to enrich the original methodology with these agile processes' core concepts were: (1) usage of a metaphor, which synthesizes the system's ultimate goal and is easily understood by everyone; (2) continuous delivery of useful and working software instead of documentation (the best measure for determining project's progress); (3) all requirements changes are welcome, even in late phases; (4) continuous attention to technical excellence and good design (via Test-Driven Development); (5) simplicity of the

implemented solution; (6) sustainable development at a constant rhythm; (7) frequent adaptation to changing circumstances instead of following a rigid plan; (8) continuous refactoring and integration; and (9) usage of coding standards.

The major advantage of using these agile best practices, besides being based on iterative and incremental processes, is to achieve a greater productivity and, simultaneously, to gain the capability to rapidly respond and surpass the challenges and obstacles that can jeopardize the project, namely the ones resulting from (1) constant changes in the requirements specifications, (2) the knowledge available about the system, and (3) the surrounding environment (organization, department, or team specific goals in this case).

The reason for not fully applying one of these approaches is simple: it would be inappropriate for this work bearing in mind that it was mainly a research project (not involving interaction with real-world clients) without clearly defined requirements specifications (the major guideline was the vision/metaphor of creating a tool for "writing natural language free-form text requirements documents as one can use Microsoft Word to write text"). Therefore, the evolution of this work was dependent on the results achieved with the research performed crosswise to the development activities. Additionally, there was neither team work nor notorious coordination effort among team members, and the time-frame was relatively short for adopting one of these industry-oriented software development processes.

1.5 Case Study – MyOrders 2.0

The present work was validated with a simple case study called MyOrders 2.0. This example has been continuously used across the entire ProjectIT initiative's history to provide a concrete proof-of-concept of the implemented tools functionality. The suggestive name of this software system model reflects its main goal, i.e., it focuses in the design and implementation of an information system to support an abstract scenario of interaction between clients and suppliers, namely: (1) authentication; (2) business entities management (typical CRUD operations for clients, suppliers, and products); and (3) client-to-supplier roles detailed product orders. To provide a better insight of this case study it was reserved an entire chapter to explain it. Chapter 5 presents a detailed description of this academic system's example and demonstrates the achieved results.

Bearing in mind that this work's spotlight consists in providing an adequate tool support for the early phases of the software development process (specifically Requirements Engineering activities, such as requirements specification and validation), the MyOrders 2.0 example was essentially used to provide a concrete domain model to be specified through the usage of ProjectIT-RSL language and its supporting CASE tool, the ProjectIT-Studio/Requirements.

The process adopted to specify this system was very simple: initially the domain model was analyzed to detect eventual translation problems due to concepts that are not clearly mapped from the UML eXtreme modeling Interactive Systems 2.0 (XIS2) to the ProjectIT-RSL metamodel. This issue occurs since there is an inherent information loss during the passage from an application's domain-specific knowledge language (PIT-RSL specification) at the MDA's CIM level – Computation Independent Business Model – to the respective MDD templates at the MDA's PIM level – Platform Independent Model. After this initial analysis, the translation process of MyOrders 2.0 system, passing from an UML graphical notation to a PIT-RSL textual requirements specification, was straightforward.

1.6 Document's Organization

This document is structured as follows: after introducing a brief context and motivation, for the ProjectIT and its subprojects (namely ProjectIT-Requirements and the respective PIT-RSL language, the focus of this document) in this introductory chapter (*Chapter 1*), the reader will be presented with the state of the art of the topics of research and commercial tools related with this project in *Chapter 2*. Each section of Chapter 2, is focused on the areas of research carefully chosen, revealing the progressive path of research topics analyzed to develop this work. They are deeply described in the relevant areas for PIT-RSL and ProjectIT-Studio/Requirements tool, once they influence the concepts, algorithms and technologies adopted. *Chapter 3* presents the ProjectIT initiative, providing a thorough contextualization of this ground base project. *Chapter 4* describes all issues related with ProjectIT-Requirements analysis and design. The architecture's description is provided in *Chapter 5*, enunciating the supporting rationales. *Chapter 6* is devoted to case study undertaken as a proof-of-concept of this tool. In the end, a thorough analysis of the work performed in the context of ProjectIT-Requirements and a brief description of the planned future work (DEIC's PhD scope) are presented (in the last two chapters, *Chapter 7* and *8*).

This document provides a comprehensive set of appendixes, allowing the reader to gain a deeper understanding of issues that are only superficially analyzed in the body of this document, due to the structure and format restrictions imposed for this graduation thesis report.

2 STATE OF THE ART

In this chapter one will be presented with the results from the initial research phase of this project. Having in consideration the peculiarities of the Natural Language Processing (NLP) techniques involved, most of them related directly or indirectly with Artificial Intelligence (AI), this research phase was somehow more extended then it was expected from the very beginning. This additional effort was crucial to gather a considerable amount of specific knowledge out of this research area. The insight gained in this field allowed to achieve a deeper understanding of the underlying techniques and algorithms required to correlate and choose the best solution for the problem under discussion among the available alternatives.

2.1 Requirements Engineering

During the last few decades we have assisted a growing concern about the Requirements Engineering discipline and related techniques, given their importance to the success, namely in terms of quality, of the final software product. As previously mentioned in the preamble of this document, the main cause to the software development process failure is related with the fact that the majority of IT projects do not follow the best practices proclaimed by Software Engineering [Silva2001]. This problem gets worse with the growing dimension and complexity of modern applications.

Bearing in mind that early detection of errors in the upstream phases of the product's life cycle allows minimizing their cost and effort impacts [Cibulski2001], it's straightforward to comprehend the growing importance of requirements engineering, from which we emphasize: (1) requirements reuse, with still minor tool support but crucial for mitigating repetitive error-prone work; (2) alignment with industry standard, widely-adopted OMG's languages and standards; (3) integration of validation techniques of formal methods based upon mathematical notations, such as Z [Spivey1992] and VDM [Jones1990], to leverage the quality of requirements specifications by enhancing their rigor and correctness; and (4) assimilation of the best practices of agile approaches [Beck1999], to simplify the process of information gathering of the system functionalities, by objectively capturing only the sufficient and necessary user needs.

To attain the best from both of these worlds, like smoothing the non-technical users' learning curve, some authors proposed the use of a "controlled natural language" for requirements specification [Fuchs1996a]. Although not formal, it facilitates the process of writing, analyzing and verifying requirements [Johnson1991]. This concept's ultimate goal is to obtain machine-computable requirements specifications to support systems' simulation [Fuchs1996b], while still allowing stakeholders to validate them by using natural language.

During the 90's several research projects sharing ProjectIT-Requirements vision emerged, namely: (1) CIRCE [Ambriola2000b] [Ambriola2003], focused on analysis flexibility and validation; (2) ACE [Fuchs1996b] [Fuchs1998], a controlled language for requirements formal specification; (3) NL-OOPS [Mich1999] [Mich2002], a framework for generating models from natural language requirements documents. These reference projects are further detailed in this chapter.

2.1.1 OMG's MDA Approach and Standards

The ProjectIT initiative goals of being able to provide software development process's automation, and also industry standard compatibility and interoperability, report to OMG's approaches and standards, from which is essential to refer the MDA approach. Therefore, it's important to take into consideration some concepts to understand the ProjectIT's inherited goals and integration objectives for the ProjectIT-Requirements component.

From the model driven engineering (MDE) perspective it is important to stress the following concepts: (1) MDA [OMG2003], the OMG's framework for software development life cycle driven by the activity of modeling [Kleepe2003], making models first-class entities; (2) MOF [OMG2006], the foundation of OMG's approach, supporting model exchange and transformations which can be applied to any modeling language, as long as it is MOF-based; (3) UML [OMG2005a], a general-purpose modeling language, originally designed to specify, visualize, construct and document information systems (nevertheless, it isn't restricted to software's scope, being often used for business process modeling); (4) the XMI format [OMG2005b], a standard commonly used to exchange UML models between tools, although it can also be used with other MOF-based metamodels; (5) the MOF QVT [OMG2005c], a standard under finalization for defining query, view and transformation operations on MOF-

based models, effectively allowing the transformation of source models into target models, being a critical component for MDA. Figure 2.1 presents the employment of the MDA paradigm to the ProjectIT approach.

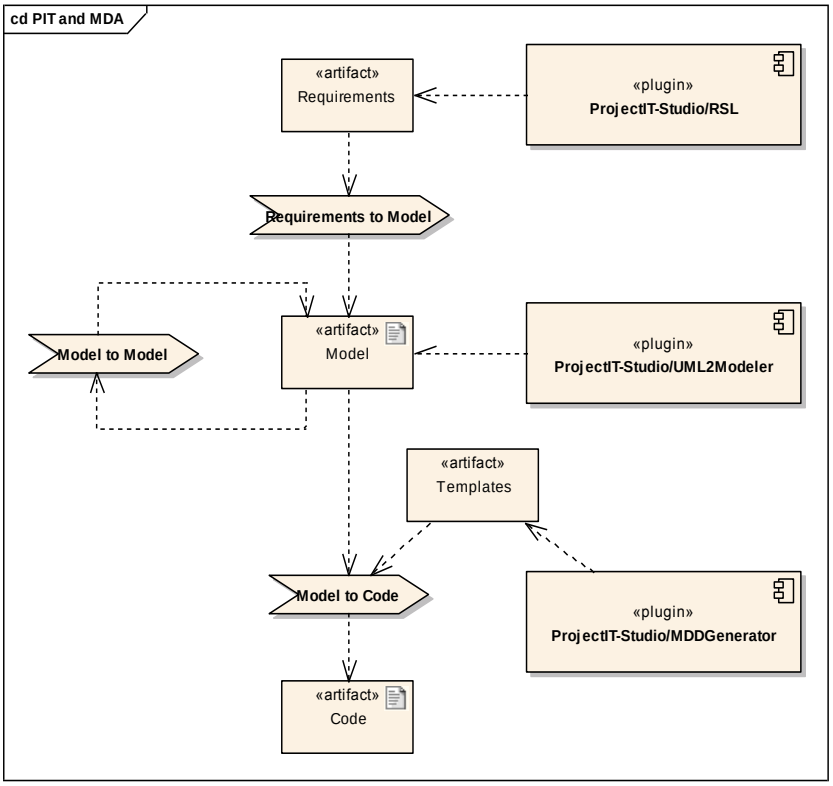


Figure 2.1 – ProjectIT and the OMG's MDA Paradigm.

2.2 Commercial Tools

Nowadays, there are several commercial requirements management solutions available This section's purpose is to present the gathered information about these solution's main features and their integration level with external and proprietary tools (typically these solutions comprise several tools from the same vendor).

2.2.1 Requirements Management Tools' Features

Related to this subject, was performed research about the main features of the requirements management tools available. The purpose was identical to the subsection above: to find the most desirable features of these tools to incorporate them in this project. The final analysis

was based on an exhaustive study [INCOSE2005] that presents detailed information about each tool and the relation between them in terms of features. Some of these applications are well-known and deserve to be mentioned, such as Analyst Pro, CaliberRM, DOORS, and IBM Rational RequisitePro. The comparative survey is recent, providing information mainly from 2004's last quarter and middle of 2005.

The most relevant aspects of this research are synthesized below in the following items, each corresponding to one of the major classes of features provided by these commercial tools:

- **Requirements capture and its univocal identification**: this feature includes input document enrichment and analysis with functionalities such as automatic unique identification according to the requirement context, requirements versioning, comparison, and merging. This category of features includes also the so called "automatic requirements parsing" that consists in import and export operations from requirements documents written in Microsoft Word (some tools also support other text documents formats). All these features are presented with fancy characteristics variants like manual vs. interactive import with requirements highlighting, batch mode operation, and requirements classification with built-in and user-defined attributes schemas.
- **Capture system's elements structure:** this is achieved by providing built-in diagrammatic tools and accepting diagram formats from external tools. This category of features allows to graphically capturing system's structure, architecture, and functional decomposition, thus allowing the user to establish links between the diagrams and textual requirements.
- **Establishment of requirements relations:** this class of features encompasses the ability to establish hierarchies and relations between requirements (and their predecessors and successors) or other related digital artifacts such as diagrams, text documents and graphics. It involves also access control mechanisms for restricting viewing/edition functionalities.
- **Traceability analysis:** this category intends to provide consistency checking such as detecting orphans and their type with reports and matrixes, allow navigation through links

between related artifacts, requirements' implementation verification, and requirements summarization measures by using data marts.

- **Configuration management:** this class offers functionality related with typical control version systems such as change history, requirements versioning, auditing information, rollback to a previous stable version, and requirements locking. This encompasses also control access rights for viewing and modification.
- **Documents and other output media:** these tools also support features such as automatic reports, graphs, and presentations generation for process status reporting, and also support typical queries and searches by filtering and grouping features. These tools offer functionality for quality verification and consistency checking by using glossaries (on a global or project base) and traceability matrixes.
- **Groupware:** this category includes collaborative forum support (some include a specific API for integration with third-party groupware tools), includes also update conflicts, concurrent revisions avoidance features, concurrent review, mark-up, comment, multi-level assessment and control (e.g. project, requirements, and attribute levels).
- **Interfaces to third-party tools:** this class covers the interchange capabilities of these tools such as inter- and intra-tool information exchange, APIs provided to connect with external tools, support of commercial and open source DBMSs, support for data exchange format standards, and consistency checking between datasets of the same tool.
- **System environment:** these group of features relate essentially to the tool's portability, architecture (e.g. client-server), concurrency (e.g. multi-users), and OS compatibility.
- **User interfaces:** these set of characteristics relate to the interfaces provided for user interaction and access easefulness and usability such as web-browser interface, Microsoft Windows look and feel, views, and macros. One of the presented features that deserve to be emphasized is multi-level UNDO/REDO.
- **Standards compliance:** it's important for this class of tools to be compliant with the industrial/commercial standards since they cannot survive alone in the marked detached of other complementary tools, since the work performed with them needs to be continued.

The steps attained with these tools correspond only to the software engineering (SE) early phases of a product life-cycle.

- **Training:** this aspect is crucial since accentuated learning curves can jeopardize the tools mainstream acceptance by the final users.

2.2.2 Requirements Management Tools' Integration

Focusing on the integration level of current commercial tools, such as the Telelogic solution's duo DOORS / TAU, and the IBM solution's trio RequisitePro / Software Modeler / Software Architect, we can observe that this category of CASE tools already present a mature and deep integration, enabling engineers to enforce traceability between requirements, models, and source code, by using a common workspace environment.

Telelogic's requirements-driven development solution, the DOORS/TAU role-based integration allows fine-grained traceability of models, and direct navigation between them. DOORS and TAU users can visualize and associate models and requirements, respectively. The adoption of UML 2.0 simplifies the communication throughout the development process, and the tool's deep integration allows to quickly evaluating the impact of any requirements change. Additionally, DocExpress provides automated reporting and documentation services.

IBM's offer for requirements and analysis products is: (1) Rational RequisitePro, an integrated solution for requirements and use case management; (2) Rational Software Modeler, an UML 2.0-based visual modeling and design tool; and (3) Rational Software Arquitect, an integrated design and development tool that leverages model-driven development with UML for creating well-architected applications.

Rational RequisitePro supports project teams' collaborative work, promoting better communication of project goals, reducing project risk and increasing the quality of applications. It provides advanced integration with Microsoft Word (supporting synchronization) to endorse the tool with a familiar environment for activities such as requirements definition and organization. Rational Software Modeler is an essential tool for ensuring that specifications, architecture, and designs are clearly defined, documented and communicated to stakeholders. Rational Software Architect allows architects and developers to create applications for target platforms, leveraging model-driven development with UML,

and unifying all software life cycle's aspects. This suite of tools covers all the facets of the software engineering process life cycle, improving accessibility and communication of requirements and models. Additionally, the Rational SoDA integration provides the ability to generate reports and documents from artifacts.

2.3 Features' Analysis of Generic Text Editors and IDEs

2.3.1 Motivation

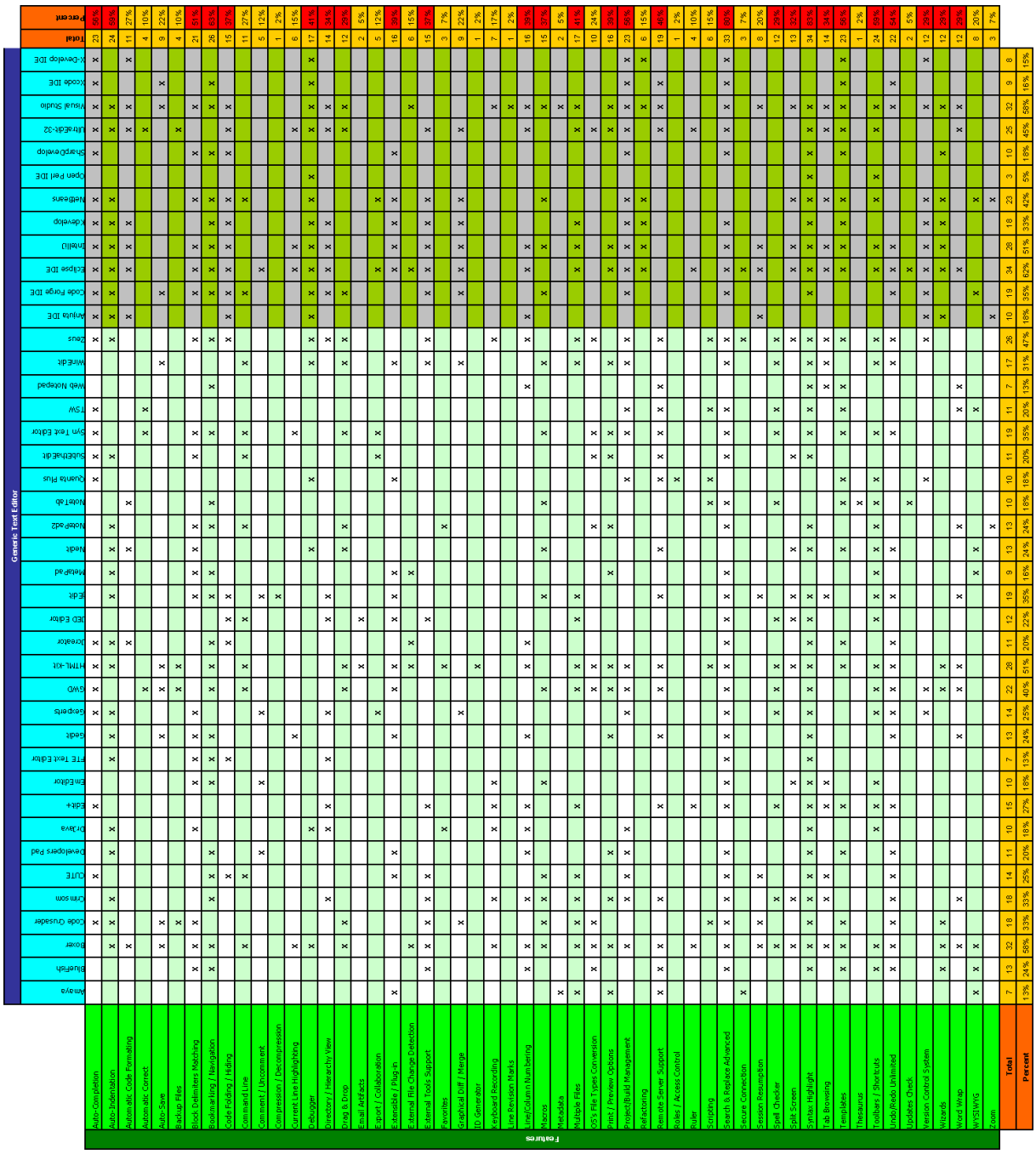
This was the first research topic explored, followed by the research of commercial tools features. This research consisted in the analysis and gathering of the most common text editor features, including also features of the mainstream IDEs. The purpose of this crucial activity was to collect a considerable set of features and their respective desirability, to have a 0concrete reference to numerically support the adopted rationales when specifying the PIT-RSL plug-in features, endowing it with the "most wanted" and useful features of text editors.

2.3.2 Research and Results

This research covered the analysis of several text editors and IDEs, performing a total of forty-one tools: twenty-nine text editors and twelve IDEs, from which it's relevant to highlight products as Anjuta, Eclipse, IntelliJ, NetBeans, SharpDevelop, Visual Studio, KDevelop, UltraEdit-32, Xcode, and X-Develop.

From the technical sheets of these products, fifty-five distinct relevant features were identified. The following table, Table 2.1, presents a compiled view of the information gathered during this research. The shadowed area corresponds to the IDE tools analyzed, whereas the other columns correspond to simpler generic text editors.

Table 2.1 – Generic Text Editors and IDEs Features' Analysis.



To draw some conclusions it was required to synthesize the exhaustive gathered information. The synthesized results, the top 10 features, are presented in Table 2.2, emphasizing the most desirable features to incorporate in this work.

Table 2.2 – The Most Desirable Text Editors' Features.

		Fx(i)	Percentage
Features	Syntax Highlight	34	83%
	Search & Replace Advanced	33	80%
	Bookmarking / Navigation	26	63%
	Auto-Indentation	24	59%
	Toolbars / Shortcuts	24	59%
	Auto-Completion	23	56%
	Project/Build Management	23	56%
	Templates	23	56%
	Undo/Redo Unlimited	22	54%
	Block Delimiters Matching	21	51%

2.3.3 Analysis and Hypotheses Comparison

Despite the information gathered during the research phase about these most frequent features (and, consequently, the most desired ones) it is necessary to bear in mind: (1) the original work's specification; and (2) technical limitations and the already specified behavior of some components of the adopted common platform of ProjectIT tools, the Eclipse.NET. Relative to the former, the main goal was to design a tool that offered the same functionality of the already implemented functional prototype [Carmo2005] (that is going to be analyzed further in this subsection), which was mainly the typical features offered by VSIP extensibility mechanisms, i.e., auto-complete (intellisense), full syntax highlighting and graphical error annotation (text underlined with wiggly lines), both based on a programming language grammar (ProjectIT-RSL was initially implemented by using a usual context-free grammar). Since these features are included in the most common text editors' features, this research reinforces the initial goal of implementing them. Focusing on the latter's issue (technical

limitations of the Eclipse.NET platform), the majority of the pointed out features are supported by the original Eclipse framework. However, there are some minor issues related with the actual state of conversion of the tool and features additions request, namely the usual JDT advanced *Find&Replace* functionality was not converted since it presents several deep dependencies on other JDT plug-ins, and given that the actual *Find&Replace* associated with the *TextEditor* class presents a hard to detect semantic error, this feature was not possible to implement.

2.4 Natural Language Processing and Parsers' Generation

As is widely known, requirements are described by using a specification language. The expressive power of the chosen language must allow conveying the information and knowledge that mirror the real needs of the stakeholders. Above all, it must ease the communication between the participants in the requirements specification activity, since the most difficult challenge to overcome is to minimize the conceptual distance between the model created in the mind of the stakeholders and the model perceived in the mind of the requirements engineers.

Focusing on requirements specification languages categories, there are two main approaches: (1) the set of formal languages, strongly supported by mathematical and logical concepts from which results a formal notation; and complementarily (2) the set of non-formal approaches, which derive from natural language variants or result from visual modeling techniques.

The solution proposed to solve the main underlying problem of this graduation thesis, that will be further detailed in the respective chapters of this document, seeks to combine the best from both worlds: (1) the benefits of natural language usage, namely familiarity, ease of learning, and expressive power; and (2) the rigor and validation from formal approaches, resulting in the concept called "controlled natural language" that is no more than a subset of terms and syntactic structures with restricted semantics defined for specific contexts, usually called domains, enabling this way tools support, that is the foundation of any DSL [DSMForum] [MartinFowler].

Influenced by the research nature of this project, the research phase has focused on three major existing natural language processing (NLP) projects with similar goals to this work,

from which have been gathered several interesting ideas to adapt to the ProjectIT-RSL and ProjectIT-Studio/Requirements tool during the implementation phase.

Therefore, taking into account the specificity of Natural Language Processing (NLP) techniques, during the initial research phase there was the concern of following the theory lectured in Natural Language discipline of LEIC course, whose main bibliography is [Jurafsky2000], for gaining some background in this area and extracting only the necessary concepts and techniques for the development of this project. It's important to emphasize the in-depth study that was undertaken when researching morphologic and stemming algorithms for the initial phases of the functional prototype. It was rewarding to analyze the main characteristics of most well-known algorithms, such as finite-state transducers, minimum edit distance (dynamic programming DPA, also known as Levenshtein), Bayes Rule (stochastic), Viterbi Algorithm (DPA and stochastic), N-Grams (stochastic), and cascaded automata, which are more oriented towards error correction and Information Retrieval (IR).

The adopted bibliography [Jurafsky2000] was used essentially as a reference for disambiguate and support the rationales that fundament the decisions taken during the design and implementation phases. The guidance achieved by consulting this book was of major importance and worthwhile for providing the variety of concepts that served as a starting point for further research in the Internet. Besides the initial intensive contribution, to rapidly gain a good insight in NLP concepts and techniques, this book continued to be a reference during the subsequent phases of this project, namely during this tool implementation.

2.4.1 Stemmers

During the research of NLP techniques and mechanisms, it was considered relevant the existence of a useful and typical NLP and Information Retrieval (IR) component called stemmer, which allows determining the stem form of a given inflected or derived word form. A stem may not be identical to the morphological root (the lemma) of the analyzed word. However, it is sufficient to establish relations between words sharing the same stem even though the stem may not be a valid root or even an existent word. The underlying algorithm generates a kind of hash value that identifies identical words like the Soundex algorithm but more meaningful.

Despite stemmers for English are easily found (however the effectiveness of stemming for English query systems is questionable), stemmers for other target languages with more complex morphology and grammatical rules become more difficult to build.

Nowadays there are good alternative approaches, namely algorithms based on: (1) n-grams searching, which require previous text corpus training to achieve accurate results; and (2) word lemmatization, a more complex approach that involves first determining the morphologic analysis of a given word and then applying different normalization rules for each part-of-speech elements (lemma and affixes).

The first mainstream adopted stemmer was written by Martin Porter in 1980, and given its widely adoption it has became the de-facto standard algorithm used for English stemming. Since then, many implementations of this algorithm were written and freely distributed. However, many of these implementations contained subtle flaws, and as a result systems using these stemmers performed less well than they ought. To eliminate this problem, Martin Porter released an official free-software implementation of the algorithm which culminated with the Snowball project (one of the analyzed projects).

The stemmers' projects that were analyzed in this work's context were: (1) Nice Stemmer, a stemmer component composed by four other specific case application stemming algorithms, namely Simple stemmer, Porter stemmer, Inflectional stemmer (Krovetz), and an invented combination stemmer (available at http://ils.unc.edu/yangk/nstem.htm); (2) Lancaster, which implements three widely known algorithms, namely Porter, Lovins and Paice/Husk, and two other non-mainstream algorithms (Dawson and Krovetz) (available at http://www.comp.lancs.ac.uk/computing/research/stemming/); (3) Snowball, a framework for writing stemming algorithms, which implemented an improved English stemmer together with stemmers for several other languages (to gain a deeper insight in this project, the reader may consult the official web site, http://snowball.tartarus.org/).

The explanation of these algorithms is beyond the scope of this work, since the goal was to analyze the available stemmer components to evaluate the feasibility of their adoption.

The adopted project was Snowball due to its wide adoption and, consequently, available community support. Another reason for this decision refers to the effort of adapting the other

analyzed projects to this work, emphasized by the inexistence of concrete evidences that they presented better performance and accuracy than Snowball.

2.4.2 Part-of-Speech Taggers

Another crucial natural language processing component is the morphologic analyzer, usually known as part-of-speech tagger. This component analyzes natural language words and determines their grammatical classes, based on both its definition as well as its context (for instance, its relationship with adjacent and related words in a phrase, sentence, or paragraph).

The objective is to use computational linguistic algorithms to identify words categories, (typically eight, namely: (1) nouns; (2) verbs; (3) adjectives; (4) prepositions; (5) pronouns; (6) adverbs; (7) conjunctions; and (8) interjections) and associate them with discrete terms, in accordance with a set of descriptive tags. However, due to the frequent natural language ambiguity issues (morphologic ambiguity, to be exact, in which a word can represent more than one different grammatical classification), this word's cataloging process isn't as trivial as it might appear at first glance. Due to natural language's inherent ambiguity, often, in practice, there are much more categories and sub-categories than the eight previously mentioned, depending on the natural language processing and recognition problem at hands. Usually, when considering part-of-speech tagging by using an algorithm, there are typically about 50 to 150 different possible word classifications (tags) for English (like the tags [Treebank1999] used in the Brown Corpus project [W3-Corpora1998]). The underlying tagging mechanisms involve stochastic models and algorithms.

This research branch focused in the following two part-of-speech tagger: (1) TreeTagger tool (the tagger used by CIRCE's shallow parser – CICO), belongs to a broader project called TC (available at http://www.ims.uni-stuttgart.de/projekte/tc/) and it supports lemmatization and has been used to tag several languages (German, English, French, Italian, Spanish, Bulgarian, Russian, Greek, Portuguese, and French) being easily adaptable to other languages if a lexicon and a manually tagged training corpus are provided (it is available at http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/DecisionTreeTagger.html); and (2) Brill Tagger, which is an error-driven transformation-based tagger (available at http://www.cs.jhu.edu/~brill/). The Brill Tagger's algorithm (the solution adopted) is possibly the most well known part-of-speech tagger algorithm and there are several implementations

available in different programming languages. It is labeled as "error-driven" since it uses supervised learning and "transformation-based" due the inherent process of assigning a tag to a under analysis word. In a similar way to the former, it also possesses the ability to enhance its accuracy by performing corpus training (for a high-level description of this algorithm the reader may consult http://en.wikipedia.org/wiki/Part-of-speech_tagging).

The adopted solution was a Brill Tagger original algorithm's .NET port (available at http://www.codeproject.com/cs/library/semanticsimilaritywordnet.asp). It was chosen mainly due the following factors: (1) it's widely adopted and there are several implementation projects available; (2) it was written in C# and Basic, thus not requiring any conversion to the .NET Framework; (3) the source code was available; and, finally, (4) it presents several advantages when compared with Hidden Markov Models (HMM) taggers like the previous one (TreeTagger tool), since the generated rule set is easier to understand than the usual numeric transition-weight tables of the HMM taggers. The advantage of this feature is to support the process of manually correcting the automatically induced knowledge captured by the underlying stochastic model, since in most cases it is undoubtedly simpler to change an explicit tag by rewriting rules than by changing probability tables.

2.4.3 Frameworks for Building Compilers and Parsing Tools

This research phase encompassed also the analysis of several compiler and parser tools available and the selection of the most appropriated for this project. Having in consideration one of the technical requirements, the restriction that the implementation language should be C#, the research was automatically narrowed to this condition. There was also the possibility of using tools not natively written in C#, but there was the obligation of it being provided as a valid .NET Dynamic Link Library (DLL), such as Java tools converted with IKVM.NET.

The research was guided by following the specifically created framework's criteria:

- **Parser's class:** this criterion is related with the generated parsers characteristics, that is, if they belong to the bottom-up or top-down architecture. This is important to considerer since they influence the grammars construction and the parser performance. Yacc, the parser recommended in the specification of this project, is a LALR(1) parser, belonging to later class of parsers. From the LR category of parsers, the LALR(1) have the benefits

from both the SLR analyzers and canonical LR: having the same number of states as SLR analyzers and to recognize almost every LR languages [Crespo2001].

- **Functionality:** this criterion is based on the tool's API richness in terms of built-in functionalities such as parsed data manipulation, abstract trees creation and navigation, and parsing errors and exceptions handling.
- **License:** this is a crucial aspect because frequently the final software product has to maintain the same conditions that are specified in the license of the imported component due to legal authorship issues. Although it's an intellectual property protection measure, it imposes restrictions on the eventual commercial purposes of ProjectIT-Studio.
- **Lex & Yacc resemblance:** one of the technology prerequisites of this project was the use of Lex and Yacc. Having in consideration the problems related with the use of unmanaged code, the author decided to search for compliant tools such the ones analyzed in this section. However, the initial similarity with Lex and Yacc notations, is considerably important when considering issues such as familiarity and broad adoption.
- **Documentation:** almost as important as the API's richness is the documentation. Without it, it's extremely difficult to learn the specific notations and extract all the benefits provided by the tool's API.
- **Support and examples:** the support is essential for recent and innovative projects, such as PIT-RSL, for reporting errors and getting feedback in useful time. Examples, as well as the documentation, are critical for a beginner's fast learning process. Products with accentuated learning curves usually are condemned at first place.
- **Implementation language:** having conversion tools like IKVM.NET, this criterion has little impact. However, it depends on further usage intentions that are more related with source code availability for tool's extension with new features and also the overhead incurred by the necessary API's and examples conversion.
- **Project activity:** it's important to adhere to a tool that offers some durability guaranties since ProjectIT encompasses several subprojects such the one that this report focus on. Therefore, this work's future plans implies the adoption of tools that are stable and active

at the moment, and have some background that transmits confidence of continuing active for a relative long period.

- **Popularity:** this criterion is also important since it reflects the broad adoption of a tool. If everyone prefers it, hence hypothetically it has greater support and activity, meaning better documentation, feedback, and most probably is the best tool for the problem.
- **Sources:** as it had been pointed above, the availability of tool's source code is related to the possibility of further enhancement of the tool by the implementation of new specific features leveraging it to a higher level of functionality.

After defining the criteria enunciated above, the search solution space was limited to the following five tools, presented in Table 2.3. The selection criteria are ordered top-down from the most priority to the less important ones considered during the tool selection process. The adopted tool was CSTools framework.

Table 2.3 – Parser Tools Comparative Analysis.

		Parsing Tools				
		ANTLR	C# Cup&Lex	CoCoR	CSTools	Grammatica
Selection Criteria	Parser's class	LL(k)	LR	LL(k)	LALR(1)	LL(k)
	Functionality	Good	Medium	Medium	Good	Good
	License	BSD	Copyright	GPL	N / A	GNU GPL
	Lex & Yacc resemble	Medium	Good	Bad	Good	Medium
	Documentation	Yes	No	Yes	Yes	Yes
	Support / Examples	Good / Good	Bad / Bad	Good /	Good / Good	Good / Good
	Implemt. Language	Java	C#	C#	C#	Java
	Project Activity	Active	Inactive	Active	Active	Active
	Popularity	Good	Bad	Medium	Good	Medium
	Sources	Yes	Yes	Yes	Yes	Yes

2.5 Semantic Networks: Ontologies and RDF/OWL

An important research topic was related with the potentialities and feasibility of adopting the Web Ontology Language (OWL), which is built upon the Resource Description Framework (RDF), as the internal data representation language, inheriting all the advantages of XML format, being all of them W3C's standards. OWL is one of the most recent XML standards for representing metadata, performing an equivalent role to XMI. To gain a deeper insight in the Semantic Web Architecture and supporting languages the reader may consult Appendix B.

The main purpose of this research was the discovery of a robust framework capable of manipulating, querying, and storing RDF/OWL information, performing the role of typical knowledge-base and inference-engine characteristic components of this class of NLP projects.

After checking the expressive power equivalence between UML 2.0 and OWL [ATT2004], the analysis of available frameworks and tools started. The achieved results are presented in the next subsection. The adopted framework to further conversion and integration with ProjectIT-Studio/Requirements was Jena project (for further information the reader can consult the List of Software Applications available in Appendix A).

2.5.1 Frameworks / Tools

- **DRIVE:** is a C# RDF parser for the .NET platform. It's fully compliant with the W3C RDF syntax specification and is available as open source under terms of the GNU LGPL license. DRIVE parses RDF/XML documents and builds an abstract directed linked graph that can be recursively traversed, and can merge graphs from multiple sources. However, despite these features variety and its extensible API, DRIVE has neither a reasoning system, nor a simple querying mechanism. This tool wasn't adopted since it lacks crucial features needed for the PIT-RSL's inference engine. This framework is available at http://www.driverdf.org/.
- **RedLand:** is a set of free software libraries that provide support for RDF. The modular object-oriented C libraries provide a framework with rich API for all sorts of RDF manipulation typical operations, and also parsers and serializers (Raptor RDF Parser Toolkit), persistent storage in different formats, and querying capabilities through SPARQL and RDQL (Rasqal RDF Query Library). Since these libraries are written in C

there is a binding's package (Redland Language Bindings) for other programming languages such as C#. However, during the framework's evaluation there were several problematic issues and the simple test examples went wrong, therefore, and despite of the announced features' potential, the study of this framework was abandoned. This framework is available at http://librdf.org/.

- **Semantic Planet:** is a framework for parsing RDF composed by two components. The first one is Spiral, which constitutes the Semantic Planet RDF library and provides the foundation for RDF services that can be used by other software applications for RDF parsing and serialization. The second component is Carp (Convenient API for RDF Programming), which conceptually corresponds to a higher layer that assembled upon Spiral. Carp is designed to provide a simple API for programming with RDF without losing the power of the underlying model. This project suffers from inherent problems of being too recent (version 0.3) for example: less stability, less support, and inconsistent documentation (API reference) with the examples provided, thus a translation is required to adapt them to the API's thorough reformulation. It was extremely difficult to evaluate an inconsistent API by a trial and error approach. Another problematic issue related with this topic is that future versions can also present the same API inconsistency problems thus jeopardizing PIT-RSL's code maintenance. This framework is available at http://www.semanticplanet.com/.
- **Jena:** this framework provides all the required functionality, and solves all the earlier pointed out potentially problematic issues detected in the previous tools' analysis. However, this tool is not written in C# thus requiring its previous conversion into a .NET DLL with IKVM.NET. For further research or to gain a more detailed description of this framework, for consolidating these choice rationales, the reader may consult List of Software Applications. This framework is available at http://jena.sourceforge.net/.

2.6 Related Research Projects

The research phase culminated with the in-depth analysis of the following three major projects, from which CIRCE deserved special attention, having been exhaustively studied

with the objective of extracting concepts and techniques for implementation purposes, applied in the functional prototype, presented further.

2.6.1 CIRCE

It is a research project started in the middle of the 90's in an Italian university. Their main goal is to conceive a tool that provides a specialized cooperative and interactive environment for requirements specification, based on client-server architecture. This tool supports the tasks not only from requirements engineers, but also from non-technical users, such as project owner and high management board members, which are very important stakeholders, crucial to the success of an IT project. This goal is achieved by using Natural Language (NL) as the specification language and providing feedback with a multiple-views approach [Ambriola2000a]. CIRCE's main functionalities allow:

- **Knowledge extraction:** achieved by using fuzzy matching domain-based parsing techniques implemented in CICO.
- **Views and models synthesis:** generation of flow diagrams and behavior.
- **Models validation:** intra- and inter-model consistency validation.
- **Quality and cost measures' analysis:** evaluation measures associated with the document, system, and process [Ambriola2000b].

Nowadays, since 2003, CIRCE's authors have embraced the idea of also developing an Eclipse plug-in to allow CIRCE's integration with other software development tools, beneficiating from the flexible plug-in architecture provided by the Eclipse platform. Their purposes are similar to PIT-RSL's main goals.

CIRCE's main advantages are related to the fact that instead of treating NL requirements as black boxes, similar to the approach adopted by the mainstream commercial requirements management tools, it tries to extract all the specific domain knowledge from NL requirements, taking into account their specificity [Ambriola2003].

The proposed workflow is based in an interactive "write-evaluate-feedback" cycle as presented in Figure 2.2. In this process, initially the user introduces NL requirements and specifies the glossary entries (designations). Then it submits the NL information to CICO that

parses it and generates an intermediate representation similar to a project's specific knowledge base.

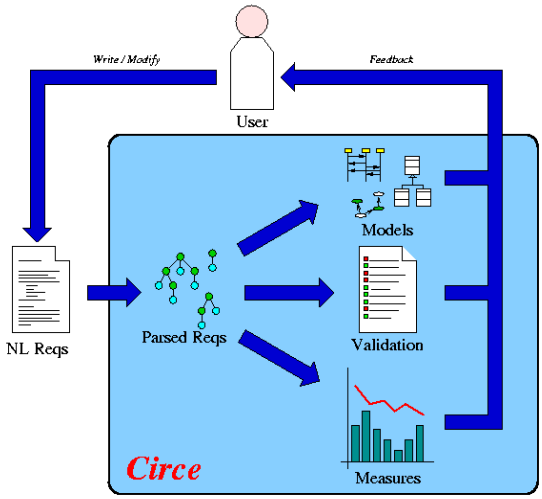


Figure 2.2 – CIRCE's Interactive-Based Architecture (extracted from [CIRCE2003]).

After parsing the NL requirements, the generated abstract intermediate data representation can be manipulated in two phases by projectors and translators [Ambriola2003], as illustrated in Figure 2.3, to be presented back to the users, providing this way an important feedback of the work performed (system specification). These two information manipulation entities can be combined together generating higher-level functionality by composite architecture, enhancing the tool with a robust and flexible extension mechanism.

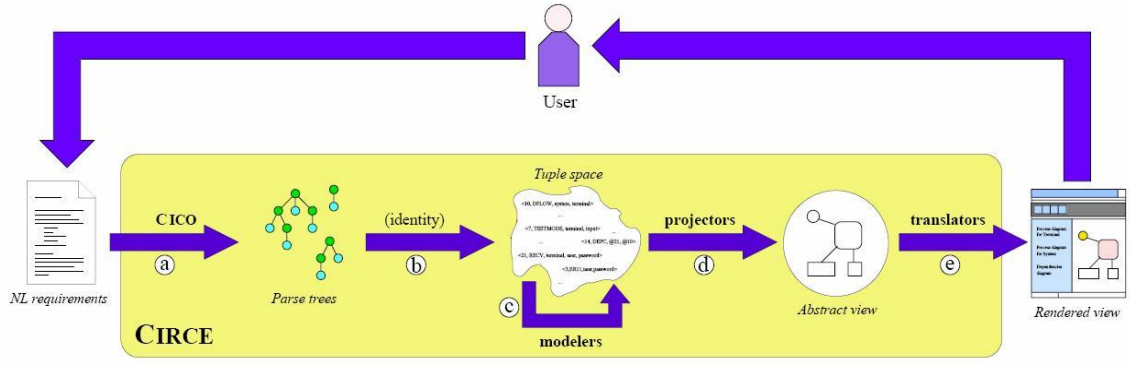


Figure 2.3 – CIRCE's Process Workflow (extracted from [CIRCE2003])

2.6.1.1 CICO

It's essential to emphasize the importance of the role that CICO plays in this project, since it assumes a crucial position in CIRCE's architecture and, consequently, in its overall functionality. CICO encompasses all the parsing NLP techniques and helper tools that parse, analyze and extract the knowledge form requirements written in NL free-form text.

CICO is a shallow domain-based parser that uses innovative techniques such as fuzzy matching rewriting, backtracking, and heuristic-based optimization strategies to find efficiently the optimal parse tree among all the possibilities in the search space [Ambriola2003].

The shallow parser designation derives from the fact that it takes advantage of the semantic adherence to the syntactic structure of the requirements sentences. This requirements' characteristic is particularly true if we consider that most of the times requirements are short declarative sentences. The whole CICO's internal mechanisms and functionality is based on this simplification assumption. Besides this central premise, there are also other minor restrictions that have to be verified to assure the correct operation of this NLP component. All of these peculiarities have to be taken in consideration when analyzing the parsing mechanisms that CICO's algorithms employ [Ambriola2003] to determine their range of applicability, namely according to the type and length of statements.

The fuzzy matching parser presents a recursive and heuristic-based algorithm, having in consideration several potential problematic issues to optimize performance and to guarantee a time-useful responsiveness [Ambriola2003].

The NLP technique is based in three types of input files:

- The free-form text file that contains the requirements introduced by the users.
- The glossary entries file (designations) that force the users to reflect over the specified domain entities to be used in the requirements text file. It contains for each domain entity a base term, a tag description list, and base term synonyms. To guaranty consistency, all occurrences of the synonym terms are replaced by the base terms followed by the respective tag list.

- The MAS rules file that contains the specification language definitions. Each rule is composed by three components: (1) the model part, which specifies the template used during the match; (2) the action part, which is evaluated when the rule is fired; and (3) the substitution part, which replaces the statement fragment that matched with the model part.

CICO main functionalities are:

- Typographical and format normalization.
- Glossary-based substitution.
- Fuzzy matching parsing based on MAS rules language definition.
- Heuristic selection and semantic validation.

CICO presents several advantages when compared with traditional parsing techniques as the ones used for programming languages compilers generation, because it presents an innovative, flexible, and robust parsing approach. This CIRCE's component, CICO, was a major reference to this work. However, it presents a major drawback when compared with other approaches since it requires a model parameterization for the correct operation of the heuristic-based algorithm employed.

2.6.2 Attempto

This project is also underdevelopment since the middle of 90's in a Swiss university. The main goal of this research project is to develop a controlled English specification language, called ACE. It defines a correct Standard English subset with a specific vocabulary. The strict grammar enforces the reduction of inherent NL ambiguity and assures that the specified information is computer-understandable [Fuchs1998].

The parsing mechanism consists in the translation of the strict (grammar verified) natural language specification provided as input into a first order logic variant called discourse representation structures (DRSs) [Fuchs1998].

The main goal is providing to the user the best from both worlds: the NL **familiarity** and the **rigor** of formal languages.

The offered functionalities are: strict specification formalism, specifications' composition, querying, and execution.

The requirements translation and analysis process encompasses the following steps:

- Requirements text translation into ACE discourse representation structures (DRSs).
- After the translation into DRSs, the knowledge-base and the inference-engine allow to extract explicit and implicit information by using queries.
- It's also possible to convert the DRSs into other data representations. This can be attained automatically into First Order Logic (FOL), causal logic and Prolog clauses, or manually. The latter can be useful for further employment in a generic theorem proving environment.

The described process is presented schematically in Figure 2.4. In the two upper left boxes, the paraphrase results obtained after the parsing stage are presented. Attempto includes in the paraphrased requirements all the implicit information interpreted from the original statement. In the right box the reader can see the result from the translation process into the discourse representation structure (DRS), emphasizing the resemblance with FOL, since DRS is a variant of the latter. The reader can also observe a small example from the querying and execution possibilities.

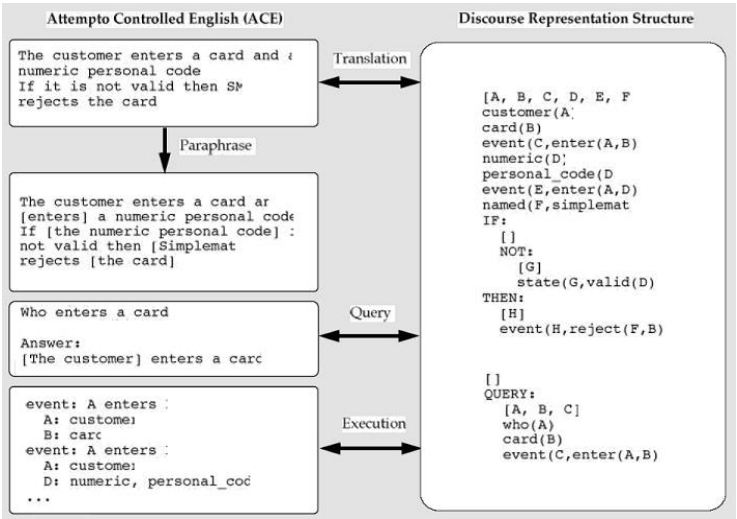


Figure 2.4 – Attempto Requirments Analysis Example (extracted from [Schwitter1998])

Attempto main advantages [Fuchs1998] are:

- It is not based on a parameterized heuristic-based parsing algorithm like CIRCE. This characteristic provides a desirable model independence (there is no need of performing previously a model parameterization) and offers greater efficiency because of its strict and deterministic grammar.
- Complete elimination of NL undesirable characteristics such as ambiguity, imprecision and inconsistency.
- Use of CICO's similar approach that exploits the semantic adherence to the syntactical structure of requirements statements, which simplifies the semantic analysis.
- The formalism features allows the specification's translation into other notations and further execution in logic programming languages environments such as Prolog.

However, this project also has certain important disadvantages, such as:

- The ACE specification language, as other formalisms, require a learning process that, although reduced when compared with other more complex formal languages learning processes, can difficult its adoption by domain experts.
- The ACE specification forces the user to adapt to the predefined syntactic structures. If the interpreted information do not correspond to user's initial intentions (requirement's implicit semantic), the user has to rewrite the requirements in other syntactic form until the ACE specification language produces the expected interpretation analysis [Fuchs1998].
- Although it offers the possibility of further specification execution, the provided method is neither portable, nor compatible with today's mainstream technologies solutions such as .NET framework, one of the technical requirements of PIT-RSL. Attempto takes advantage from the specificity of a family of programming languages, classified as logic programming languages in the literature. The exploited paradigm belongs traditionally to the Artificial Intelligence (AI) research area.

Besides the disadvantages mentioned above, from the insight gained during this initial research phase, it was clear that the complexity (an overkill when considering the current state

of the art in NLP techniques) of this approach was far beyond the scope of this graduation thesis, since the effort involved (theoretical and implementation) would certainly exceed by far the available time-frame to conclude this work, resulting in the end in a pour quality software tool when compared with the initial project's requirements.

2.6.3 NL-OOPS

NL-OOPS, like Attempto, is also a CIRCE's contemporary project. It results from a synergy effort between an Italian and a British University. NL-OOPS project main goal is to provide a CASE tool for object-oriented requirements analysis by using NLP, thus unifying the knowledge between the non-technical stakeholder and the requirements engineer. It's extremely useful when offering other tools' support in early Software Engineer's phases. NL-OOPS allows performing requirements specification and validation. Despite the promising features of this project it will just be superficially analyzed since there is less information available (e.g. white papers) of this project because it has become a commercial tool – that was the information provided by one of the project's co-author (Luisa Mich) via e-mail.

NL-OOPS emerges as a domain independent system using NL without any kind of restrictions [Mich2002]. Therefore, due to the NL requirements' high complexity, it requires a robust parsing system capable of addressing a series of potential problematic issues inherited from NL characteristics such as ambiguity, contradictions, omissions and redundancy. All this flexible set of features can only be achieved by a generic NLP parsing and inference engine, specially tailored to this effect.

2.6.3.1 LOLITA

LOLITA is the NL-OOPS parser and has a role similar to CICO in the CIRCE project, since their main goals are almost identical. It implements the previously mentioned generic NLP system by using optimized C parsing algorithms (e.g. Tomita algorithm) and Haskell programming language, since the latter provides several advantages for text manipulation, simplifying the inference engine logic layer implementation [Mich2002].

The parsing mechanism also performs the required pre-processing stages such as typographic, morphologic and normalization transformations. After the parsing stage, the extracted information is stored in a semantic web, called SemNet, which is basically a concepts'

navigable hyper-graph. At this point the LOLITA system is capable of generating object-oriented domain models with different detail levels through the application of algorithms that manipulate the information available in SemNet by applying translation and filtering operations.

A feature that is important to emphasize in this project is the highly flexible traceability system, which allows establishing links and navigating between artifacts through them at three distinct levels [Mich2002]: the textual requirements, internal structures for information representation, and the generated domain models.

LOLITA's core architecture is diagrammatically presented in Figure 2.5, where it is possible to observe the inter-component connections, the several NL parsing stages workflow, and their communication with SemNet. LOLITA also allows reports generation with the NL Generator component, and supports a complex template mechanism with the possibility of inter-template reference called hyper-templates [Mich2002].

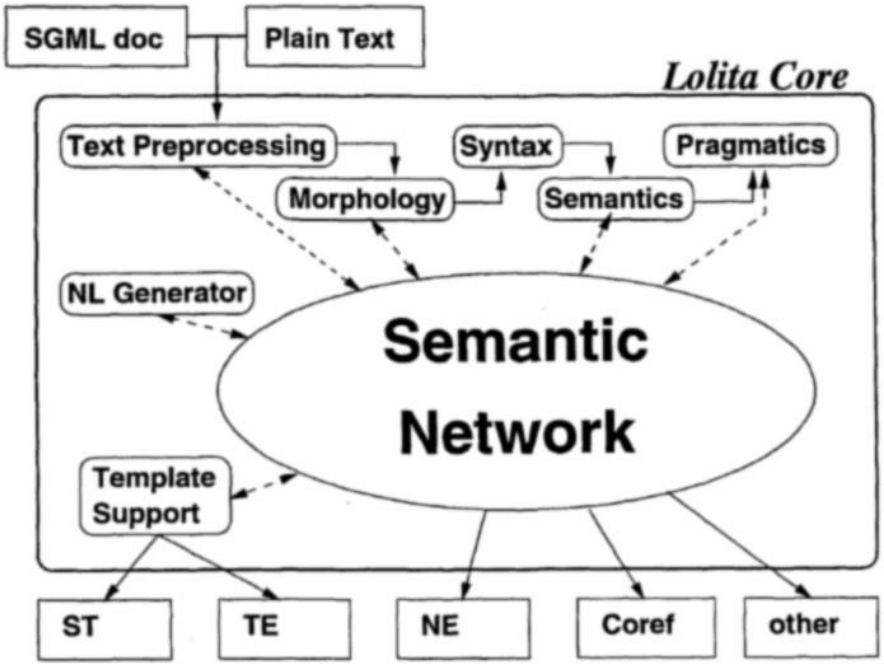


Figure 2.5 – NL-OOPS Framework's Core Architecture (extracted from [Mich2002])

2.7 Software and Requirements Reuse

This section will present some of the interesting results achieved by Jacob Cybulski in his PhD Thesis, that focuses essentially on software and requirements reuse [Cibulski2001]. However, it does not pretend to be exhaustive, presenting only some synthetic information:

- **Artifacts' attributes that facilitate their reuse:** expressivity, precision, formality, adaptability, computable representation, auto-contention, programming language independence, expressive power, and modifiability.
- **Reuse over the entire development life-cycle:** the artifacts should be reused over the entire development life-cycle span, with special focus on the upstream stage of the methodologies' processes in early activities such as requirements elicitation and design. The early higher-level artifacts reuse potential benefits are achieved by the automatic reusing of the subsequent artifacts. However, this type of reuse rarely occurs nowadays.
- **Reuse process activities:** this process involves three major activities, presented in Figure 2.6. The first one corresponds to the existent product **analysis** for searching reusable components. The second engages the reusable artifacts library **organization**. The third employs the new product's **synthesis** based on the reusable components.
- **Reuse processes' models:** there are two types of reuse process models. One is called development-for-reuse and the other is development-by-reuse. The former's main objective is the construction of a software reusable library, focusing in the **analysis** tasks, and then the subsequent *classification* and *storage* tasks presented in the **organization** activity for artifact's later reuse – the reader may consult Figure 2.6 for a detailed tasks enumeration. The latter's principal goal is to allow a simplified and efficient effective reuse of the libraries' previously catalogued components, during a new software product development process. It entails the rest from the **organization**'s tasks such as *searching* and *retrieval*, and also all the tasks presented in the **synthesis** reuse process activity.
- **Reusable software library management:** this process, which includes artifact classification, storage, search and retrieval, is one of the most fundamental services expected from any reuse environment. There are specialized technologies such as database and dictionaries that support this process. However, they are not entirely suitable for

handling poorly structured and informal artifacts, for instance free-form NL requirements. For these specific textual artifacts the best suited solution is a hybrid approach that utilizes other technologies, namely Information Retrieval (IR) techniques. The implied processes implemented in the majority of IR systems and their relations to the three major activities in software reuse (Figure 2.6), are schematically presented in Figure 2.7.

Cybulski's work is truly interesting, but it is also extensive. There are several other appealing issues focused on his PhD Thesis, however they are far beyond this report's objectives, so they will no be exhaustively analyzed here.

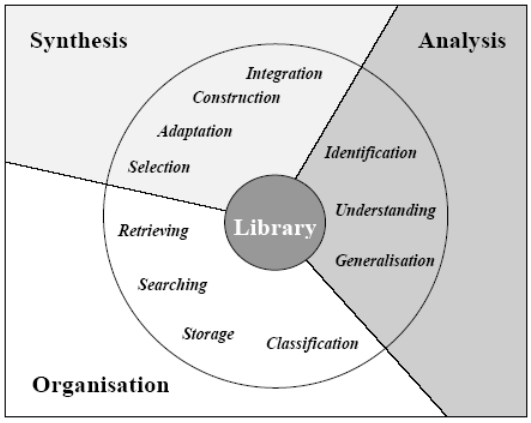


Figure 2.6 – Reuse Process Phases

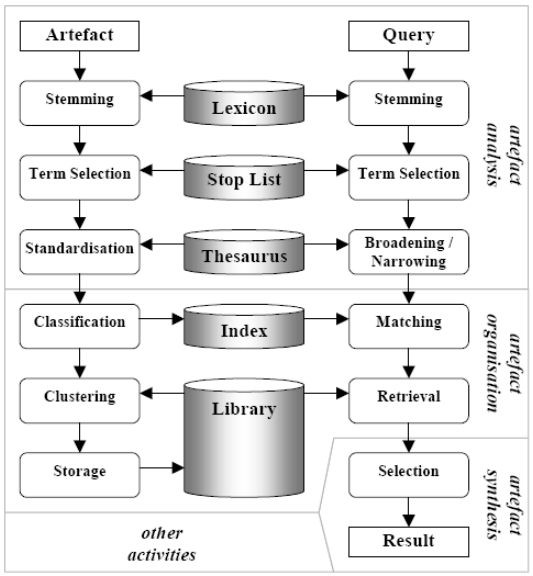


Figure 2.7 – Classification and Retrieval Process

3 PROJECTIT INITIATIVE

3.1 ProjectIT Background

The concretization of the ProjectIT initiative is a homonymous project, developed in the scope of an Information Systems Group (GSI) of INESC-ID research program, which results from the experience gathered from the daily projects in which it is involved. The outcomes of this initiative are supported by two prototypes: *ProjectIT-Studio* and *ProjectIT-Enterprise*. The idea is to study and research new approaches for requirements engineering, analysis and design activities as well as project management and code generation issues. The project's architectural components are presented in Figure 3.1. For further insight of these components the reader may consult [Silva2004].

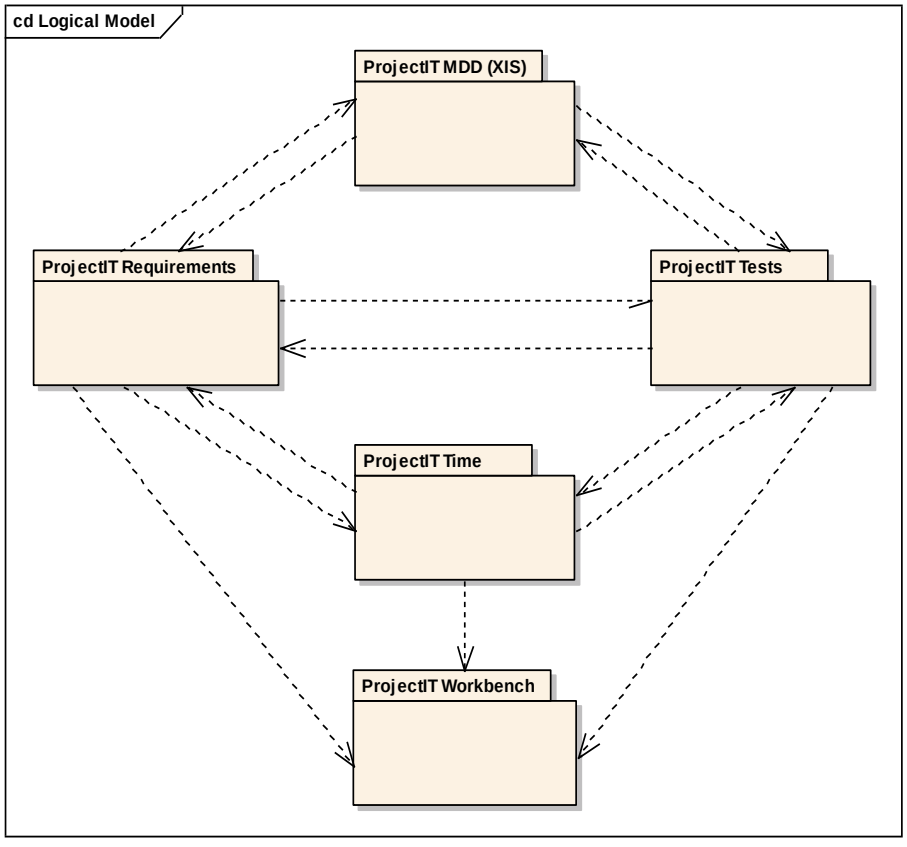


Figure 3.1 – ProjectIT's functional architectural components (adapted from [Silva2004]).

3.1.1 ProjectIT-Requirements

ProjectIT-Requirements, the focus of this graduation thesis, is the component responsible for requirements engineering issues, and its main goals are: (1) to define a language for requirements specification and documentation which, by raising the specification rigor, facilitates the reuse, traceability and integration with model-driven development environments; (2) to implement a supporting CASE tool for on-the-fly syntactic and semantic analysis and validation of the produced specifications, based on the previously mentioned requirements specification language. The ProjectIT-Requirements component's architecture is one of the main topics of this work and, consequently, it is presented and deeply discussed further in this document. From the previous description, it becomes clear that this component's goals belong to the scope of Requirements Engineering, being the activities involved crucial to the system's success, since through them we try to obtain a rigorous description of what the system should do, i.e., its behavior. Therefore, the greatest challenge of the ProjectIT-Requirements component is to minimize the undesired characteristics of natural language, namely its inconsistency, inadequacy, ambiguity, and incompleteness. The supporting vision of this component was, from the beginning, to build "a tool for writing requirements documents, like a word processor, and as we write, it will warn us of errors that violate the requirements language rules".

3.1.2 ProjectIT-MDD

ProjectIT-MDD is the component related to the area of information systems modeling and model-driven development. This component allows the specification of models, transformations between models defined in different languages, and the automatic generation of artifacts (such as source-code and documentation). Figure 3.2 presents a high-level overview package diagram of this component's architecture.

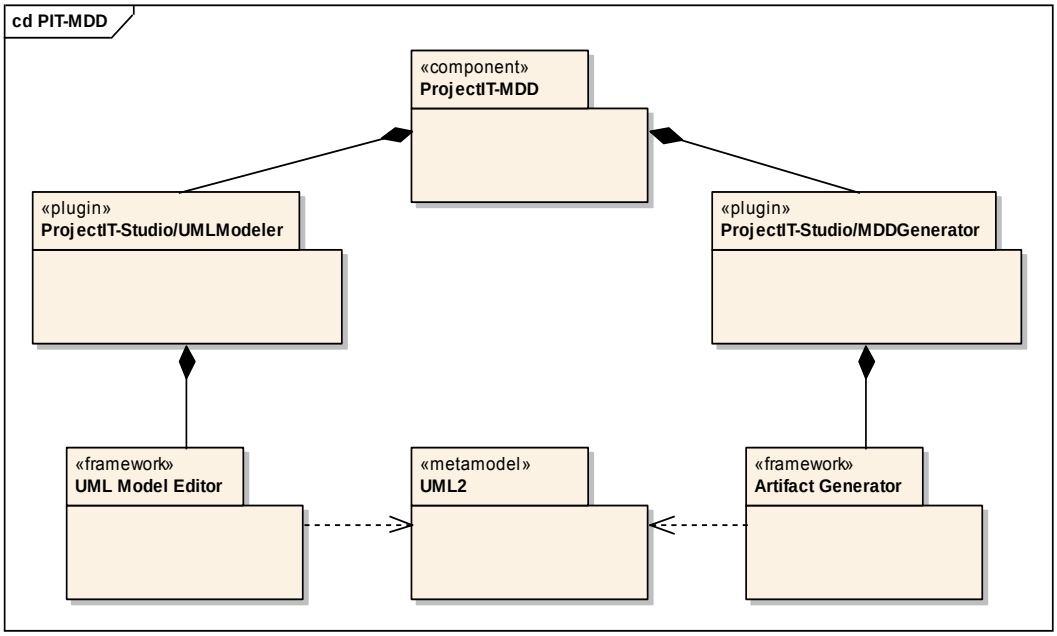


Figure 3.2 – ProjectIT-MDD's architecture (extracted from [Silva2006]).

3.2 ProjectIT Approach

The ProjectIT approach involves several tasks performed by different roles as presented in Figure 3.3, in two main moments during the software development process: (1) the definition tasks (usually executed by the Architect) for creating the artifacts that are the core of the ProjectIT approach, which are later used in specific projects; and (2) the use of these artifacts to develop specific projects, involving the other roles (Requirements Engineer, Designer, Programmer, Tester, and Integrator).

From a macro analysis point of view, we can say that the application of the ProjectIT approach receives system requirements from different stakeholders as its main input, and produces a set of artifacts as its output. Among all tasks involved, the ones performed by the software architect are crucial to the operation of the ProjectIT approach. Relatively to this work, the tasks that are assigned to Architect that worth to be mentioned are: (1) to define/adapt the linguistic terms and the syntactic and semantic rules of ProjectIT-RSL which can be defined on a project basis (thus providing a true support for a Domain Specific Language [DSMForum] [MartinFowler]); and (2) to define a suitable and easy-to-use UML

profile (when dealing with interactive systems the XIS2 profile could be used; nevertheless, the ProjectIT approach is profile-independent, which means that different UML profiles can be used).

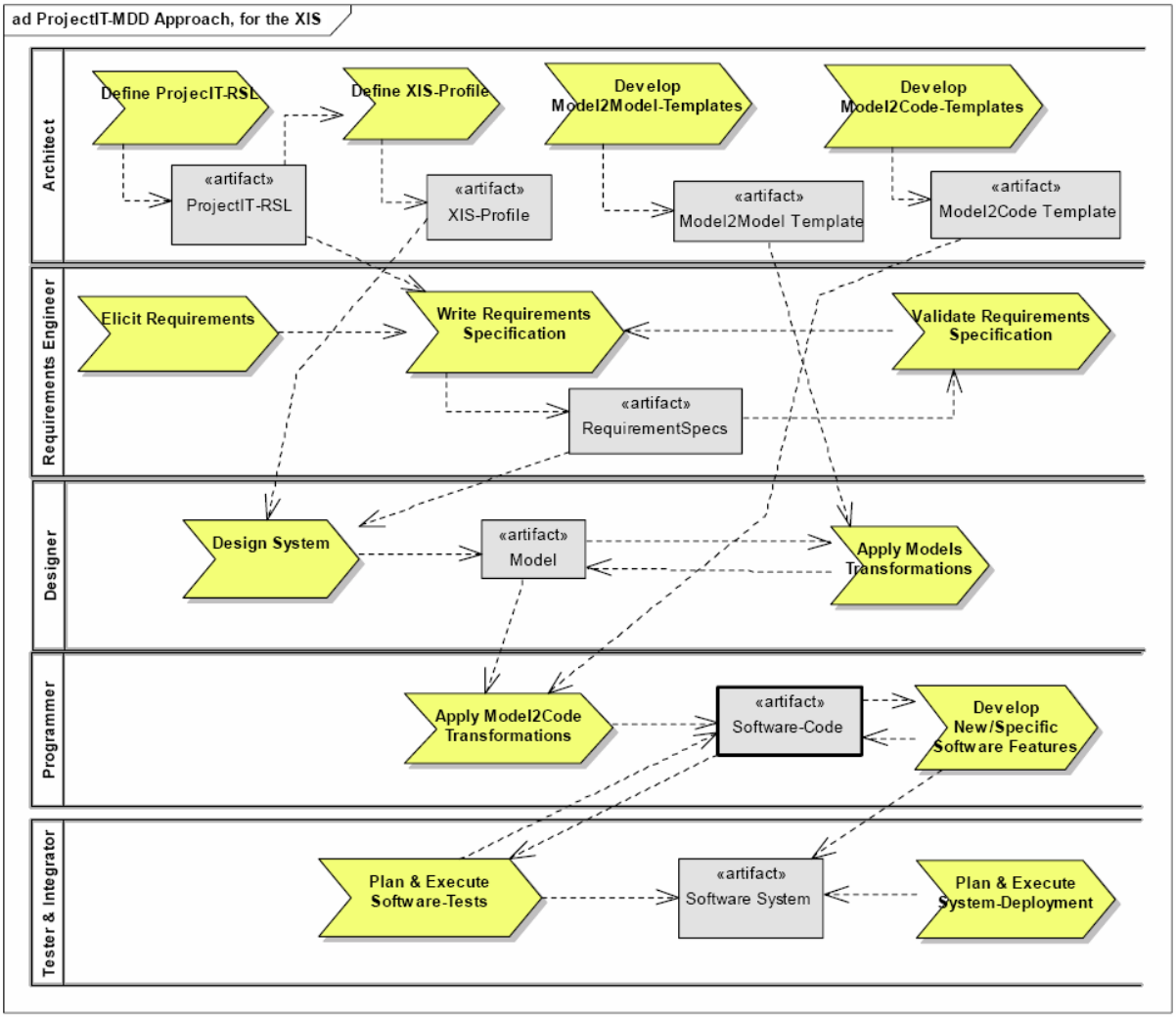


Figure 3.3 – Roles and Tasks Involved in ProjectIT Approach.

Focusing now on the scope of this project, the other role that deserves special attention is the Requirements Engineer, which in the scope of the ProjectIT approach (like in other approaches and methodologies) is responsible for: (1) gathering different system requirements using well-known requirements elicitation techniques (such as meetings, interviews, JAD sessions among designers, clients, end-users and other stakeholders); and (2) specifying the captured requirements with ProjectIT-RSL for further validation by using a set of parsing,

knowledge extraction and inference tools belonging to the ProjectIT-Requirements component, which the ultimate goal is to allow a non-technical user to specify the system requirements by his own.

From this point further, the validated requirements are then passed on to the Designer, whose responsibility is to produce an integrated set of models (the "Design System" task).

3.3 ProjectIT's Chronology

When analyzing this project's chronology, the key milestones worth mentioning are: (1) in 2004, the initial definition of the ProjectIT initiative and the ProjectIT-Requirements project were designed, which comprises the ProjectIT-RSL language; (2) in 2005, the first prototype of ProjectIT-Requirements was developed, based on the Microsoft's Visual Studio 2003 extensibility mechanisms (VSIP APIs) and, simultaneously, the GSI team migrated the well known Eclipse framework to the .NET framework by using IKVM.NET and JLCA tools [Saraiva2005b]. This port, suggestively called Eclipse.NET, was crucial once it constitutes the common platform for integrating all other ProjectIT tools, through the usage of its plug-in based architecture; finally, (3) in 2006, this work began, consisting in the development of the ProjectIT-Studio/Requirements component, which is divided in a Eclipse.NET plug-in and a package of parsing mechanisms, which corresponds to the work detailed in this document.

3.4 ProjectIT-RSL Previous Functional Prototype

When this project was initially proposed there was already a functional prototype available, which was developed by another student in a different evaluation context [Carmo2005]. This prototype served as a base for gaining a better understanding of the PIT-RSL language applicability and inherent limitations. This prototype was based on Visual Studio .NET and its extensibility mechanisms. However, since ProjectIT-Requirements project and its requirements specification language (PIT-RSL) belong to a broader initiative, the need of a proper tools integration through a common software development workbench become notoriously imminent: the different components were able to communicate, but the integration was insipiently supported by the usage of a common XML file that was manually exchanged between the tools along the entire process, which was clearly not the most adequate solution. As mentioned before, the tool's integrations problem was solved with the adoption of

Eclipse.NET platform (conversion of the open source Eclipse platform, written in Java, to .NET Framework), which provides a very extensible and flexible plug-in-based architecture capable of streamlining the communication and data exchange process between the tools, thus fulfilling the required functionality for ProjectIT-Studio (the set of tools created to support the whole software product life-cycle). Therefore, with the adoption of this common software development workbench it was necessary to convert all the ProjectIT initiative tools to this new framework's architecture, to achieve a greater productivity level by eliminating error-prone and time-consuming manual integration tasks. This functional prototype was one of those tools that needed to be converted.

The previous work [Carmo2005] comprehended a meta-model definition and a context-free grammar that recognized it. The platform used was based on extensibility mechanisms of Visual Studio .NET 2003, namely Visual Studio Industry Partner Program SDK (VSIP) and an integrated high-level library provided, named Babel.

The functional prototype provided some components, specifically a text editor and a new type of project based on VS .NET 2003, and a compiler for the grammar implemented language. However, there were no debugging features implemented. Besides VS .NET 2003, this project required Flex and Bison as parser tools to create the LALR(1) parser used by the compiler.

The major functionalities provided were: on-the-fly validation based on the grammar, syntax highlighting, and auto-completion. The result was a XML file in the XIS format [Silva2003a] for using as input in ProjectIT-MDD, with the purpose of automatic code generation.

Despite the hard work involved to achieve these results, this prototype suffered from some restrictions. After its examination, it was possible to recognize some of the problems and limitations with the adopted approach, which was based on the mechanism of a typical programming language compiler. After analyzing other research projects, such as CIRCE, it was clear that this approach was limited by nature and that it would jeopardize some of the ProjectIT-Requirements goals in the long term, related to the characteristics needed for interpretation and analysis of natural language requirements.

Therefore, PIT-RSL urged for the adoption of a disruptive approach in relation to the previously developed work, since it does not provide the flexibility and robustness that a

shallow domain-based parser like the one used by CIRCE (CICO) offers. This way, ProjectIT-Requirements will be endorsed with a parsing engine that allows it to easily extend the natural language linguistic patterns supported by ProjectIT-RSL and, therefore, the language itself according to the specific problem domain at hands for a particular project (in conformance with the DSL concept [DSMForum] [MartinFowler]).

In conclusion, despite sharing the same goals, this functional prototype had some drawbacks. One of those problems that was early detected with the first version of the ProjectIT initiative was the integration difficulty between the various tools, already explained above. The other was the lack of flexibility to process natural language requirements specified with free-form text. Although it was a risk, by taking advantage of the necessity to build ProjectIT-Requirements from scratch to integrate it in the new common workbench (development of a Eclipse.NET plug-in), the initial work's specifications were modified towards a more flexible approach similar to CIRCE's project. Therefore, this new project tried to achieve a more adequate solution to the Natural Language Processing (NLP) underlying problem, with the cost of entering in this still immature field of Artificial Intelligence (AI) and loosing some support from the previously developed work.

3.5 Eclipse.NET Project

The Eclipse.NET platform is a Java to .NET Framework migration of the well-know and widely adopted Eclipse platform. The free, open-source Eclipse platform had a broad acceptance in the Java community due to the complete Java-based software development workbench provided by the JDT plug-in and respective extensions, and by offering an excellent extensibility mechanism and a flexible and effective communication model between its extensions. To better understand the Eclipse's plug-in based architecture, and consequently the Eclipse.NET provided internal mechanism associated with its architecture, it's necessary to present the core concepts in which they are based on: (1) *plug-in*, represents the smallest unit of functionality that can be developed to the Eclipse platform, supporting the modular development and independent distribution concepts; (2) *extension point*, provides a plug-in's variability point that constitutes a practical and incremental way of extending or customizing portions of its functionality; and (3) *extension*, provides a way of contributing with extended functionality to an extension point (a sort of contract that obligates the extension plug-in to

implement it) declared by other plug-in. The greatest benefit of this architecture is that the plug-in being extended knows nothing about the plug-in that is connecting to it, beyond the scope of that extension point contract. This mechanism provides isolation to separated developed plug-ins allowing a seamlessly interaction between them.

The Eclipse.NET, as a result, mirrors all the same tools development and architectural advantages towards the .NET platform, consisting in an application through which the programmer can develop new applications by creating extensions, i.e., the Eclipse.NET is a software development framework. The underlying platform deals with the internal I/O details of the newly created application and provides an extensive set of abstractions and libraries of reusable utilitarian APIs, namely for Graphical User Interfaces (GUI).

Besides these attractive and useful technical features, the Eclipse platform has a considerable community of developers that support this project, by providing extensive documentation and plug-in examples available on-line, which can be adapted to the Eclipse.NET platform manually or by using the JLCA tool.

However, besides the major advantages of using the Eclipse.NET platform for developing other tools such as the ones belonging to the ProjectIT initiative context, this common base platform adopted has some problematic issues: (1) it has a considerable dimension and complexity which can be overwhelming for certain situations; as a consequence (2) it has an significant learning curve, since it involves a constant search of the platform's documentation to properly explore the semantics of the provided APIs; and (3) it requires a permanent conversion effort, even when analyzing simple examples, which can be minimized by using JLCA, although they involve time-consuming tasks.

3.6 ProjectIT-Studio Framework

ProjectIT-Studio [Silva2006] is the proof-of-concept tool of the ProjectIT initiative. It provides an integrated environment that supports the central tasks of the software development life-cycle, mainly: (1) requirements specification, (2) architecture definition, and (3) system design. One of the underlying goals of this tool is to promote productivity by providing a set of innovative features like natural language processing for formal requirements specification, models-to-code transformation techniques, template managing,

and UML profile definitions. ProjectIT-Studio tool is built on top of the Eclipse.NET platform [Saraiva2005b] (located at http://www.sourceforge.net/projects/eclipsedotnet), which provides an extensible plug-in based framework for developing other tools. Figure 3.4 presents the generic architecture of the ProjectIT-Studio environment, as an orchestration of a set of components developed on the top of the Eclipse.NET framework. Consequently, ProjectIT-Studio should be considered as an extensible, modular and plug-in-based environment.

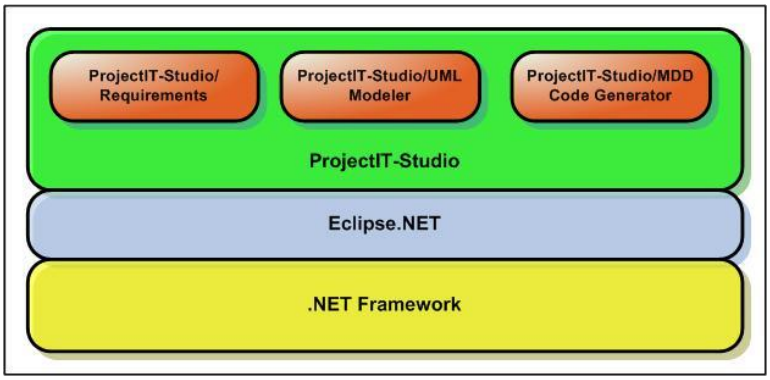


Figure 3.4 – ProjectIT-Studio's High-level Architecture

The ProjectIT-Studio/MDD component supports the modeling and transformation tasks by applying different techniques and mechanisms to accelerate and improve software engineering activities. These techniques are aligned with model transformation techniques, and in particular with the XIS UML profile [Silva2003], a set of coherent UML extensions allowing a high-level visual modeling approach to design interactive systems. This component aggregates two main plug-ins, the ProjectIT-Studio/UMLModeler and the ProjectIT-Studio/MDDGenerator, which are described below.

The ProjectIT-Studio/UMLModeler plug-in consists of a standard UML modeling tool that allows the designer to create visual models of the system using the UML 2.0 language [OMG2005a] or a given UML profile, e.g. the XIS UML profile. These models are used in ProjectIT-Studio/MDDGenerator to generate artifacts according to specific software architectures. Besides the creation of UML models, the tool also features a Profile definition mechanism, which allows the further customization of UML, adapting the modeling language to templates used by the subsequent artifact generation process.

On the other hand, ProjectIT-Studio/MDDGenerator is the plug-in responsible for the generation of system artifacts. Its input is a generative process, which is represented as a configuration file that specifies the details necessary to create the final application: (1) the name; (2) the model, an abstract representation of a software system; and (3) the software architecture, a generic representation of a software platform, based on templates (generic representations of software artifacts that support model-to-code transformations) for a target platform.

4 PROJECTIT-REQUIREMENTS ANALYSIS AND DESIGN

As already mentioned before, the purpose of ProjectIT-Requirements is to innovate by creating an approach-independent requirement's specification language. At the moment, for development purposes it is oriented towards the XIS2 UML profile. The main goal is to allow this specification language to be as looser as possible, resembling Natural Language, for minimizing new users' participation learning curve's sharpness (especially non-technical ones). The project also embraces the design and the implementation of a text editor with identical features of an IDE for this language, based on the plug-in architecture of Eclipse.NET, the base platform of ProjectIT-Studio. The specialized text editor's main goal is to support edition features, error reporting, validation, and consistency verification of the introduced requirements. Additionally, this project intends is to allow requirements reuse, providing coherent packages that the users can later import for other projects, allowing the concrete application of the concept developing-by-reuse[ATT2004].

4.1 Main Problems to Solve

The main problems to be solved can coarsely be classified in the following major groups:

- **Parsing:** implementation of all the necessary transformation steps needed for morphological and syntactical analysis, required for information extraction from free-form NL requirements text, including helper tools that perform pre-processing stages and the recursive algorithms necessary for navigating through the abstract syntax trees.
- **Semantics:** this set of issues encompasses the correct storage and information manipulation for extracting specific implicit information from the abstract syntax trees generated during requirements parsing. It's crucial to create the correct domain model and implement the most adequate data structures to avoid compromising non-functional requirements such as performance (typically penalized by the heavy recursive background computations incurred by this type of manipulation and backtracking algorithms).
- **Presentation:** this group entails aspects relative to the user interface and its usability. It's fundamental that PIT-RSL presents the user only the most relevant information for each feature provided. Only by this way it will be possible to achieve the goal of enhancing the

requirements' specification and validation process, thus streamlining the involved workflow.

- **Integration:** since PIT-RSL is a module of a greater and more ambitious project, ProjectIT-Studio, despite the possibility of working in standalone mode, it's more appropriate to use it in conjunction with ProjectIT-MDD tools (ProjectIT-Studio/UML Modeler and ProjectIT-Studio/Generator), thus covering the entire software product development life-cycle. Therefore, having in consideration the complexity of both projects, it's crucial to design and apply an exhaustive series of tests to ensure the quality of the integration performed in the later phase of this work, involving these two components implementations.

As the reader may have noticed there is a thin resemblance with the MVC architectural pattern: (1) the semantics group corresponds to the model part; (2) the presentation to the view; and finally (3) the parsing clearly matches with the controller part. This similarity with MVC architecture will be further exploited in Chapter 4. However, the stretched MVC's transposition does not include the integration topic because this group of problems is related with the plug-ins' interfaces binding (Eclipse.NET's extension-point mechanism) and semantic models transformations.

4.2 Project's High-Level Requirements

This section briefly describes the main functional requirements, grouped by actors, who roughly map to the three users' stereotypes identified, namely:

- **System Administrator:** this role is associated with the administrative tasks, as configuration of the access control mechanism based on user profiles, and with tasks that involve tool configuration, such as redefining or adding domain specific TS rules to the built-in libraries.
- **Requirements Engineer:** this role matches with authorization levels assigned to the requirements engineer, the one that can perform typical edition operations on project's specific domain Template Substitution (TS) rules (this core concept will only be meticulously explained further in the document). Besides these particular functionalities, this actor may perform all the tasks available for the next actor.

- **Non-technical Stakeholder:** this role corresponds to the typical user (domain expert) that simply writes down the system's requirements specifications, according to the expertise area of knowledge that he/she belongs. The activities associated with this role match up with the envisioned interactive cycle of tool's usage.

Eventually, in practice the roles from the first two actors can be collapsed in the same person, since it makes sense to think that the requirements engineer can perform most of the configuration and domain expertise tasks performed by architects in this tool.

4.2.1 Functional Requirements by Actor

The major functional requirements that ProjectIT-Studio/Requirements needs to support are:

4.2.1.1 System Administrator

- Can manage users' accounts.
- Can manage access control rights.
- Can create, modify, and delete individual built-in TS rules.
- Can include and exclude TS rules libraries.

4.2.1.2 Requirements Engineer

- Can create and edit RSL projects.
- Can import and export requirements packages.
- Can generate status reports and export them to typical office tools (Microsoft Word).
- Can edit the TS rules specific for a project and validate them.
- Can edit requirements and validate them.
- Can access advanced feedback views, which display important metrics for a RSL's project, covering aspects related with the quality and evolution of the document structure, requirements writing process, and system's under development complexity, through time.
- Can generate RDF/OWL output for further analysis and validation in specific ontology design and manipulation IDE tools like Protégé.

- Can generate UML models in order to passing them to the PIT-MDD tools the requirements extracted information, for them to continue the software development process, assuring a deep integration of ProjectIT initiative tools.

4.2.1.3 Non-technical Stakeholder

- Can create and edit RSL projects, but cannot modify neither the predefined project's template configuration, nor the project's associated TS rules.
- Can edit requirements and validate them.
- Can only access to a simplified set of the available project's wide set of views (provided by the multi-view perspective approach), which allows the user to obtain the necessary and sufficient visual feedback, specifically aimed to the requirements that need correction and are under the user's responsibility.

4.2.2 Non-Functional Requirements

The major non-functional system's requirements (which specify criteria that can be used to judge the operation of a system, rather than its specific behavior) that ProjectIT-Studio/Requirements needs to support are the following:

4.2.2.1 Usability

The tool must provide an easy to use and intuitive graphical user interface (GUI) in order to achieve the ultimate goal for which it was built for: supporting the users during the specification and validation of the requirements documents written in ProjectIT-RSL. The GUI must also provide mechanisms that enhance its perceived efficiency and elegance. Moreover, the tool should respect the widely accepted Jakob Nielsen's usability heuristics [Nielsen2006].

4.2.2.2 Platform compatibility

The tool must be fully compliant with the underlying Eclipse.NET platform, exploring all the provided mechanisms, namely the ones offered via the plug-in-based architecture for communication with other ProjectIT components (also plug-ins).

4.2.2.3 Performance

The tool must implement mechanisms that ensure that the GUI remains responsive independently of other tasks running in parallel or events that may be raised during the user interaction with the tool.

4.2.2.4 Efficiency

The developed tool must present a good ratio of machine resources (memory and CPU) consumption for a given load (e.g. a requirements document with a considerable dimension); to be precise a heavy load should neither lag the system nor trash it.

4.2.2.5 Resource constraints

To be widely adopted, the tool should not present serious restrictions on the hardware required to run it (processor speed, memory, disk space, network bandwidth).

4.2.2.6 Quality

The developed tool must present a low rate of faults delivered and a report of the major faults discovered.

4.2.2.7 Documentation

It is necessary to provide adequate documentation (design and architectural rationales, and also the user and technical manuals of the software product developed) to support a deep understanding of the system's behavior and to provide a quick insight for maintenance purposes.

4.2.2.8 Maintainability

Besides the documentation, the developed source code developed must follow the commonly instituted best practices for programming, such as using well defined architectural and design patterns and respect code format standards, to ease maintenance activities.

4.2.2.9 Legal and licensing issues

The developed work should avoid from the start the incorporation of third-party components that have underlying legal implications in their licenses, which can restrict an eventual commercialization of this software product. Moreover, it's mandatory that this work do not

use or include third-party components in a way that it violates any intellectual property authorship or copyright.

4.2.3 Use Cases

In this subsection, the reader will be presented with the use cases diagrams resulting from the previous functional requirements specification, mentioned before. The diagrams respect the previous established actors' categories order. The first one, Figure 4.1, presents the administrator's typical actions, and the last one, Figure 4.2, presents the several functionalities that ProjectIT-Studio/Requirements will provide to the rest of the users roles, namely to the requirements engineer and the non-technical stakeholder. As the reader can observe from these figures, there was the concern of explicitly presenting a "login" use case in Figure 4.1 because, although every user must first authenticate in the ProjectIT-Studio system before performing any type of operation to ensure that that traceability mechanism is correctly used, in this figure it assumes a extremely important role. This use case was created to emphasize the security issue associated to the Administrator role, namely its ability of managing all the parameters of the ProjectIT-Studio tool.

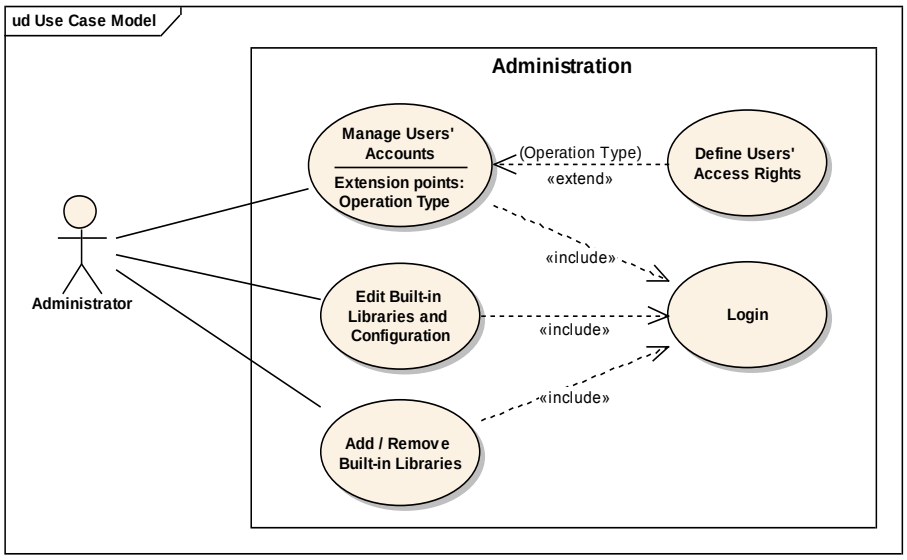


Figure 4.1 – Administrator Use Cases.

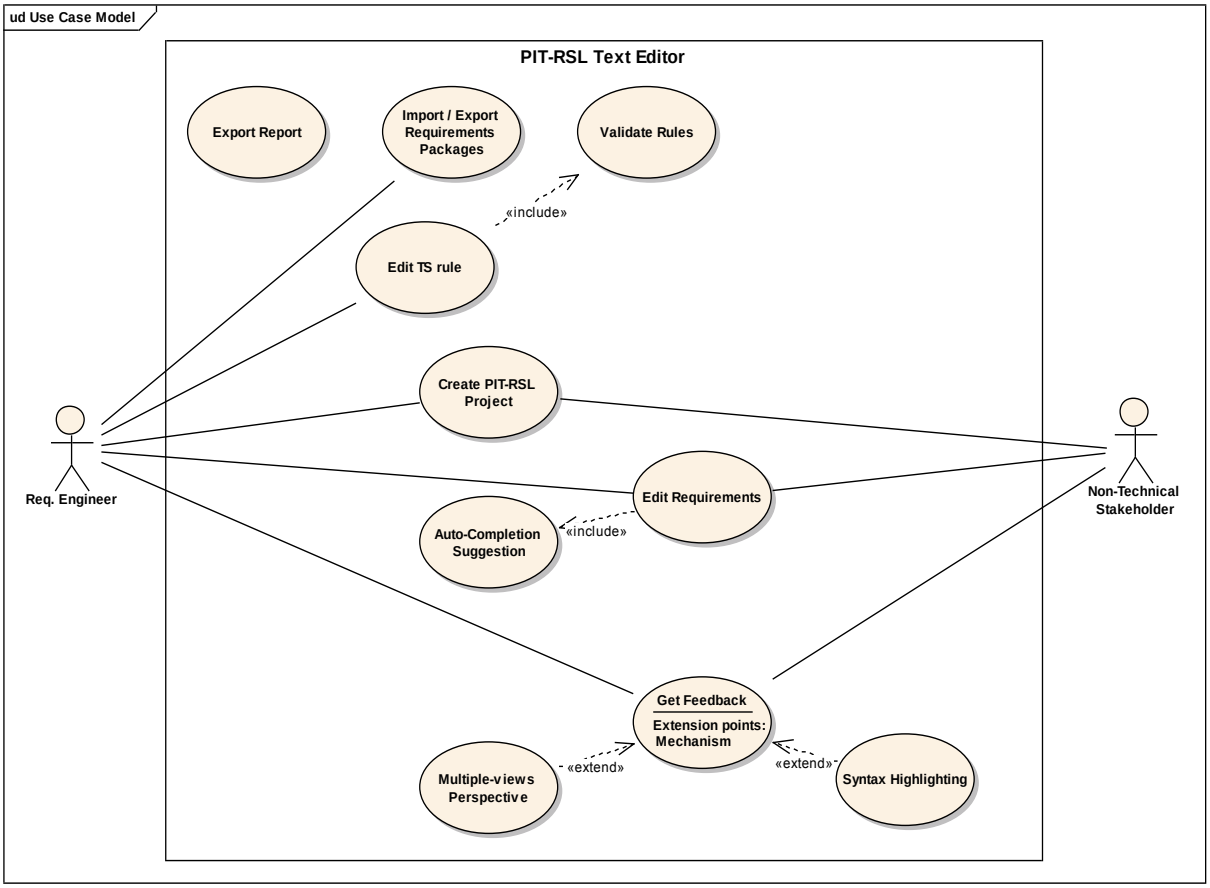


Figure 4.2 – Req. Engineer and Non-technical Stakeholder Use Cases.

4.3 Domain Model

ProjectIT-Studio/Requirements has a simple domain model (presented in next subsections), which is complemented by a large set of auxiliary data structures used by the parsers and error-checking pipelines. For a matter of simplicity these data structures' diagrams are not presented in this thesis report.

4.3.1 ProjectIT-RSL Metamodel

The definition of the ProjectIT-RSL Metamodel followed a simple and empirical approach. It is based on the analysis of several requirements documents of interactive systems, the major category of systems in which GSI of INESC-ID is involved in its daily projects, which provided a good insight of the typical structure and format of this type of documents and

allowed to capture the most common natural language linguistic patterns that expressed the main concepts of this kind of specifications: (1) *actors*, which are active resources that perform operations involving one or more entities; (2) *entities*, which are static resources affected by operations and have *properties* that represent and describe their internal state; and (3) *operations*, which are described by their respective workflows, consisting of a sequence of simpler operations that affect entities and their properties; in turn they are specialized in *actions*, which are atomic and primitive (and provided by default), and *activities*, which are not atomic. The elaborated metamodel of ProjectIT-RSL language is illustrated in Figure 4.3.

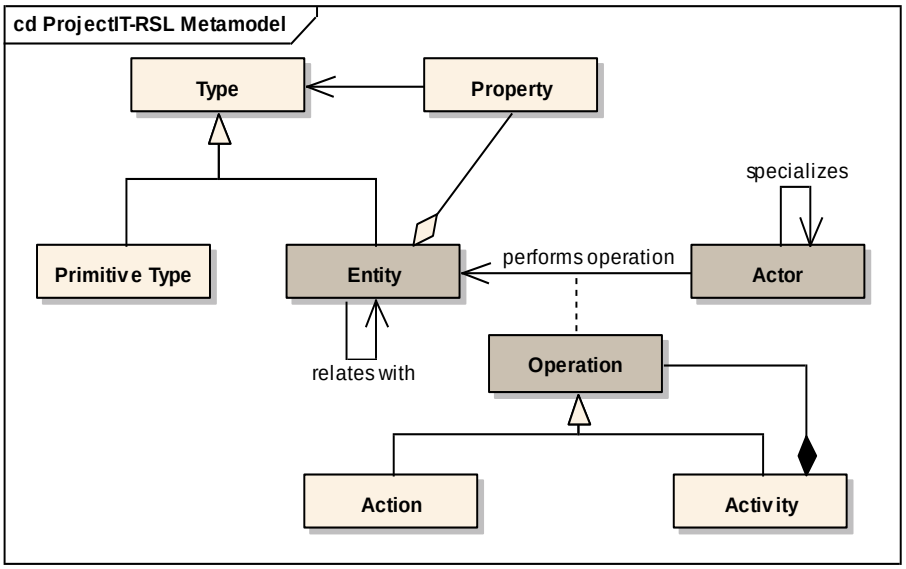


Figure 4.3 – ProjectIT-RSL's Metamodel

The core concepts of this metamodel, which are highlighted, reveal a typical natural language construct, designated as the statement construct, which allows to specify what the system should do through an operational perspective in which a subject (an actor) performs an operation (typically a verb) on an object (an entity), which affects the object by eventually modifying its internal state (entity's attributes change), i.e., it follows and imperative approach. This simple construct pattern can be enriched by using conditions that constraint this ternary relation. Besides this core construct the ProjectIT-RSL also supports declarative constructs to just name or enumerate concepts. This simple metamodel is the base for the profile common to all ProjectIT initiative tools.

4.3.2 ProjectIT-RSL Structure

One of the major contributions of this work to ProjectIT-RSL was to introduce the structural concepts of this specification language's documents. Besides the basic concepts previously presented, one can observe that all requirements information is written in documents that follow a specific format. The decision of defining a normalized and standard structure for this kind of documents reflects the best practice of adopting a common organization and format for requirements documents with the final goal of enhancing the reading and comprehension tasks performed by different stakeholders and also to simply the validation process. The PIT-RSL requirements documents structure model is depicted in Figure 4.4.

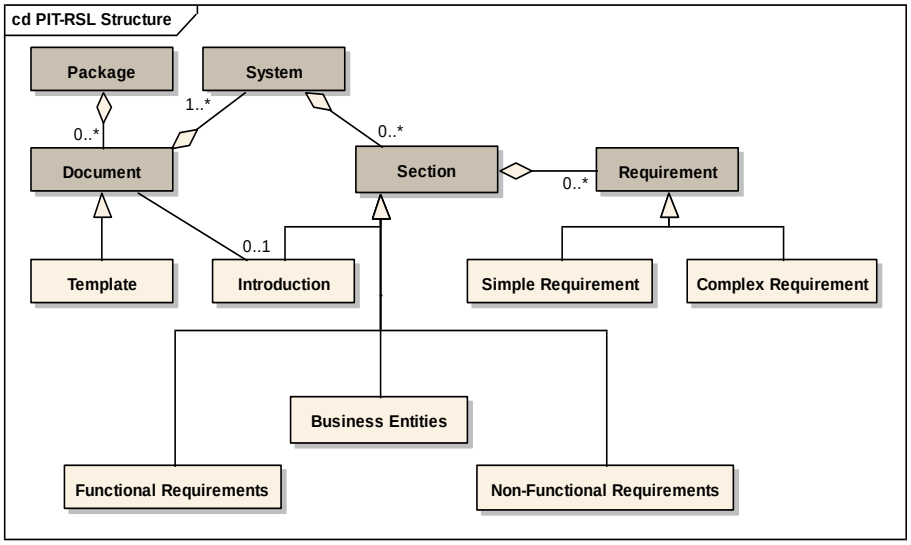


Figure 4.4 – ProjectIT-RSL's Structure Model

The broader concept of ProjectIT-RSL is the *package* concept, whose semantic is very similar to the programming languages concepts of package, i.e., it represents a set of related requirements documents grouping them in a single namespace. Practically, this concept represents the file system directory path of each document, since the directories hierarchies should reflect some kind of conceptual organization or classification of the information stored within it. Following the package concept, the *document* concept appears representing the physical digital artifact concept, signifying a consistent unit of specification information. The document can be of two different types: it can be (1) an *applicational template*, where an

instantiation of a template takes place to accelerate the requirements specification process by using the concept of develop-by-reuse [Cibulski2001] or it can be (2) an *architectural template*, where the main goal is to produce abstract documents for a predetermined type of system, emphasizing the concept of develop-for-reuse [Cibulski2001], whose the main objective is to build a large and robust library of architectural templates for later reuse, allowing to instantiate documents of type (1). Next, it was introduced the *system* concept that clearly maps to an executable component (either a complete application or a reusable software subcomponent) with which an actor interacts, by executing operations to access or manage entities and their properties. Then it was decided to divide the system concept into different specialized types of *sections* to provide a coherent organization of the document by grouping the requirements *sentences* specified, according to their goal and context. There are two main types of sections: the (1) *auxiliary sections*, which are used to append additional information to the document or to the system under specification, namely an introduction, comments, and imports sections; and the (2) *requirements specification sections*, such as business entities, functional requirements (that is further divided in other subsections), and non-functional requirements sections; that contain the requirements specifications of the expected software system under analysis. Finally, it was considered the most basic structural concept of the ProjectIT-RSL, the *requirement*, which can be a *complex requirement* or *simple requirement*, if it has or hasn't a nested items' context associated, respectively.

4.3.3 ProjectIT-RSL Sentence Construct

The sentence rules of ProjectIT-RSL can be of two distinct types: (1) *declaration*, if it just names or enumerates a list of concepts of the system under specification; (2) *definition*, if the sentence construct expresses detailed information about one of the declared system's concepts. If we analyze the possible sentence constructs under the spotlight of the three main ProjectIT-RSL language's concepts (actor, operation, and entity) to determine the underlying object-oriented computational concepts supported, one can observe that despite minor differences they are very alike. From a high-level point of view, one can observe from the EBNF notation of the supported natural language linguistic patterns recognized by PIT-RSL (the detailed EBNF description of valid PIT-RSL sentences is presented in Appendix H) that it can express the following relations between concepts: (1) *single inheritance*, which is

specified by using the "is a" linguistic pattern; (2) *equivalence*, through the usage of "same as" language construct; (3) *elements enumeration*, used to specify composition relations (composite design pattern [Gamma1995]) and to support element listing for which a predicate apply (e.g. list of entities that are manipulated by a given operation; process's workflow definition with a sequence of the operations involved); (4) *quantifiers*, to express the cardinality of the associations between entities. Beside these relations, it was also considered the *association* and *association class* concepts. To correctly capture these relations it was defined that it would be used the active and passive voice patterns to establish bidirectional relations between entities (e.g. from the following requirement statements "a supplier *provides* a list of products" and "each product is *provided by* a supplier" we can capture the association relation between the supplier and the product entities by observing the two verb forms; to establish an association class relation we can associate a third entity identified by the action expressed by the verb used in the sentence and then associating properties to it).

4.4 Application's Solution Sketch

As pointed out earlier, the identified problems can be grouped according to several areas for which the most suitable solution sketch is a variant of the common MVC architecture [Fowler2003], given the resemblance previously mentioned.

- **Parsing:** this part of the solution involves activities such as the definition and implementation of algorithms and suitable data structures for the constituent parsers. These helper parsers perform transformations such as formatting and normalization of free-form natural language text. After these initial parsing stages it is necessary to implement the core parser that actually performs the analysis and manipulation of the information presented in the free-form text requirements. This parser is responsible for populating the knowledge-base with the explicit and implicit information presented in the supplied requirements, under the form of domain-based parsing trees, one for each requirement.
- **Semantics:** following the architecture patterns presented by other projects with similar goals and specifications (the reader can consult Related Requirements Specification Language Projects available in Chapter 2), it will be adopted a semantic network as the

knowledge-base and an inference-engine that manipulates the information stored in the semantic network with the objective of providing the correct and specific information needed for each concrete operation that PIT-RSL will offer, most of them related with validation and presentation issues.

- **Presentation:** this solution's guidelines class is essentially related with the features to be provided by the ProjectIT-Studio/Requirements plug-in, namely its text editor component. It focuses on the creation and manipulation of a perspective-based multi-view approach, the syntax-highlighting and auto-complete mechanisms, and the contextualized tool-tips and suggestions needed for the requirement's validation and consistency check process.
- **Integration:** this part relates with tasks of integration with the rest of PIT-Studio, especially with ProjectIT-Studio/MDD tools, specifically with ProjectIT-Studio/UML Modeler and ProjectIT-Studio/Generator. This involves the conversion of OWL to UML models and the execution of unitary and integration tests needed to achieve the final degree of product's quality expected. Despite the fact that PIT-RSL can work as a standalone component, exporting the achieved results to other tools via the adoption of standard formats such as XMI and RDF/OWL, the greater benefit of this tool, as pointed out before, derives from the integration with the rest of the ProjectIT-Studio components, accomplishing an higher level of coverage of the entire software product development's life-cycle. Therefore, integration is essential to assure PIT-RSL will provide the correct and required information, in the adequate format, as the input for ProjectIT-Studio/MDD tools and whichever other components might need PIT-RSL's generated information.

5 PROJECTIT-STUDIO/REQUIREMENTS ARCHITECTURE

The ProjectIT-Studio/Requirements tool architecture, illustrated in Figure 5.1, contains several components that can be aggregated in three main packages: (1) *PIT-RSL Parsers*, the package that includes all the parsers needed to perform Natural Language Processing (NLP), using as input the requirements document's text; (2) *ProjectIT-Studio/Requirements Plug-in*, which includes all the functionality associated with the text editor plug-in developed for the Eclipse.NET platform; and (3) *Jena .NET port*, which contains the knowledge-base and inference-engine components required for performing overall semantic validation of the capture information through reasoning techniques.

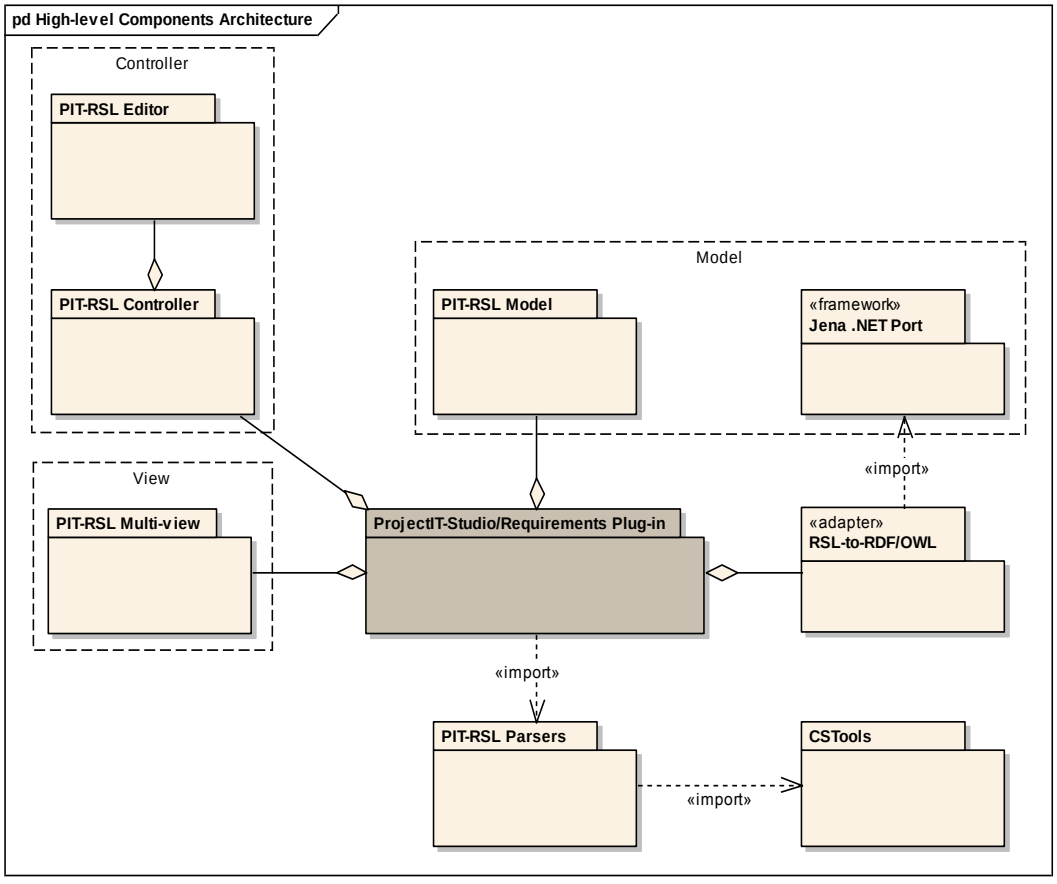


Figure 5.1 – ProjectIT-Studio/Requirements High-level Components Architecture.

The *PIT-RSL Plug-in* package is the root package of this tool, and it encapsulates three sub-packages, namely: (1) *PIT-RSL Model*, which contains the domain-specific nonvisual representation of the information (all the data and behavior other than that used for the UI) on which this tool operates, such as abstract types of information for efficiently manipulating strings, the requirements model and its requirements entries, parsing context tables, RDF/OWL oriented PIT-RSL metamodel, TS rules, etc; (2) *PIT-RSL Multi-View*, which contains a set of views that render the model into a form suitable for interaction, typically a set of user interface elements (representing views of the underlying model); and (3) *PIT-RSL Controller*, which contains the mechanisms and algorithms that handle any change to the information by processing requests or responding to events (typically user actions) and then invokes changes on the model, eventually updating all the dependent views. From this description of the architecture's diagram, it's clear the usage of Model-View-Controller (MVC) [Fowler2003] architectural pattern, in which the software system's data model, user interface, and control logic are separated into three distinct components. A key point in this separation is the direction of the dependencies, in which the presentation depends on the model but the model doesn't depend on the presentation. Therefore, programmers working in the model should be entirely unaware of what presentation is being used, which both simplifies their task and makes it easier to add new presentations later on, and additionally, it also means that presentation changes can be made freely without altering the model. In conclusion, through component's isolation, modifications to one component can be made with minimal impact to the others.

One can observe that software applications are usually broken into three main layers: domain, data access, and presentation (GUI). The main achievement with the MVC architectural pattern is to split the presentation layer into Controller and View roles. However, when building an application the most important separation is still the one between presentation and domain, the foundation of this architectural pattern.

Figure 5.2 depicts the typical relationship between the Model, View, and Controller roles. The solid lines indicate a direct association in which there is an explicit invocation, and the dashed line indicate an indirect association, in which the receiver is indirectly notified by using the OBSERVER (PUBLISH-SUBSCRIBE) design pattern [Gamma1995].

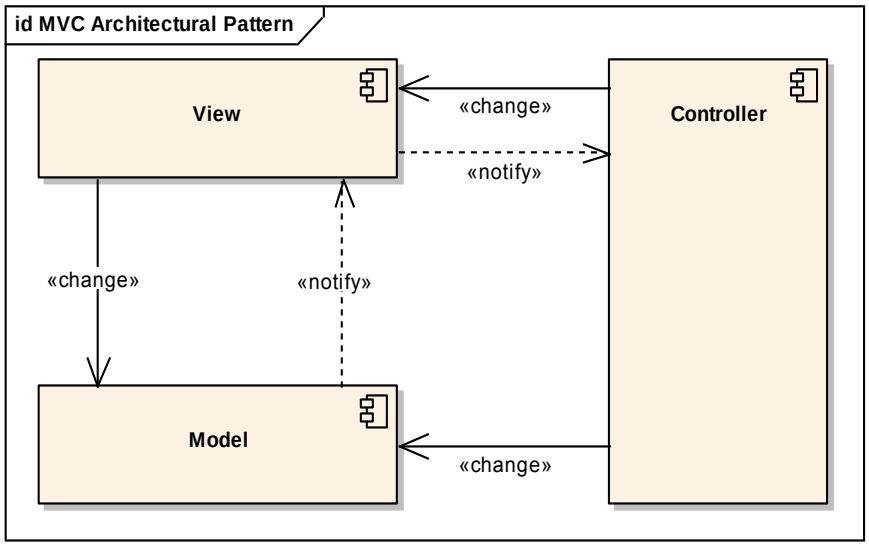


Figure 5.2 – MVC Architectural Pattern.

In this architectural pattern, upon user interaction the Controller accesses the Model, possibly updating it in a way appropriate to the user's action (when using complex controllers it's recommended to use the COMMAND [Gamma1995] design pattern to encapsulate actions), and the view uses the model to generate an appropriate user interface. The view gets its own data from the model, but the latter hasn't any direct knowledge about the view.

The isolation principle introduced by this architectural pattern is ideal to support the multi-view approach implemented in ProjectIT-Studio/Requirements. This way, it's possible to have several presentations of a model on a screen at once. If a user makes a change to the model from one presentation, the others need to change as well. To achieve this behavior without creating a dependency, an OBSERVER pattern [Gamma1995] implementation is required. The presentation acts as the Observer/Subscriber of the model (Publisher): whenever the model changes, it sends out an event and all subscribed presentations refresh their graphical information, thus the model is the role responsible for indirectly notifying all interested parties – potentially including all views – of a given change that have occurred.

The second division, the separation of view and controller, is less important since it's achievable with other design patterns, such as the Strategy design pattern [Gamma1995] (a view with several strategies as controllers). However, in practice most systems have only one

controller per view, so this separation is usually not done unless we are dealing with a web application, in which it makes sense.

5.1 Detailed Tool's Architecture

Figure 5.3 presents ProjectIT-Studio/Requirements' detailed architecture, namely it emphasizes all internal sub-packages and all the relevant plug-ins' behavior classes.

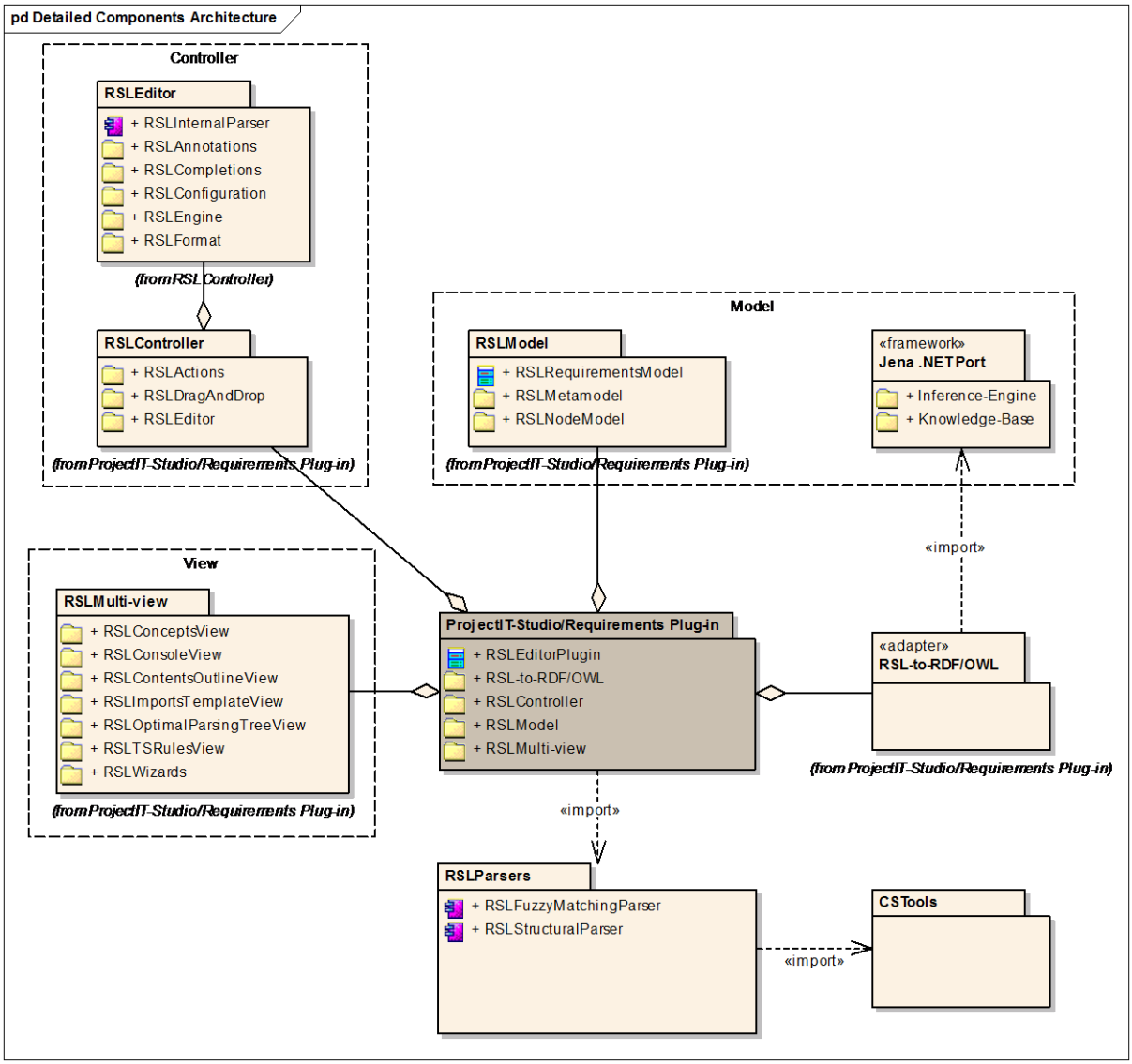


Figure 5.3 – ProjectIT-Studio/Requirements Detailed Components Architecture.

The *PIT-RSL Parser* package contains all PIT-RSL text editor independent parsers, namely: (1) the RSL Fuzzy Matching Parser, which implements a rule-based fuzzy matching algorithm for Natural Language Processing (NLP); and (2) the RSL Structural Parser, which is a parser generated by CSTools (an imported framework) that deals with the early format/typographical normalization and structural analysis stages of the parsing process. The former is responsible for processing natural language text and its goal is to find the optimal parsing tree, by successive comparisons between natural languages patterns (defined by a set of rules called Template Substitution (TS) rules) and the written requirements; every time a valid match occurs, the parser performs the respective rules substitutions in a controlled iterative and exhaustive process, until no more patterns can be applied. Each of these rules is composed by a template part that is used to match the requirements sentences and a substitution part, which replaces the original statement when a match is found; the idea is that, with the generated information, more knowledge is obtained and further analysis and validations can be performed.

In the center, the reader can observe this tool's main package. The *PIT-RSL Plug-in*, whose main purpose was previously described, besides the three sub-packages that implement the MVC architectural pattern package's, also contains all the base classes responsible for the plug-in initialization, logging, and termination.

Due to its importance it becomes essential to analyze PIT-RSL Editor sub-package is very important, as it contains the classes that implement the specific text editor's (the reader may consult Appendix F to observe some screenshots) behavior, namely it encompasses: (1) the code responsible for extending the text editor provided by the Eclipse.NET platform; (2) the *PIT-RSL Annotations* package, which allows the definition of annotation rules for text underlining and overview ruler side marks for warnings and errors; (3) the *PIT-RSL Completions* package, which offers a context dependent content proposal that displays completion tips for the current cursor edit position, being the tip valid only in a determined range of characters and its search space is reduced while the user is typing the initially wanted keyword or import entry (at each moment, the user can specify graphically which option is the more adequate); (4) the *PIT-RSL Configuration* package, which contains the specific scanner classes that allow the definition of context areas in the requirements document, supporting

different behaviors according with the current active area, such as syntax-highlighting, annotations, and auto-completion proposals; (5) the internal sub-package *PIT-RSL Engine*, which implements the required text editor Internal Parser in order to construct the abstract syntax tree of the requirement documents that feeds the input of all the perspective views (such as Content Outline View, RSL Concepts View, and Optimal Parsing Tree View) and allows requirements semantic analysis for providing feedback to the user (such as syntax-highlighting, warnings and errors annotations, and other visual elements that provide guidance to the user in the requirements specification process); and (6) the *PIT-RSL Format* package, which contains the code required for auto-format operations.

The set of views encompassed by PIT-RSL Multi-View package will not be analyzed here since Appendix XXX provides a thorough description of its functionality.

Finally, there are two other packages of extreme importance in this architecture, *RSL-to-RDF/OWL* and *Jena .NET Port*. The latter represents a .NET port of Jena framework, which endows the PIT-Studio/Requirements plug-in with knowledge-base and inference-engine capabilities, typical of a natural language parsing tool. The former contains the adapter pattern code that provides a clean C# API for using the .NET ported Jena framework without the necessary traces of Java syntax code.

5.2 Architectural Components Interaction

Figure 5.4 presents the detailed workflow between the low-level controller components, depicting the underlying parsing process of the ProjectIT-Studio/Requirements tool.

There are several steps involved in the parsing process of a requirements document, which can be done by using one of two types parsing modes: (1) the *batch mode*, or (2) the *incremental mode*. The later provides a rich set of on-the-fly validation features to aid the user during the requirements specification activity because it provides feedback while the user is typing; this parser is deeply interconnected with the ProjectIT-RSL *TextEditor* component, since it uses its mechanisms for graphical presentation of the generated warning and errors during the incremental parsing mode process.

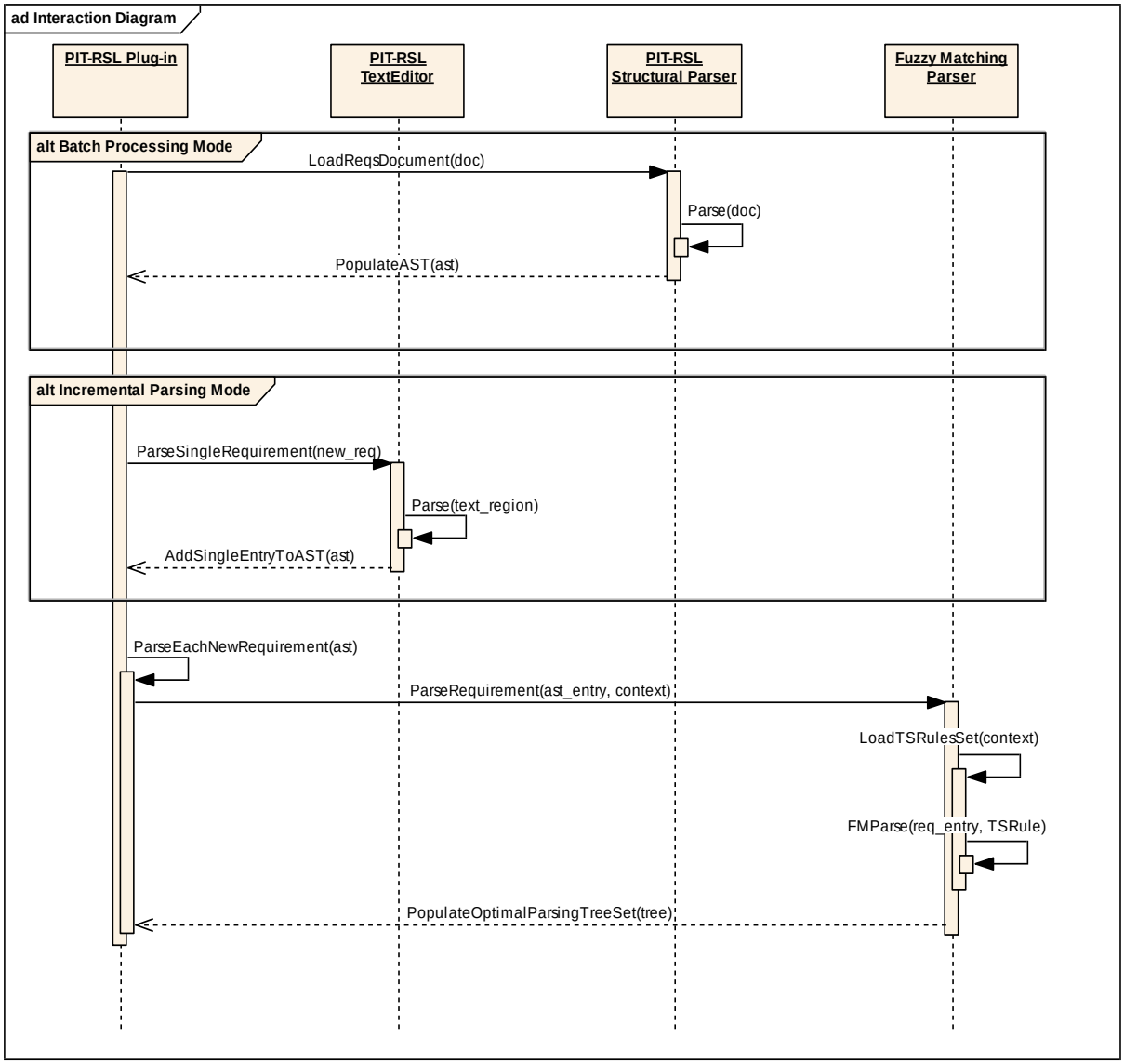


Figure 5.4 – ProjectIT-Studio/Requirements Interaction.

It provides a smooth HMI interaction with the user so that the parsers only analyze newly created requirements in an incremental manner, instead of processing repeatedly the whole document. The former, the batch mode, is useful for processing a bundle of PIT-RSL documents during the import of already-existing requirements documents, or for using the RSL language in a standalone mode (i.e. without the ProjectIT-Studio/Requirements' specialized text editor). It can be used in a stand-alone version of the RSL without any support of the base framework, the ProjectIT-Studio.

Figure 5.4 also illustrates the reader also can observe the dependence of Fuzzy Matching Parsing in the availability of a specific TS rules set as input to work correctly.

The early transformation stages performed by the referred parsing processes manipulate each requirement statement in the following order: (1) typographical and format transformations (including putting all text in one line, normalization of white spaces, and word contractions expansion); and (2) morphological syntactic analysis, which includes the text annotation with the relevant part-of-speech tags.

After the conclusion of one of these alternative parsing modes processes (which are essential to analyze the document's structure and normalize the requirements document text), the ProjectIT-RSL plug-in iterates over the abstract syntax tree produced by Structural Parser (SP) and delegates on the Fuzzy Matching Parser (FMP) the responsibility of finding the optimal parsing tree for each of the parsed requirements, according to the requirement's context identified earlier. The context identification allows the fuzzy-matching parser to load only the relevant template substitution (TS) rules; then, for each rule in the retrieved set of TS rules, the Fuzzy Matching Parser iteratively finds the optimal parsing tree for that requirement by exhaustion of TS rules set or timeout. The process continues, as long as there are new requirements to parse. The result is stored as a set of domain-based parsing trees, one for each requirement.

Fuzzy Parsing is a parsing technique that goes further than the traditional compiler-based parsing techniques. It offers several advantages in the context of natural language processing since it provides a more flexible, robust, and configurable approach for matching predefined patterns without the syntactic strictness of usual programming language's compilers. By defining patterns' templates, it is possible to discover those whose match best suits the implicit free-form natural language text semantics, due to the semantic adherence to the syntactic structure of Natural Language sentences.

The information generated by the Fuzzy Matching Parser is stored in a knowledge base. This knowledge-base, which will be implemented as a semantic network and stored as RDF/OWL, is later analyzed by an inference engine, with the goal of extracting implicit knowledge from the initial requirements. To implement the semantic network used as the knowledge base and inference engine, it was chosen the Jena project, a Java framework that provides a solid

environment to define and manipulate concepts via RFF/OWL data specification; the conversion of Jena to .NET Framework was done using IKVM.NET, a Java Virtual Machine for the .NET runtime (for further information the reader may consult Appendix E).

5.3 Components Integration and Underlying Workflow

Figure 5.5 presents a non-standard UML diagram whose main objective is to provide an overall view of ProjectIT-Studio/Requirements' internal components, the process' workflows, the intrinsic dependency relations, and the information exchanged between them.

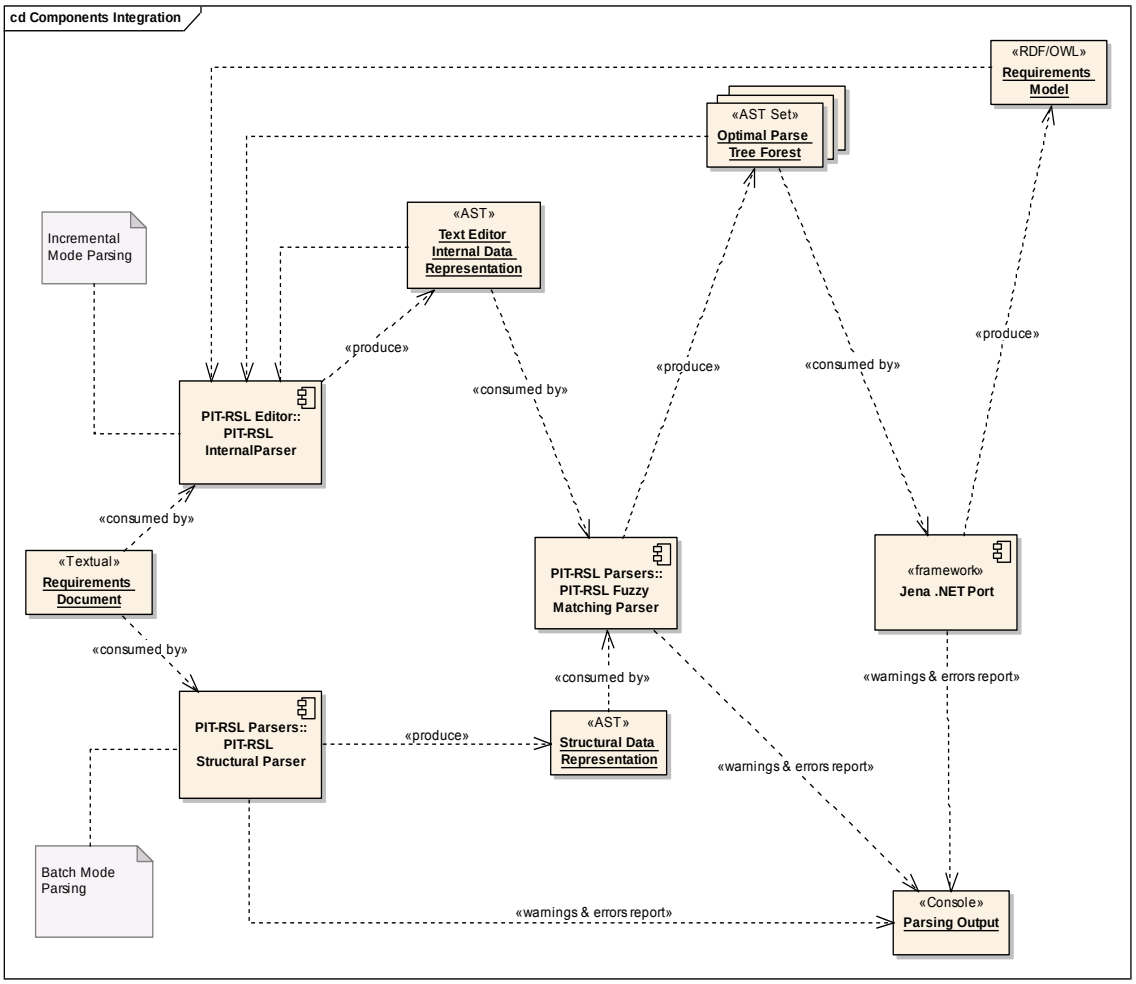


Figure 5.5 – ProjectIT-Studio/Requirements Components Integration.

Figure 5.6 presents the overall workflow of the tool's components and the exchange artifacts.

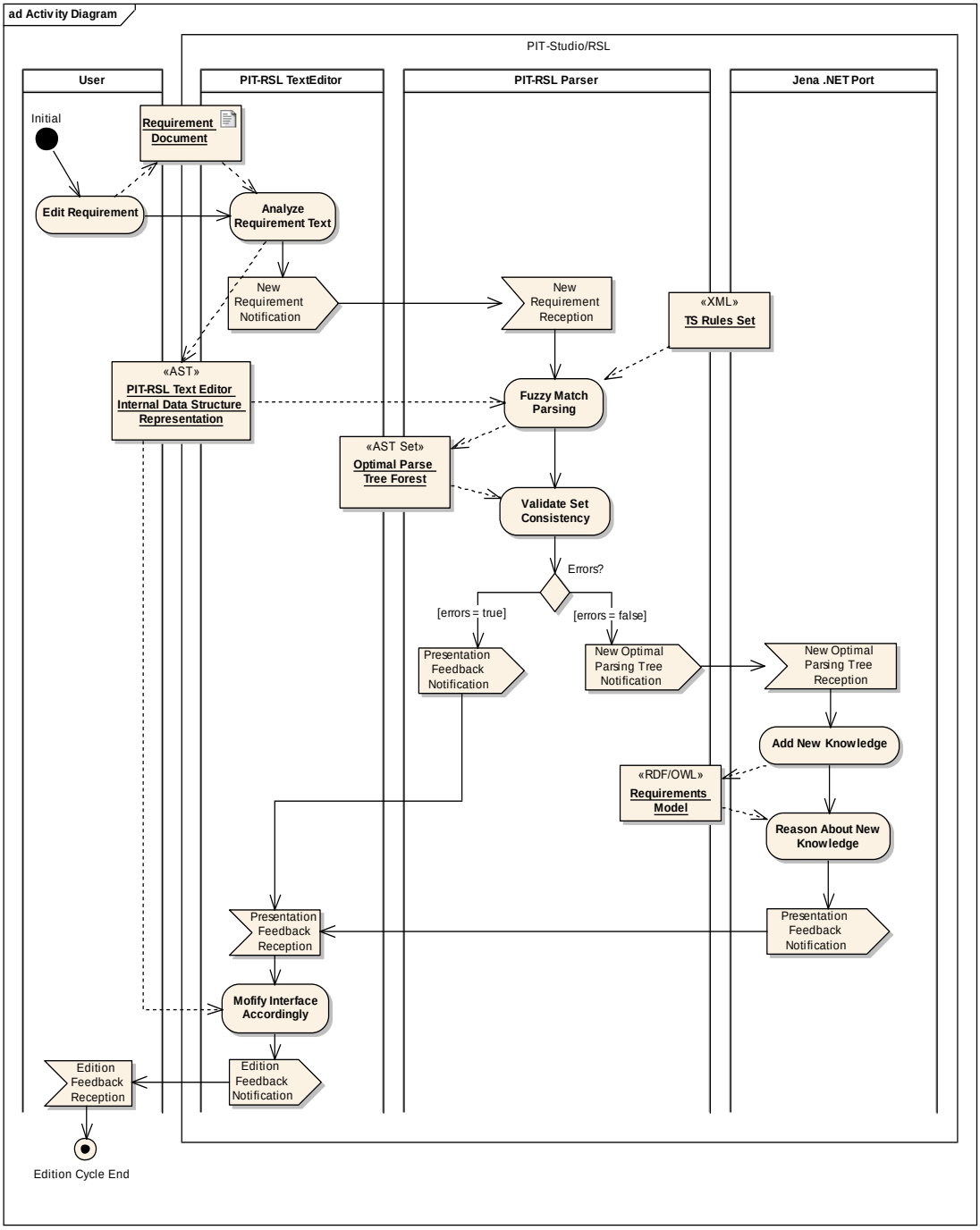


Figure 5.6 – ProjectIT-Studio/Requirements User's Interaction Workflow.

5.4 Integration Architecture with ProjectIT-Studio

One of the final goals of this work was to integrate it with the rest of ProjectIT-Studio tools. Eclipse.NET provides the required level of independence between all tools (plug-ins) while simultaneously providing extension-based mechanisms to ease cooperation between them [Saraiva2005a] through an elegant and effective communication channel. Figure 5.7 illustrates how the ProjectIT-Studio plug-ins are combined and working together.

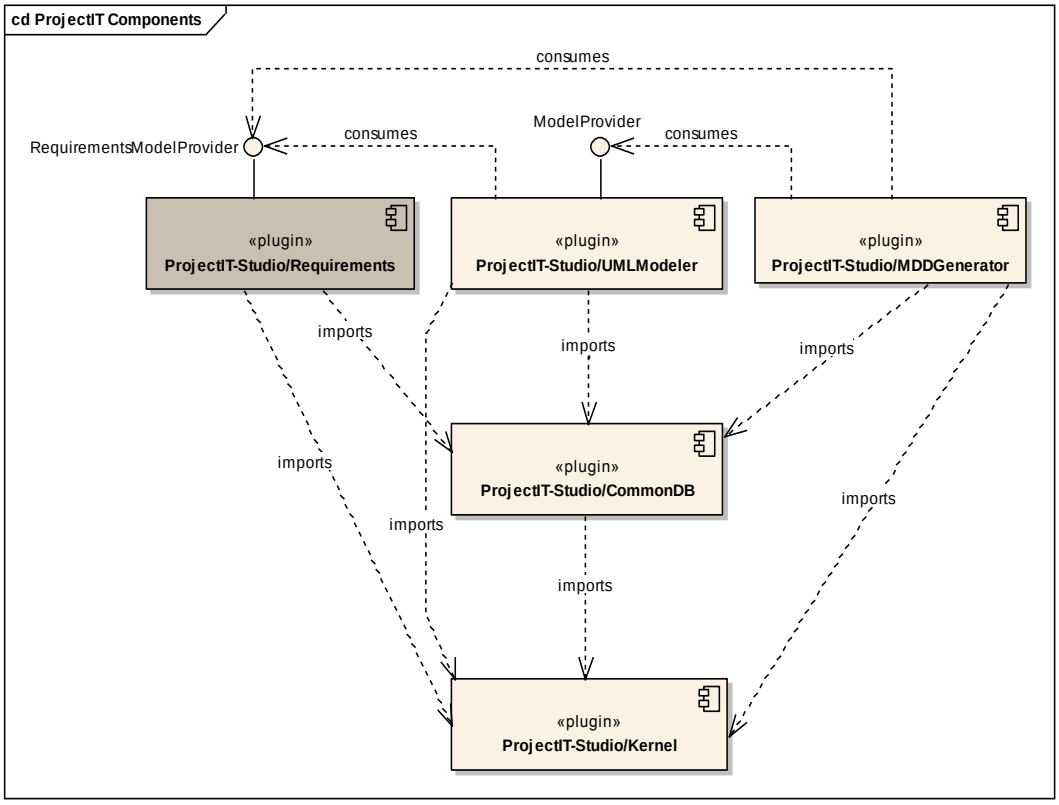


Figure 5.7 – ProjectIT-Studio/Requirements Integration Architecture (extracted from [Silva06])

The ProjectIT-Studio/Kernel and ProjectIT-Studio/CommonDB plug-ins are foundation plug-ins, whose function is to support the development-oriented, top-level plug-ins, namely ProjectIT-Studio/Requirements, ProjectIT-Studio/UML Modeler and ProjectIT-Studio/Generator. This plug-in provides basic framework facilities, like the UML2 metamodel, to the top-level plug-ins.

As common sense dictates, applications typically use a persistent storage mechanism (such as a Database Management System (DBMS)) to store data; this is also the case of ProjectIT-Studio. The ProjectIT-Studio/CommonDB plug-in allows the top-level ProjectIT-Studio plug-ins to serialize and deserialize information to a persistent storage mechanism (such as a relational database), decoupling persistence details from those plug-ins. Serialization to a persistence medium used by multiple ProjectIT-Studio instances will allow those instances to be synchronized with each other. However, this component is still under design. To achieve a deeper level of integration the ProjectIT-Studio/Requirements plug-in should provide, beside the in-memory models integration already accomplished, a mechanism to interact with the ProjectIT-Studio/CommonDB plug-in in a near future. Since ProjectIT-Studio/Requirements implements the MVC architectural pattern its data access layer is not illustrated in the previous architectural diagrams, because it is understood to be underneath or encapsulated by the Model component.

Nevertheless, top-level plug-ins also interact with each other, through Eclipse.NET's extension-point mechanism, namely ProjectIT-Studio/Requirements declares an extension-point, designated "RequirementsModelProvider", which means that the plug-in can provide models; any plug-ins that can process (i.e. consume) models produced by ProjectIT-Studio/Requirements should declare themselves as consumers of this extension-point.

For further information on the interaction mechanisms of the other top-level plug-in the reader may consult [Saraiva2005a] [Silva2006].

6 PROOF-OF-CONCEPT CASE STUDY – MYORDERS 2.0

The main purpose of this appendix is to provide a deeper insight in the academic system model chosen to provide a proof-of-concept of the elaborated requirements specification language, the ProjectIT-RSL, and the supporting CASE tool, the ProjectIT-Studio/Requirements, that provide the underlying parsing mechanisms that can extract computer-understandable information from natural language free-form text, thus analyzing and validating the system's specification, giving instant visual feedback to the user through a set of complementary views (multi-view approach).

As previously mentioned in chapter 6, the translation process adopted, to convert the domain model specified in UML's XIS2 profile to the respective PIT-RSL textual representation, was very simple, since it basically consisted in analyzing the standard UML graphical model and write it down in natural language text. There were some idiosyncrasies, namely in terms of specific XIS2 concepts that are not clearly mapped into the more general purpose PIT-RSL metamodel, and some minor unnatural language explicit constructs that are required to avoid the specification of "messy" Template Substitution (TS) rules.

This conversion between the previous specified MyOrders 2.0 domain model into a PIT-RSL specification can be made following two different approaches that reflect the respective different scenarios of usage of the ProjectIT-Studio/Requirements: (1) the user can write down the entire MyOrders 2.0 specification in a typical character-based editor and then create a new project and import this requirements document file (with .rsl extension) with the ProjectIT-Studio/Requirements tool; alternatively, (2) the user can use the developed CASE tool to create the project and a requirements document from scratch, taking advantage of the full potential provided to assist the activity of requirements specification and validation, such as error reporting and a comprehensive set of views (multi-view approach).

6.1 MyOrders 2.0 Informal Description

As mentioned before, the purpose of this simple case study system is supporting the typical interaction activities between clients and suppliers, to be specific the activity of ordering products. The underlying scenario provided by this case study is so abstract that it can be

applied to almost every enterprise's category, since, although this is a simplification of the real world thus having some inherent restrictions, it can be observed in a wide range of enterprises, from the most rudimentary to the most complex ones.

By observing the UML domain model depicted in Figure 6.2, the reader will identify the following entities: (1) *ThirdParty*, represents an abstract entity corresponding to the concept of a typical enterprise having as attributes its name, a list of Affiliates, and a list of Orders; (2) *Affiliate*, corresponds to an enterprise's partner or a enterprise's representative member (hierarchy relation), and has a list of attributes essentially containing information about its contact (address, phone, and fax) and the category of affiliation; (3) *AffiliateType*, specifies the relation between the ThirdParty and its Affiliate; (4) *Customer*, represents the typical buyer role in which an entity poses an order request to a product provider, and it is characterized by a list of at least one Market, an importance, and flexibility attributes; (5) *Market*, match the real market concept representing the place (real or virtual) where buyers and sellers interact to exchange goods and services for money, for the sake of simplicity this concept is described only by its name; (6) *Supplier*, clearly maps to the entity that provides the solicited Product during the buying interaction usually initiated by the Customer, this concept is described by two evaluation metrics, its quality and responsiveness, and a list (products catalog) of provided, and eventually produced, Products; (7) *Product*, represents the traded good or service being described by a set of typical attributes such as name, price, units, and production state; (8) *Order*, embody the Product's request and is described by an identification and its relative events' dates attributes; (9) *OrderDetail*, captures the information relating a client's order to a specific product provided by a seller, providing a concrete item's order description.

As mentioned in Chapter 1, all the ProjectIT's tools, namely ProjectIT-Studio/Requirements, ProjectIT-Studio/UMLModeler, and ProjectIT-Studio/Generator, are specialized and incorporate the knowledge gathered by experience of the daily projects in which GSI of INESC-ID are involved, which are mainly interactive systems (as the sharp reader may already noticed from some of the acronyms already introduced). This category of software information systems, by definition (a system that allows a dialog between the computer and a user), requires the definition of the user roles (the so called actors) that will interact with the

system. In this particular case, MyOrders 2.0 system's context, the required definition of the participants that can interact with the system are present in Figure 6.3, which presents through an UML diagram the hierarchy of user roles (actors view). The user can observe that there is an abstract parent actor, the "User", which as two concrete children, "URegistered" and "UnRegistered", indicating if the user has performed logged-in (authentication) operation in the system or not, respectively. After being recognized (correctly authenticated) by the system, the user can assume two other specialized roles, namely: (1) "UManager", which corresponds to a user that has the role of managing the application's data, having the authority to perform typical CRUD operations on the business entities modeled by this system (explained above in domain model description); and (2) "UAdministrator", whose main responsibility is to perform usual systems' administration tasks like managing users' accounts.

6.2 MyOrders 2.0 ProjectIT-RSL Specifications

This section presents the textual specifications used to capture MyOrders 2.0 domain model concepts. Each frame will reveal an important requirements specification fragment.

6.2.1 Textual Specifications

As common sense dictates, the first section is used to provide an overview of the requirements document's purpose and contents. To achieve this goal the user must use *Section Introduction*, instead of a *Section Comments*, because the former will be stored for further usage and the latter won't. *Section Introduction* can be very useful for providing a brief description during document's search and cataloguing operations (to facilitate its contents reuse), or even to including document's synopsis as a header in an automatic generated report.

```
1 Section Introduction
      Description of a simple order management software information system.
      Main objective: to provide a simple case study for proof of concept
      of the ProjectIT-Studio/Requirements.
End Section
```

After defining the *Section Introduction*, the user must specify the main application unit region, called *System* context, followed by the system's name, in this case MyOrders2. At this scope level, users can specify as many *Section Comments* as they want because they are neither parsed nor stored, since they only provide an auxiliary annotation mechanism for

enriching the specifications description. Besides the *Section Comments*, at *Systems* scope level the user can specify a *Section Imports* (only one per *System* region). This specific section type is used to declare import entries, which when parsed by Fuzzy Matching Parser (FMP) will make it invoke Structural Parser (SP) import mechanism to include their contents in the document's underlying concepts model.

```
2 System "MyOrders2"

      2.1 Section Comments
            The will describe the domain view, corresponding to the
            static/structural view of the system.
      End Section

      2.2 Section Imports
            - use \Documents\Document.
            - use \Documents\Invoice.
      End Section
(...)
```

The section that follows is of extreme importance because, through its specifications, one can define the under specification system's domain model, in this case MyOrders2 domain model.

The next frame also presents the textual entries that will be used in a near future to introduce variability points to requirements documents (implementing an edit point template-based reuse mechanism), hence providing specifications' placeholders like a template's form.

```
2.4 Section Business Entities
    <business entities pre edit point>

    2.4.1 A thirdParty is an entity.
    2.4.2 Each thirdParty has a 'company name' and a list of affiliates.

    2.4.3 An affiliate is an entity.
    2.4.4 Each affiliate has an 'affiliate type', and a 'contact name', and
          an address, and a 'contact title'.

    2.4.5 An address is composed by a city, and a 'postal code', and a
          country, and a phone, and a fax.

    2.4.6 An 'affiliate type' has a description.

    2.4.7 A supplier is a thirdParty.
    2.4.8 Each supplier has a list of 'order details', and an integer
          quality, and an integer responsiveness.

    2.4.9 A customer is a thirdParty.
    2.4.10 A customer is the same as a client.
    2.4.11 Each costumer has an integer importance and an integer
          flexibility.
```

```
    2.4.12 Each costumer has one or more markets.
    2.4.13 Each market has a list of markets.

    2.4.14 A market is an entity.
    2.4.15 A market has a name.

    2.4.16 Each supplier provides a list of products.
    2.4.17 Each product is provided by one or more suppliers.

    2.4.18 A product is an entity.
    2.4.19 A product has a 'product name', and a double 'unit price', and
           an integer 'order units', and an integer 'stock units', and
           boolean 'discontinued'.

    2.4.20 A customer emits a list of orders.
    2.4.21 Each order is emitted by one customer.

    2.4.22 An order is an entity.
    2.4.23 An order has an 'order code', and an integer 'order number', and
           a date 'order date', and a date 'required date', and a date
           'shipped date'.

    2.4.24 An order is composed by a list of "order details".

    2.4.25 An 'order detail' is an entity.
    2.4.26 An 'order detail' has a supplier, and an integer quantity, and a
           double 'unit price', and a double discount.

    <business entities pos edit point>
End Section
```

The next frame basically presents MyOrders2 user roles (actors) hierarchy specification.

```
2.5 Section Functional Requirements
    <functional requirements edit point>

    // actors and operations declarations, operations definition

    2.5.1 Section Actors Declaration
        2.5.1.1 The User is an actor.
        2.5.1.2 The UUnRegistered is a User.
        2.5.1.3 The URegistered is a User.
        2.5.1.4 The UManager is an URegistered.
        2.5.1.5 The UAdministrator is an URegistered.
        2.5.1.6 The Person is the same as User.
    End Section

    2.5.2 Section Actors Definition
        2.5.2.1 UManager can create, and edit, and delete a customer.
        2.5.2.2 URegistered can create a customer.
        2.5.2.3 The User can create, and read, and update, and edit, and
                delete a thirdParty.
    End Section

End Section
```

6.2.2 TS Rules Usage Example

This section intends to give a brief overview (following an user perspective) of the steps involved in the construction of an optimal parsing tree by using fuzzy match parsing techniques (for further investigation the reader may consult Appendix C, which presents all the concepts and operations involved, through a medium complexity example's explanation).

By only considering the small Template Substitution (TS) rules subset presented in Table 6.1 and a simple requirement statement, such as "*A client is an entity and has a name, and an address, and an account*", ProjectIT-Studio/Requirements can obtain the optimal parsing tree illustrated in Figure 6.1.

Table 6.1 – TS Rules Subset.

RULE	TEMPLATE	SUBSTITUTION
TS_1	`new_entity/PROP IS ENTITY`	`$NODE/ENT`
TS_2	`new_entity/ENT HAS list/LIST`	`$NODE/ENT`
TS_3	`noun/NN`	`$NODE/PROP`
TS_4	`prop1/PROP AND prop2/PROP`	`$NODE/LIST`
TS_5	`list/LIST AND prop/PROP`	`$NODE/LIST`

The best parsing results would be achieved by performing the following sequence of steps:

- Apply **TS_3** to all terminal tokens (statement's words), thus getting the first parsing layer only composed by *Property* (PROP) non-terminal nodes (nodes **@6**, **@3**, **@2**, and **@1**).
- Apply **TS_1** to achieve an *Entity Inheritance Definition* (EID) node (node **@7**).
- Apply **TS_4** to create the first *List* (LIST) non-terminal node (node **@4**).
- Next apply **TS_5**, which supports left-based recursion, to create the second *List* (LIST) non-terminal node (node **@5**).
- Finally, by applying **TS_2** one can achieve an *Entity Property Definition* (EPD) non-terminal node **@8**, which corresponds to the optimal tree root node.

Despite being simple, this example provides a good insight on the underlying parsing step performed by the Fuzzy Matching Parser provided by ProjectIT-Studio/Requirements.

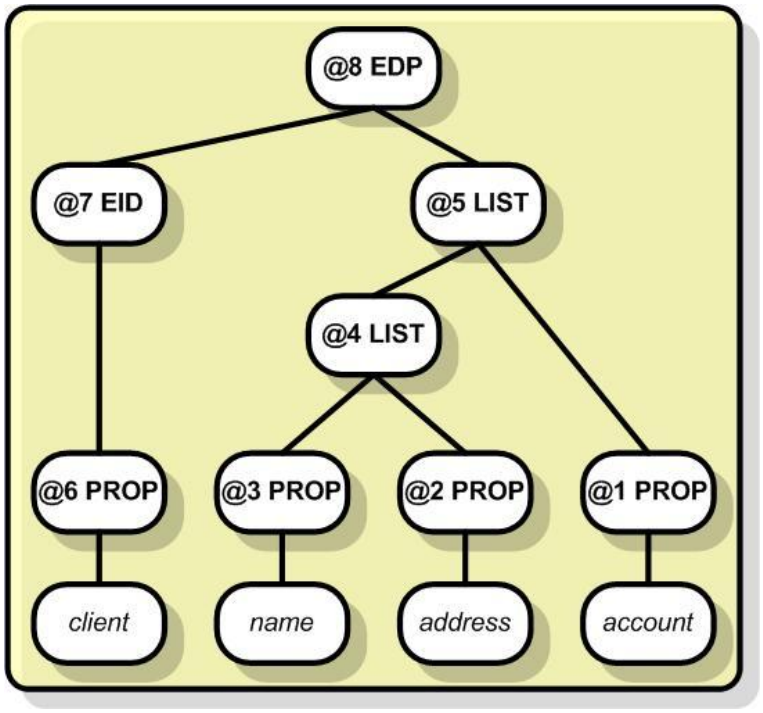


Figure 6.1 – Optimal Parsing Tree Example.

6.2.3 MyOrders 2.0 XIS2 Models

Figure 6.2 presents the UML diagram, after UML XIS2 profile has been applied, of the MyOrders 2.0 system's domain model. It depicts the core concepts modeled by this simple abstract system and the relations between them. This diagram is thoroughly explained above in MyOrders 2.0 Informal Description subsection.

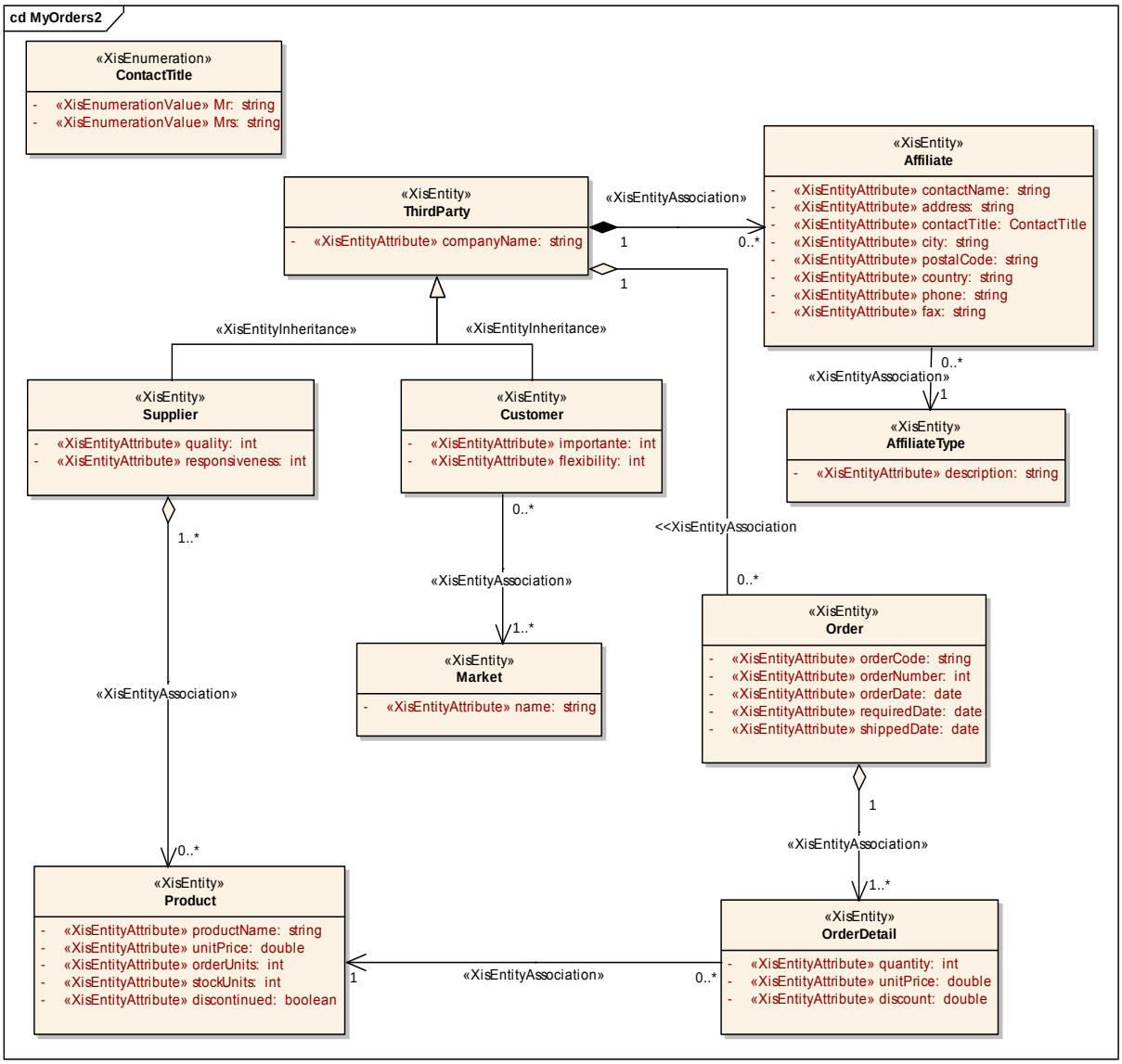


Figure 6.2 – MyOrders 2.0 XIS2 Domain View.

Figure 6.3 presents the UML diagram, after UML XIS2 profile has been applied, of the MyOrders 2.0 system actors' model. It depicts the roles that a user can assume while operating with this interactive system by means of the authentication mechanism. This diagram is thoroughly explained above in MyOrders 2.0 Informal Description subsection.

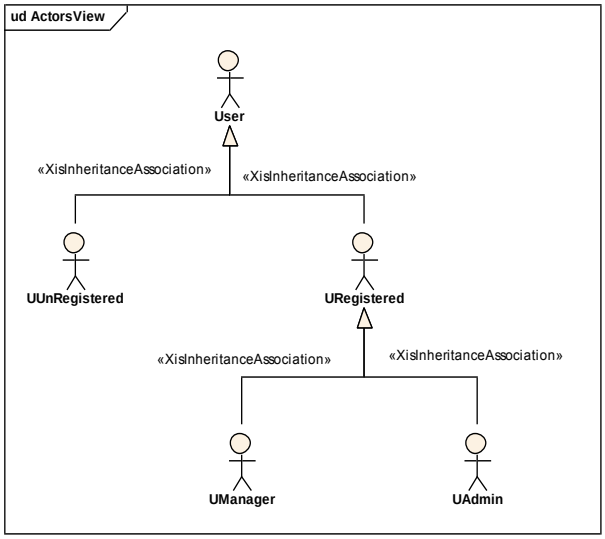


Figure 6.3 – MyOrders 2.0 XIS2 Actors View.

Figure 6.4 presents the UML diagram, after UML XIS2 profile has been applied, of the MyOrders 2.0 system of use case for managing a customer (business entity). It depicts the actor and which operations each of them can perform on the customer entity, according to their roles.

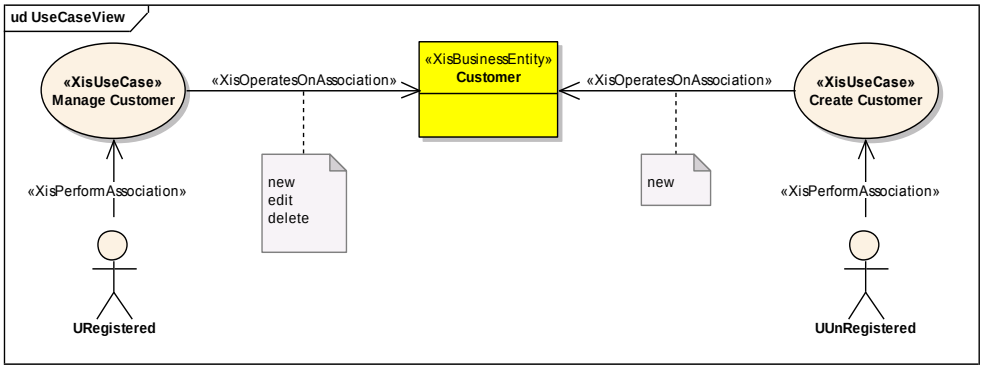


Figure 6.4 – MyOrders 2.0 XIS2 Use Case View.

6.2.4 MyOrders 2.0 Runtime Screenshot

This subsection is devoted to the presentation of some screens of MyOrders 2.0 system in an execution environment, after it has been specified with ProjectIT-Studio/Requirements, edited with ProjectIT-Studio/UMLModeler to apply XIS2 UML profile, and finally generated by ProjectIT-Studio/Generator. These screenshots emphasize the potential of ProjectIT-Studio tools to cover the complete software product life-cycle, assisting the participants (requirements engineer, software architect, designer, and programmer) of the software development process with specific CASE tools that accelerate the time-consuming and error prone activities they have to perform repeatedly. The end-to-end process coverage offered by this set of tools, represent a major gain of productivity and quality of the software development process for interactive systems.

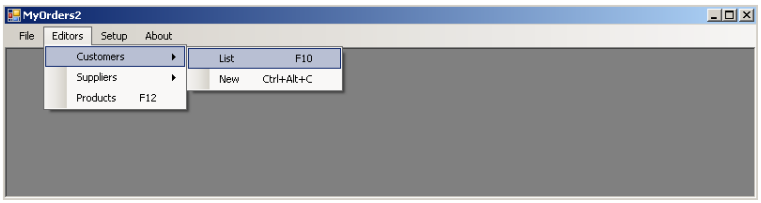


Figure 6.5 – MyOrders 2.0 Application Screenshot.

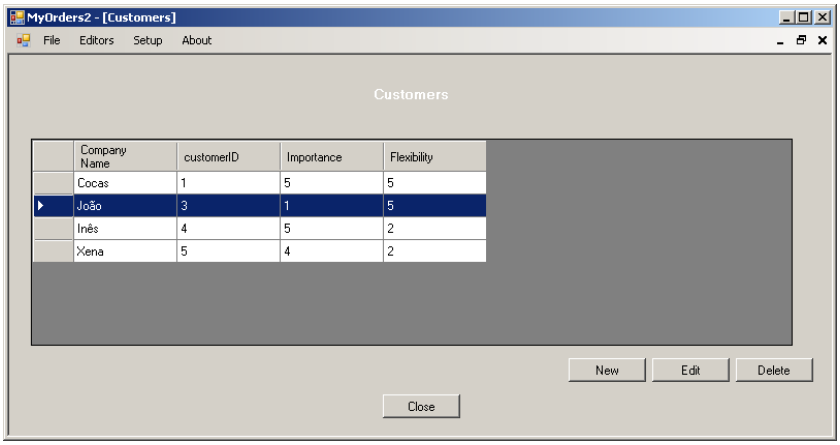


Figure 6.6 – MyOrders 2.0 Customers Screenshot.

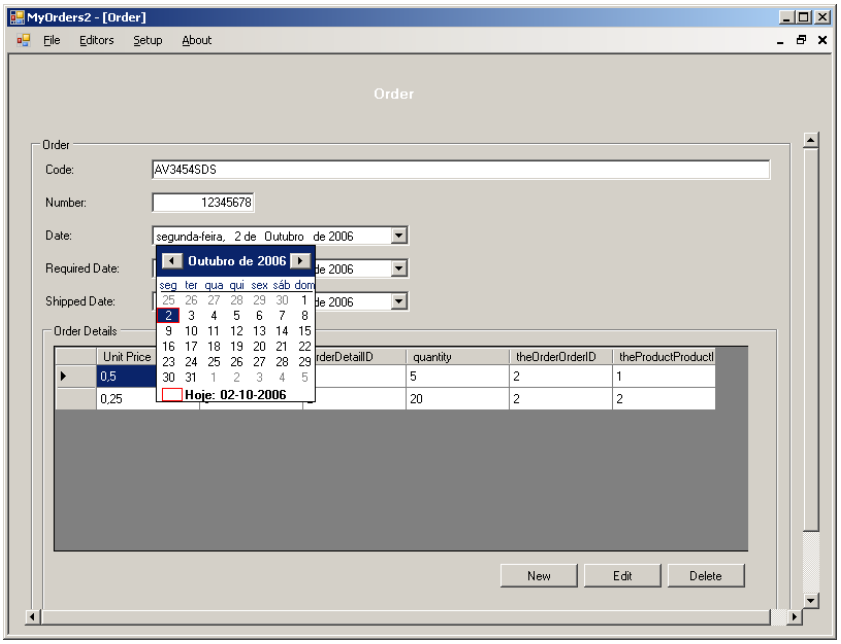


Figure 6.7 – MyOrders 2.0 Customers Screenshot.

6.3 Discussion

This case study provided a good insight of ProjectIT-Studio/Requirements inevitable immaturity, since it lacks some functionality to improve users interaction, namely to provide more contextualized and user friendly information, to enhance the validation process. This tool still presents some natural language processing limitations due the underlying part-of-speech tagger, which not always achieves the expected results (a common NLP field problem). Since ProjectIT-Requirements parsing mechanisms (namely Fuzzy Matching Parser (FMP)) are strongly dependent on the results provided (tag-based words' morphologic classification) by this external component, every time a word is incorrectly labeled, it compromises all the subsequent steps of the parsing and validation mechanisms. However, the achieved results are satisfactory, given the project's research nature and the context (time-frame and manpower involved) in which it was developed.

7 RELATED WORK DISCUSSION

The purpose of this chapter is to present the main advantages achieved with the software application developed under the scope of this work, namely the existence of a requirements specification language and a supporting CASE tool that performs syntax and semantic on-the-fly validation, when compared with other projects and available commercial tools. Additionally, one has to bear in mind the influence resulting from the underlying broader context in which this work is inserted (ProjectIT initiative), namely in terms of the benefits accomplished by a deep integration between this work and other ProjectIT-Studio's Model Driven Development (MDD) tools.

7.1 ProjectIT-RSL Visual Studio Functional Prototype

Initially, this work was supposed to be a direct migration of the already existent ProjectIT-Requirements functional prototype [Carmo2005] to the Eclipse.NET platform, the adopted common software development workbench for all the ProjectIT tools. However, the NLP study and related work analysis, which were undertaken during the research phase, revealed that this functional prototype suffered from several critical limitations in terms of used parsing techniques and from the absence of a language extensibility mechanism. Considering these problems combined with some feasible solutions found, it became essential to make a decision, which ended to be the most ambitious but, simultaneously, the most risky one.

7.2 General Model of Language Understanding

Despite not being a full-fledged Natural Language Processing (NLP) framework (whose development would certainly require several years of hard work and a considerable complex interdisciplinary team composed by linguistics, psycholinguistics, computational linguistics, and programmers), like most projects of this category [Allen1995]), this work presents a pragmatic rule-based approach for processing and capturing the information specified in requirement documents, the only kind of solution attainable in a so short period for such a project. If it was decided to start building a general model of language understanding, it would not be completed at the end of this project time-frame and so it would perform miserably on the final tests. However, the adopted solution may represent a problem in the

near future if its growth is not correctly managed. Although the rule-based approach is much more flexible and in a few months of programming it will appear to outperform by far more complex general Natural Language Processing (NLP) frameworks during tests, it will be required to take double precautions about the misleadingly high performance reported in tests due a strong bias resulting from only including typical domain interactions that do not test the system's limits. This solution is optimal for short-term performance achievement (if this is the only progress criteria adopted) and for low complexity language domain applications, in which the main objective is to fine-tune the underlying language understanding model. However, since the ProjectIT initiative represents a generic approach, even though there is a domain specific adaptation mechanism through TS rules management, it may become troublesome to handle the growth of rules cardinality and complexity, since it will be more likely that rules become interdependent or collide among themselves.

7.3 Text editors

Despite the fact that ProjectIT-Studio/Requirements (the CASE tool developed in the scope of this work) not offers all the generic text editors and IDEs features detected during the thorough analysis made during the initial research phase, it still provides most of the top ten features captured (the ones that were not implemented resulted from platform's internal problems and, since they were taking too long to solve, they were temporarily suspended). Moreover, this tool supports specific Requirements Engineering activities guidance, assisted by a rich set of GUI components that support the multi-perspective approach, specifically on-the-fly multi-channel feedback during the specification activity, aiding the user to formulate a correct specification and comprehension of the whole requirements model through the exploration and validation of its constituent parts, the individual requirements. This type of functionality (integrated multi-view approach) is only available in highly complex IDEs, like Visual Studio or the Eclipse JDT and CDT.

7.4 Commercial tools

The commercial tools available at the time are mostly requirements management tools. As already mentioned in Chapter 2, current commercial tools are focused on: (1) requirements traceability (authorship and interconnected hierarchies); (2) manual validation through check-

boxes; (3) artifacts connected by links and navigation between them; and (4) import and export features to the usual textual and image formats. So bearing in mind the features provided by the implemented tool, one can observe that the adopted approach is clearly complementary to the one adopted by the software industry. On one hand, we have the commercial tools with a table-based or graph-based approach that, despite having the merit of solving some important issues during requirements specification and validation, lack the ability of interpreting and perform inference-based unitary and overall requirements validation, treating them as black-boxes. On the other hand, we have the ProjectIT-Requirements approach which adopted a natural language text-based approach to specify requirements (the most usual one among humans, technical or non-technical stakeholders) enriched by a set of views that provide instant feedback of the written specifications. As a matter of fact, ProjectIT-Requirements' initial philosophy was never to build a requirements management tool, but rather to build a requirements writing tool like a word processor with error correction, as previously mentioned. However, to be a full-fledged solution for Requirements Engineering activities, ProjectIT-Studio/Requirements tool will require in a near future (after the underlying parsing mechanism is considered stable) to implement some additional concepts, such as: (1) a wide ProjectIT-Studio end-to-end traceability mechanism to relate a given requirement with all the artifacts produced along the software development process; and (2) a hyperlink-based navigation mechanism between artifacts, namely by performing a predefined keystroke and a mouse click (like JDT navigation mechanism) it would be possible to navigate from a requirement's entity or actor to all the UML domain models and use cases (ProjectIT-Studio/UML Modeler artifacts) in which they are involved, or even to the respective source code that will implement them (ProjectIT-Studio/Generator artifacts).

7.5 Other Research Project

ProjectIT-Studio/Requirements tool has several similarities with other NLP and requirements specification language research projects mentioned in Chapter 2. This work sought to incorporate the best concepts and techniques from each one of them, namely: (1) CIRCE, which proposes a "lightweight formal method" approach based on fuzzy matching domain-based parsing techniques to produce a formal validation of requirements written in natural

language, complemented with a GUI multiple-perspective-based approach to provide feedback to the user; (2) ACE, which presents the concept of using a controlled natural language to write precise specifications that, for example, enable their translation into first-order logic (FOL); and (3) NL-OOPS, which influenced this work by presenting its knowledge-base and inference-engine architecture. The final result was a solution that incorporates the best from both worlds: (1) the flexibility, expressive power, and familiarity of natural language; and (2) the robustness, validation, and inference of formal approaches.

Additionally, it applied the approach proposed by Abbot [Abbot1983] in the syntax-highlighting mechanism, emphasizing the suggested implicit relations between the following natural language morphologic categories matches with computational concepts pairs: nouns and classes/entities; adjectives to describe attributes, and verbs to identify methods.

As a side note, it's important to note that the three projects that more contributed to the theory foundations and to the current state of art of this work, mentioned above, were recognized as remarkable by the software industry: (1) CIRCE received the IBM's "Eclipse Innovative Award" in 2003 and it's currently a Eclipse Research Community project (available at http://www.eclipse.org/technology/research.php); (2) ACE, has received several article related distinctions and was adopted as the controlled language of the EU Network of Excellence REWERSE (available at http://rewerse.net/) (Reasoning on the Web with Semantics and Rules); and (3) NL-OOPS became a commercial tool.

7.6 Comparative Analysis

Figure 7.1 provides a good insight of this analysis subject through a graphical overview of the issues previously analyzed. Has one can clearly observe there are three main groups, each one representing a peculiar type of requirements solutions, namely: (1) *Requirements Management Tools*, comprising well-know commercial applications such as IBM's Rational RequisitePro and Telelogic's DOORS; (2) *ProjectIT-Studio*, with its specialized requirements plug-in (ProjectIT-Studio/Requirements); and (3) *NLP Research Projects*, which encompasses more mature NLP projects, such as CIRCE, NL-OOPS, and ACE. From this figure, the reader can verify that: (1) *Requirements Management Tools* offer deeply integrated solutions with a strong emphasis in traceability issues; (2) *NLP Research Projects* present a

strong component on natural language and formal validation topics (as expected), but their weakness, the lack on robust traceability mechanisms and deep integration with MDD tools, comes from their specificity; (3) ProjectIT-Studio/Requirements stands in the middle because, despite offering NLP functionality, it has additionally a strong focus on its integration with ProjectIT-MDD component tools, namely ProjectIT-Studio/UML Modeler and ProjectIT-Studio/Generator.

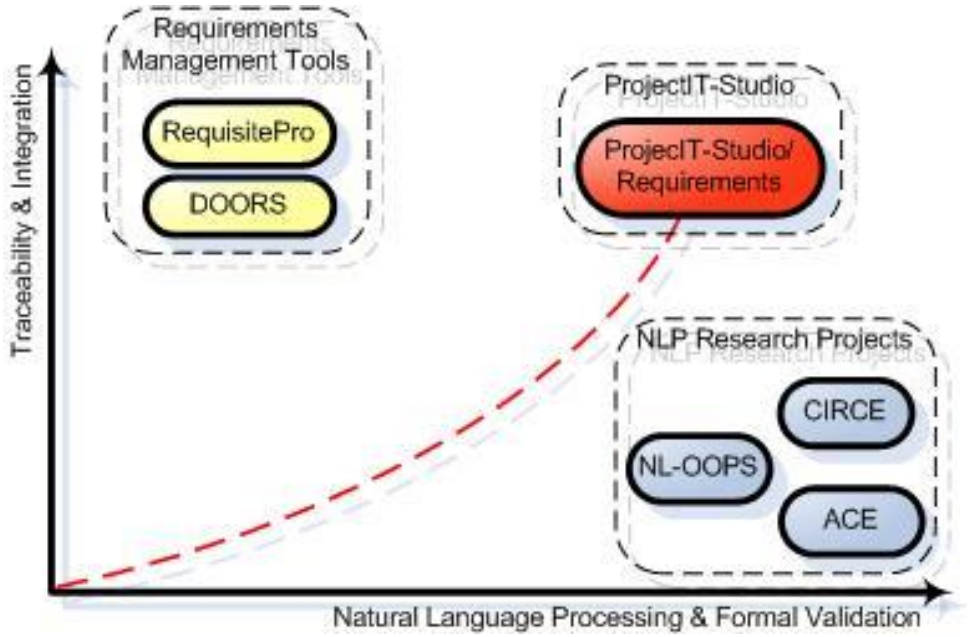


Figure 7.1 – Comparative Analysis.

8 CONCLUSIONS AND FUTURE WORK

This chapter is dedicated to the presentation of overall considerations about how this work was conducted and how it has evolved during the time-frame available to its conclusion, which was almost a year. It will be presented a comprehensive discussion of the work developed and improvements that could have been made during the software development process. In the end, the reader will be presented with a descriptive subsection of planned future work to continue this project in a DEIC's PhD context.

8.1 Argumentative Analysis of the Developed Work

This subsection focus on the technical and software development contributions of this work, and on some of the author's reflections and learnings achieved with this work.

8.1.1 Performed Work

The results achieved with this work considerably contributed to the ProjectIT-Requirements project, namely by enhancing the ProjectIT-RSL language with natural language and DSL characteristics – controlled natural language – and by introducing a supporting CASE tool as an Eclipse.NET plug-in, the ProjectIT-Studio/Requirements, which due to its deeply integration with other ProjectIT initiative's tools also contributes to the global improvement of this underlying innovative initiative.

The implemented tool provides a specific PIT-RSL text editor that supports all the typical IDE features such as on-the-fly syntactic verification and syntax highlighting, which means that as we are writing, all expressions are validated and in case there is an error, this one is immediately detected and highlighted. The auto-complete feature is also always available presenting hints of how to complete the sentence. Additionally, the alignment with ProjectIT-MDD component metamodel and the deep accomplished integration with ProjectIT-Studio/MDD tools allows a straightforward mapping between them, thus, upon the creation of a correct and consistent requirements model, the process can be continued by the MDD component, which thereafter becomes responsible for the automatic generative process, allowing further refinement and automatic code generation. The ProjectIT-RSL follows the

emerging DSL approach, supporting the ability to define new languages to better tackle the problem at hands, hence raising developers' abstraction work level.

Despite the success of this work, it's never inconvenient to emphasize the innovative objectives and their associated risks. The fundamental premises adopted, that the author took for granted with the purpose of achieving higher levels of functionality, are, themselves, based on research projects from which there are few real proof of concept tools available. The overall NLP techniques are also themselves still under developing and research, and most of them can't be broadly applied, since they are conceived for specific domain areas, namely Artificial Intelligence (AI) expert systems.

8.1.2 Desirable Features Not Implemented

There was a minor set of desirable features that were not implemented due to the tight schedule available for late implementation tasks, namely: (1) a text editor hyperlink navigation mechanism (through CTRL+CLICK key binding) from requirements textual model references to the respective UML2 models, providing an enhanced traceability between related artifacts (requirements and the respective UML2 models), thus contributing to a deeper integration between ProjectIT-Studio/Requirements and ProjectIT-Studio/UML Modeler; (2) Fuzzy Match Parsing mechanism refactoring, providing a clever way to implicitly extract node's pragmatic information based only on the transformation from the Template part to the respective TS rule's Substitution part; (3) one of the typical text editors' "top ten" features previously analyzed, the *Find&Replace* operation, since it was lately detected that there were deep semantic bugs, and hence hard to identify, in the base platform – despite of this time consuming platform's bugs, which strongly affected the planned schedule, it's impressive the Eclipse.NET stability when considering ratio between the complexity/dimension of this framework and the time frame for converting it; and (4) the integration of the Jena .NET port with external DIG reasoners like Pellet (the reader may consult Chapter 5 and Appendix E for further information).

8.1.3 Development Process Improvement Suggestions

Considering all the developed work, there were some issues in the software development process and the respective methodology that could be improved, namely: (1) although it was

worthy, the research phase was too exhaustive and, consequently, it took too long to accomplish, taking some time from the implementation phase; (2) there could have been more brainstorms for validation of the underlying concepts and theories discovered; (3) despite being necessary and rewarding, there were too many paperwork elaboration tasks; and finally (4) the main problem to solve (natural language processing, which is still an immature field of Artificial Intelligence) was underestimated and, additionally with the disruptive adopted approach, there was an over dimensioning problem during the implementation phase. However, the latter was a risk to be taken to achieve more ambitious results in the near future.

8.2 Conclusions

This work consisted in the design and implementation of a CASE tool for supporting the early activities of the Software Development Process, namely the ones covered the Requirements Engineering, such as requirements specification and validation. The adopted approach was based on the identification and capture of common natural language linguistic patterns (strongly focused on the specification of interactive systems), which were used to define a requirements specification language that can be further used to analyze and validate requirements documents. The goal was to combine natural language with enhanced rigor in order to achieve improved quality, reuse and traceability.

However, this graduation thesis proposal consisted in the conversion of a previous functional prototype of ProjectIT-RSL to the new adopted common software development workbench, the Eclipse.NET project. However, after the initially undertaken research about Natural Language Processing (NLP) and some older and more mature projects (with available and working tools) with similar goals with this work, it was clear that the time was propitious (the supporting tool was going to be build from scratch) to adopt another approach to correct some problems detected with the first implementation of ProjectIT-RSL prototype. A decision had to be made, and the adopted solution was to innovate by incorporating some interesting ideas from the above mentioned NLP projects specialized in capturing the underlying system's models from their requirements specifications. Despite the inherent risk of this new course of actions, it was required due to the limitations of the previous approach and the long term benefits and promising results that can be achieved using a more flexible tool. Therefore, with the broadening of the initial work's proposal, the project was oriented towards new research

areas, passing from a simple programming language endowed with syntactic sugar for resembling natural language, to a more general approach based on natural language processing NLP, gifted with robust and flexible parsing mechanisms and a rule based extensibility mechanism. By this way, the tool offers not only a single fixed requirements specification language, but also a generic framework to specify DSLs.

Since the majority of knowledge-based applications also include an additional reasoning component, the inference-engine, which enables further knowledge manipulation operations such as implicit knowledge extraction, querying, and validation, it was decided to incorporate one in this project. The adoption of an ontology-based approach for knowledge manipulation and querying, besides the enormous advantages in terms of knowledge interchange through heterogeneous environments, brought the possibility to somewhat reduce the gap between the wider software engineering population position and the disregarded, or even unknown, potential of AI techniques. These AI techniques will certainly enhance the potential of common software engineering processes and techniques for specific problems by providing a platform for building intelligent information systems, such as developing CASE tools like this one, for supporting the free-form textual requirements specification activity of non-technical stakeholders by using natural language processing (NLP) techniques.

Besides the development component, this work involved several academic initiatives, such as article submissions to renowned conferences and participation in those same conferences as a speaker, for presenting the ProjectIT-Requirements project and the progress and efforts that have been made during the last year. This complementary facet was truly rewarding in terms of academic projection, providing a good insight of the tasks involved in the research field. It was fruitful and noteworthy the fact that four papers submitted to notorious conferences such as CISTI 2006 [Videira2006a], Euromicro 2006 [Videira2006b], ICSOFT 2006 [Videira2006c], and IVNET 2006 [Silva2006]) were accepted (all of them presenting low acceptance rates). Moreover, this document's author was chosen as the conference speaker for presenting the first three articles, thus attending in a single year to one Iberian Conference ([CISTI2006]) and two International Conferences ([Euromicro2006] and [ICSOFT2006]). The relevance of these activities was a driver to the final decision of the author's ingress in a

PhD degree, for the continuation of his studies in the scope of this project but following a new iteration of the original ProjectIT approach, called MetaIT.

The most regretful issue of this project is the short time-frame to accomplish this ambitious project, particularly due to the professional parallel activities and vicissitudes of the software development process that usually compromise the planning and features deliverance of any IT software product.

8.3 Future Work (DEIC's PhD Scope)

This project will be continued in the next year, under the scope of this document's author's PhD work plan. The main goal of this new project's iteration it to enhance the ProjectIT-RSL language's reuse and extensibility mechanisms with the respective tool's support and to improve the integration with ProjectIT-MDD component to endorse the ProjectIT-Studio tool with higher-level abstractions manipulation features. This new approach, designated as MetaIT, will introduce a development process based on metamodeling and transformations between models, to enhance the tool's configurability and flexibility by supporting a wider range of stakeholders, allowing each one of them to use the most adequate language according to the domain specific expertise at hand. The reader can consult the Appendix J to gain detailed information on the three main topics of research of this PhD work plan, namely: (1) ProjectIT-RSL Reuse Mechanism; (2) ProjectIT-RSL Extensibility Mechanism; and (3) ProjectIT-Studio/Requirements Case Study / End User Validation.

9 REFERENCES

9.1 Bibliographic References

[Abbot1983] Abbot, R., "Program design by informal English description", Communications of the ACM, 16(11), pp. 882-894, 1983.

[Allen1995] Allen, J., "Natural Language Understanding", Addison Wesley, 2nd ed., 1995.

[Ambriola2000a] Ambriola, V., Gervasi, V., "Process metrics for requirements analysis", in Proc. of the 7th European Workshop on Software Process Technology, 2000.

[Ambriola2000b] Ambriola, V., Gervasi, V., "Supporting multiple views on requirements", in Proc. of the 6th Maghrebian Conference on Computer Sciences, 2000.

[Ambriola2003] Ambriola, V., Gervasi, V., "The Circe approach to the systematic analysis of NL requirements", Technical Report TR-03-05, University of Pisa, Informatics' Department, 2003.

[AT&T2004] AT&T, "OWL Full and UML 2.0 Compared", in White Papers, 2004.

[Beck1999] Beck, K., "Extreme Programming Explained: Embracing Change", Addison Wesley, October 1999.

[Carmo2005] Carmo, J., Videira, C, Silva, A., "Using Visual Studio Extensibility Mechanisms for Requirements Specification", 1st Conference on Innovative Views on .NET Technologies, Porto, June 2005.

[Cockburn2000] Cockburn, A., "Writing Effective Use Cases", Addison-Wesley Professional, 2000.

[Cockburn2002] Cockburn, A., et al, "Patterns for Effective Use Cases", Addison-Wesley Professional, 2002.

[Crespo2001] Crespo, R., "Processadores de Linguagens: da concepção à implementação", IST Press, Lisbon, March of 2001, 2nd edition.

[Crowe2005] Crowe, M., "Compiler Writing Tools using C#", edition 4.7, June of 2005.

[Cibulski2001] Cybulski, J., "Application of Software Reuse Methods to Requirements Elicitation from Informal Requirements Texts", PhD Thesis, School of Computer Science and Computer Engineering, La Trobe University, 2001.

[Djuric2004] Djuric, D., "MDA-based Ontology Infrastructure", International Journal on Computer Science and Information Systems, 2004.

[Fowler2003] Fowler, M., et al, "Patterns of Enterprise Application Architecture", Addison-Wesley, 2003.

[Fuchs1996a] Fuchs, N., Schwitter, R., "AttemptoControlled English (ACE)", CLAW 96, First International Workshop on Controlled Language Applications, University of Leuven, Belgium, March 1996.

[Fuchs1996b] Fuchs, N., Schwitter, R., "Attempto - from specifications in controlled natural language towards executable specifications", in GI EMISA Workshop, 1996.

[Fuchs1998] Fuchs, N., Schwertel, U., Schwitter, R., "Attempto Controlled English - Not Just Another Logic Specification Language", in Eighth International Workshop LOPSTR'98, UK, 1998.

[Schwitter1998] Schwitter, R., Fuchs, E., Schwertel, U., "Attempto — Controlled English (ACE) for Software Specifications", Poster presentation by U. Schwertel at CLAW 98, Second International Workshop on Controlled Language Applications, Language Technologies Institute, Carnegie Mellon University, Pittsburgh, May 1998.

[Gamma1995] Gamma, E., et al, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[Gamma2003] Gamma, E., and Beck, K., "Contributing to Eclipse: Principles, Patterns, and Plug-Ins", Addison-Wesley, 2003.

[Jones1990] Jones, C., "Systematic Software Development Using VDM", Prentice Hall, New Jersey, USA, 1990.

[Johnson 1991] Johnson, W., Feather, M., "Using evolution transformation to construct specifications", in Automatic Software Design, The MIT Press, 1991.

[Jurafsky2000] Jurafsky, D., Martin, J., "SPEECH and LANGUAGE PROCESSING: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition", Prentice-Hall, 2000.

[Kleppe2003] Kleppe, A., Warmer, J., Bast, W., (2003) "MDA Explained: Architecture: Practice and Promise", Addison-Wesley, 2003.

[Mich1999] Mich, L., Garigliano, R., "The NL-OOPS Project: OO Modeling using the NLPS LOLITA", in CANLIS, 1999.

[Mich2002] Mich, L., Garigliano, R., "NL-OOPS: A Requirements Analysis tool based on Natural Language Processing", in Proc. 3rd Int. Conf. On Data Mining 2002.

[OMG2003] Object Management Group, "MDA Guide Version 1.0.1", 2003.

[OMG2005a] Object Management Group, "Unified Modeling Language: Superstructure – Specification Version 2.0", August 2005.

[OMG2005b] Object Management Group, "MOF 2.0/XMI Mapping Specification, v2.1", September 2005.

[OMG2005c] Object Management Group, "MOF QVT Final Adopted Specification", November 2005.

[OMG2006] Object Management Group, "Meta Object Facility (MOF) Core Specification", January 2006.

[Saraiva2005a] Saraiva, J., and Silva, J., "Eclipse.NET: An Integration Platform for ProjectIT-Studio. Em Proceedings of the First International Conference of Innovative Views of .NET Technologies". Instituto Superior de Engenharia do Porto, June 2005.

[Saraiva2005b] Saraiva, J., "Desenvolvimento Automático de Sistemas – Relatório TFC", IST/UTL, 2005.

[Silva2001] Silva, A., and Videira, C., UML, "Metodologias e Ferramentas CASE", Centro Atlântico, Portugal, 2001.

[Silva2003] Silva, A., "The XIS Approach and Principles. In Proceedings of the 29th EUROMICRO Conference". IEEE Computer Society, September 2003.

[Silva2004] Silva, A., "O Programa de Investigação ProjectIT". INESC-ID & Instituto Superior Técnico, October 2004. (also available at http://berlin.inesc-id.pt/alb/uploads/1/193/pit-white-paper-v1.0.pdf)

[Silva2005a] Silva, A., "Programa de I&D ProjectIT". INESC-ID & Instituto Superior Técnico, April 2005. (also available at http://berlin.inesc-id.pt/alb/uploads/1/438/pit-status-2005-v0.1.pdf)

[Silva2005b] Silva, A., Videira, C., "UML, Metodologias e Ferramentas CASE – 2ª Edição – Volume 1", Centro Atlântico, Portugal, 2005.

[Silva2006] Silva, A., Videira, C., Saraiva, J., Ferreira, D., Silva, R., "The ProjectIT-Studio, an integrated environment for the development of information systems", in IVNET'06, LNCS, Springer.

[Spivey1992] Spivey, J., "The Z Notation - A Reference Manual", Prentice-Hall, New Jersey, 1992.

[Videira2004a] Videira, V., Silva, A., "A broad vision of the ProjectIT-Requirements, a new approach for Requirements Engineering", in Actas da 5ª Conferência da Associação Portuguesa de Sistemas de Informação, APSI 2004.

[Videira2004b] Videira, V., Silva, A., "ProjectIT-Requirements, a Formal and User-oriented Approach to Requirements Specification", Proc. of the JIISIC 2004 Conference.

[Videira2004c] Videira, V., Carmo, J., Silva, A., "The ProjectIT-RSL Language Overview", in UML Modeling Language and Applications: UML 2004 Satellite Activities, 2004.

[Videira2006a] Videira, C., Ferreira, D., Silva, A., "Patterns and parsing techniques for requirements specification", Proc. of CISTI 2006 Conference.

[Videira2006b] Videira, C., Ferreira, D., Silva, A., "A linguistic patterns approach for requirements specification", Proc. of EUROMICRO 2006 Conference.

[Videira2006c] Videira, C., Ferreira, D., Silva, A., "Using linguistic patterns for improving requirements specifications", Proc. of ICSOFT 2006 Conference.

9.2 Internet References

[Ambler2006] Ambler, S., "Agile Unified Process Overview". June 2006.
http://www.ambysoft.com/unifiedprocess/agileUP.html

[Brill2006] Brill Tagger
http://en.wikipedia.org/wiki/Brill_Tagger

[CIRCE2003] CIRCE Project, 2003.
http://circe.di.unipi.it/eclipse/

[CISTI2006] 1st Iberian Conference on Information Systems and Technologies, June 2006.
http://www.est.ipca.pt/cisti/

[DSMForum] DSM Forum: Domain-Specific Modeling
http://www.dsmforum.org

[Eclipse2006] Eclipse Java Development Tools (JDT) Subproject
http://www.eclipse.org/jdt/

[Euromicro2006] 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), August 2006.
http://www.sea.uni-linz.ac.at/euromicro2006/

[MartinFowler] Martin Fowler
http://www.martinfowler.com

[HP2003] Ho, E., Creating a text-based editor for Eclipse 2.1, 2003.
http://devresource.hp.com/drc/technical_white_papers/eclipeditor/index.jsp

[IBM2003] IBM Corporation and others, "Using the Plug-in Development Environment", 2003.
http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html

[ICSOFT2006] 1st International Conference on Software and Data Technologies, September 2006.
http://www.icsoft.org/

[INCOSE2005] INCOSE Requirements Management Tools Survey, 2005.
www.paper-review.com/tools/rms/read.php

[Jeffries2001] Jeffries R., "What is Extreme Programming", August 2001.
http://www.xprogramming.com/xpmag/whatisxp.htm

[Jena2005] HOWTO use Jena with an external DIG reasoner
http://jena.sourceforge.net/how-to/dig-reasoner.html

[McGuinness2004] McGuinness, D., Harmelen, F.,"OWL Web Ontology Language Overview", 2004.
http://www.w3.org/TR/owl-features/

[Nielsen2006] Nielsen, J., "Ten Usability Heuristics", 2006.
http://www.useit.com/papers/heuristic/heuristic_list.html

[Pellet2003] Mindswap Pellet OWL Reasoner
http://www.mindswap.org/2003/pellet/

[Treebank1999] Treebank, "The Penn Treebank Project", 1999.
http://www.cis.upenn.edu/~treebank/

[W3-Corpora1998] W3-Corpora Project, "The Brown Corpus", 1998.
http://clwww.essex.ac.uk/w3c/corpus_ling/content/corpora/list/private/brown/brown.html