ProjectLingo:
Rigorous Specification of Project Plans

João Francisco Vieira Gonçalves

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Examination Committee

Chairperson: Prof. Ana Teresa Correia de Freitas
Supervisor: Prof. Alberto Manuel Rodrigues da Silva
Members of the Committee: Prof. Maria do Rosário Bernardo Ponces de Carvalho

October 2017
Acknowledgments

This section is dedicated to all that made this possible. Firstly, I would like to thank my supervisor, Professor Alberto Rodrigues da Silva for his guidance and help during whole process. His encouragement and fast responding to questions, as well as his availability to last hour meetings were really important factors. Furthermore, his expertise in Project Management and about the used technologies were essential for the success of this project.

To my colleagues Miguel Conceição and Pedro Cecílio, big thanks for the comradeship and help during the last months.

Lastly, and most importantly, a special and emotional “thank you” to my father, Albertino, and my mother, Luz, for believing in me and providing me all the conditions for the good execution of this work. I promise that I will pay you for all the time I have not spent with you, and make you very proud of me in the future. I also hope we can enjoy some holidays together next summer.
Abstract

Project plans are arguably the most important artefact produced during the execution of a project. Unrealistic time and resource estimates can lead to several negative outcomes, including project failure. Since these artefacts are specified using natural language, and without automation processes, inconsistency, incompleteness, and ambiguity errors emerge. Therefore, it is of the utmost importance to improve the quality of these documents by reducing the amount of these errors, as well as automating the validation part of the project plan. Tools currently available do not comprise the entire set of aspects of project plans, but rather in only a subset of them (e.g. WBS or Charter) and do not focus on the quality and rigorousness of the artefact.

Thus, a tool that could cover all the aspects of a project plan and provide mechanisms for automatically validating parts of it, and additionally supply interoperability features between multiple formats instead of only one, would contribute for better project plan specifications with less errors to be fixed in later stages.

This dissertation describes the ProjectLingo approach, whose goal is exactly address the abovementioned problems by providing a set of tools and mechanisms to accomplish it. A new language for specifying those artefacts, PSL, was defined, and an IDE to support it, ProjectLingo-Studio, was developed. The language is based on an Excel template that can also be used for specifying project plans. The implemented validations are automatically handled by the IDE that also supports features for importing and/or exporting MS-Word and MS-Project files.

Keywords: PSL, Model-Driven Engineering, Domain-Specific Languages, Project Management
Resumo

Planos de projeto são indiscutivelmente o artefacto mais importante que é produzido durante a execução de um projeto. Estimativas irreais de tempo e recursos podem levar a diversos resultados negativos, inclusive à falha do projeto. Uma vez que estes planos são especificados com recurso a linguagens naturais, e sem automação de processos, terão erros relacionados com inconsistência, incompleiude e ambiguidade. Portanto, melhorar a qualidade destes documentos através da eliminação destes erros é extremamente importante. Atualmente as ferramentas que existem não possibilitam a especificação completa destes planos, mas apenas de alguns aspetos (exemplo, WBS ou Charter), nem se focam na qualidade e rigor dos mesmos.

Sendo assim, uma ferramenta que englobasse a maioria dos aspetos do plano e disponibilizasse mecanismos de validação automática para partes dele, assim como interoperabilidade com outros formatos, iria contribuir para especificações com maior qualidade e menos erros.

Esta dissertação descreve a iniciativa ProjectLingo, cujo objetivo é exatamente endereçar os problemas identificados através da disponibilização de ferramentas e mecanismos. Para isso, uma nova linguagem para especificar planos, PSL, foi definida, e um IDE para suportar esta linguagem, ProjectLingo-Studio, foi desenvolvido. A linguagem é baseada num template de Excel que também pode ser diretamente usado para especificar os planos de projeto. As validações implementadas são automaticamente verificadas pelo IDE que também suporta outros formatos, Word e Project, através de mecanismos de importação e exportação.

**Palavras-chave:** PSL, Engenharia Dirigida por Modelos, Linguagens de Domínio Específico, Gestão de Projetos
# Table of Contents

Acknowledgments

Abstract

Resumo

Table of Contents

List of Figures

List of Tables

List of Acronyms

1. Introduction
   1.1. Context
   1.2. Objectives
   1.3. Solution
   1.4. Work Schedule
   1.5. Structure

2. Background
   2.1. Model-Driven Engineering (MDE)
       2.1.1. GP(M)L vs. DS(M)L
       2.1.2. M2T vs. M2M
       2.1.3. MDE Overview
       2.1.4. From Abstract to Concrete MD Approaches
   2.2. Domain-Specific Languages (DSLs)
   2.3. Project Management (PM)
       2.3.1. Waterfall vs. Agile
       2.3.2. Standards
       2.3.3. PMBOK
       2.3.4. PRINCE2
       2.3.5. ICB

3. Research Context
   3.1. RSLingo (Initial Approach)
3.2. RSLingo (Current Approach) .................................................................................. 23
  3.2.1. RSL Excel Template ......................................................................................... 24
  3.2.2. RSLingo-Studio ......................................................................................... 25
3.3. REBox/ITBox ...................................................................................................... 26

4. Technologies ........................................................................................................... 29
  4.1. Xtext.................................................................................................................. 29
  4.2. Xtend .................................................................................................................. 33
  4.3. Apache POI ......................................................................................................... 35
  4.4. MPXJ (Microsoft Project Exchange in Java) ....................................................... 36
  4.5. VBA (Visual Basic for Applications) ................................................................ 37

5. ProjectLingo Approach .......................................................................................... 39
  5.1. Project Specification Language (PSL) ................................................................. 39
       5.1.1. Packages ..................................................................................................... 41
       5.1.2. Charter ......................................................................................................... 42
       5.1.3. Scope ............................................................................................................ 42
       5.1.4. Time ............................................................................................................. 43
       5.1.5. HR ............................................................................................................... 43
       5.1.6. Communication ......................................................................................... 43
       5.1.7. Cost .............................................................................................................. 44
       5.1.8. Risk ............................................................................................................. 44
       5.1.9. Views .......................................................................................................... 45
  5.2. ProjectLingo-Studio ......................................................................................... 49
  5.3. Templates .......................................................................................................... 51
  5.4. Validation .......................................................................................................... 52
  5.5. IDE Customization ............................................................................................ 55
       5.5.1. Syntax Highlighting ....................................................................................... 55
       5.5.2. Outline View & Labeling ............................................................................. 57
       5.5.3. Proposal Provider ....................................................................................... 59

6. Evaluation ............................................................................................................... 61
  6.1. Workflows .......................................................................................................... 62
       6.1.1. Arduous ....................................................................................................... 62
       6.1.2. Partial .......................................................................................................... 62
       6.1.3. Complete .................................................................................................... 63
       6.1.4. Recommended ............................................................................................. 63

7. Conclusion .............................................................................................................. 67
7.1. Future Work.................................................................................................................................................... 68

8. References....................................................................................................................................................... 72

Appendix A........................................................................................................................................................... 76
  PSL Excel Template........................................................................................................................................... 76

Appendix B........................................................................................................................................................... 83
  PSL Word Template........................................................................................................................................... 83

Appendix C........................................................................................................................................................... 86
  PSL Data Types ................................................................................................................................................. 86

Appendix D........................................................................................................................................................... 88
  Validations......................................................................................................................................................... 88
List of Figures

Figure 1. Overview of the ProjectLingo Approach ................................................................. 4
Figure 2. Model-Driven Engineering Overview ......................................................................... 11
Figure 3. Definition phase of RSLingo at process-level (5) .................................................. 22
Figure 4. Overview of the RSLingo approach: usage at project-level (5) ............................ 23
Figure 5. Example of Stakeholders sheet of the RSL Excel Template .................................. 25
Figure 6. PSL Model as a Set of Packages .............................................................................. 41
Figure 7. Diagram for PackageCharter rule ........................................................................... 42
Figure 8. Diagram for PackageScope rule ............................................................................. 42
Figure 9. Diagram for PackageTime rule ............................................................................... 43
Figure 10. Diagram for PackageHR rule ............................................................................... 43
Figure 11. Diagram for PackageCommunication rule .......................................................... 44
Figure 12. Diagram for PackageCost rule .............................................................................. 44
Figure 13. Diagram for PackageRisk rule ............................................................................. 44
Figure 14. Diagram for PackageQuality rule ......................................................................... 45
Figure 15. ProjectLingo-Studio ............................................................................................ 49
Figure 16. Hierarchy Cycle on Stakeholder's 'partOf' Relation ............................................. 53
Figure 17. Crash-causing Hierarchy Cycle Scenario .............................................................. 54
Figure 18. Xtext's Standard Rendering vs. PSL Custom Syntax Highlighting ....................... 56
Figure 19. Outline View before (right above) and after (right below) customization for the code snippet on the left .................................................................................. 58
Figure 20. Content Assist before and after .......................................................................... 59
Figure 21. Workflow 1: From a PSL specification to an Excel or Word file ......................... 62
Figure 22. Workflow 2: From Project to Excel and Word ......................................................... 63
Figure 23. Workflow 3: From Excel to validated Excel or Word ........................................... 63
Figure 24. Workflow 4: From Project and Excel to validated Excel or Word ....................... 64
Figure 25. ProjectLingo's 'Generate' Sub-Menu .................................................................. 68
List of Tables

Table 1. Work Schedule .................................................................................................................................................. 5
Table 2. Some Widely Used Domain-Specific Languages ............................................................................................. 14
Table 3. Summary of Project fragments .......................................................................................................................... 46
Table 4. Summary of WorkComponent fragments ........................................................................................................ 46
Table 5. Summary of Task fragments ............................................................................................................................. 47
Table 6. Summary of Requirement fragments ................................................................................................................ 48
Table 7. Summary of Stakeholder fragments .................................................................................................................. 48
Table 8. PSL elements and correspondent colours and font styles ............................................................................... 56
Table 9. PSL Data Type Rules and its Values .................................................................................................................. 86
Table 10. PSL Custom Validations and respective Error Messages ................................................................................. 88
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM</td>
<td>Project Management</td>
</tr>
<tr>
<td>PMBOK</td>
<td>Project Management Body of Knowledge</td>
</tr>
<tr>
<td>RE</td>
<td>Requirements Engineering</td>
</tr>
<tr>
<td>RS</td>
<td>Requirements Specification</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering</td>
</tr>
<tr>
<td>SE</td>
<td>Software Engineering</td>
</tr>
<tr>
<td>RSL</td>
<td>Requirements Specification Language</td>
</tr>
<tr>
<td>PSL</td>
<td>Project Specification Language</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>MS</td>
<td>Microsoft</td>
</tr>
</tbody>
</table>
1. Introduction

Project Management (PM) is the application of knowledge, skills, tools, and techniques to activities within a project to meet project requirements (1). This is accomplished through the proper application and integration of processes (a total of 47, according to PMBOK (1)), which are in its turn categorized into five process groups: Initiating, Planning, Executing, Monitoring and Controlling, and Closing (1).

The focus of this project is in the second phase: Planning. This phase is particularly important in PM, since the effort spent in planning activities can save countless hours of rework, and costs in subsequent phases that could, ultimately, lead to project failure. In fact, although planning does not guarantee project success, lack of planning will probably guarantee its failure (2). Most importantly than these facts, it is also during this phase that the first versions of the project plan, object of study of this project, are produced.

A project plan, according to PMBOK (1), can be defined as:

“Formal, approved document used to guide both project execution and project control. The primary uses of the project plan are to document planning assumptions and decisions, facilitate communication among project stakeholders, and schedule baselines.”

The main goal of this project was to develop tools and mechanisms to improve the quality of this artefact, with the purpose of mitigating problems related to consistency, completeness, and ambiguity. By assuring this, it is most likely that less changes will be needed to be performed on the initial project plan (developed during the Planning phase), which will, hopefully, also lead to less (or inexistent) budget and schedule deviations.

The rest of this chapter is organized as follows: (i) section 1.1 provides a brief description about the context of this project; (ii) section 1.2 presents the objectives to be achieved in the end of this work; (iii) section 1.3 describes the solution for the previously presented objectives; (iv) section 1.4 describes how the work was divided in the covered months; and (v) section 1.5 describes the structure of the rest of this dissertation.
1.1. Context

This research work has been conducted at the Information and Decision Support Systems Lab\(^1\) of INESC-ID (Instituto de Engenharia de Sistemas e Computadores – Investigação e Desenvolvimento) under the supervision of Professor Alberto Rodrigues da Silva, regarding the Master Degree in Information Systems and Computer Engineering at Instituto Superior Técnico. The work results from an ongoing research initiative in the field of Project Management and is based on two proposals: ProjectIT and RSLingo, which are briefly described below:

- **ProjectIT**\(^3\). The ProjectIT approach was proposed by the Information Systems Group, and states that the emphasis of software development should be on project management, requirements engineering and design activities, so that other activities, such as programming or testing, can be reduced and automated as much as possible. Therefore, it focused on the RE and Model-Driven Engineering (MDE) research fields, with the purpose of improving the productivity and rigor of Software Engineering (SE) activities. As a result of this approach, a wiki that provides an integrated environment to support tasks ranging from requirements specifications, architecture definitions, and system design and modelling, until code generation was proposed\(^4\).

- **RSLingo**\(^5\). The RSLingo initiative states that natural language, despite being the most common and preferred form of representation used within requirements documents, is prone to produce ambiguous and erroneous specification artefacts. RSLingo has changed over time and currently relies on RSL (Requirements Specification Language) whose grammar was defined on the basis of the RSL Excel Template\(^2\). This grammar was defined using Xtext and the Xtend programming language was used for the implementation of additional constraint checks to the grammar. An Eclipse-based IDE tool, RSLingo-Studio, was also developed in to allow users to write and edit RSL files while taking advantage of Eclipse features. Most recently within this approach, ITBox (formerly named REBox), a web-based platform that offers a management system for requirements specifications and project plans, has been developed.

1.2. Objectives

As already mentioned, the main goal of this project was to develop tools and mechanism to improve the quality of project plans by mitigating some common problems that exist on any type of natural language artefacts.

---

1. https://www.inesc-id.pt
To accomplish this, a formal set of objectives was defined and work towards its implementation took place during the executing of this project. Details related to the used technologies and to the developed tools will be thoroughly described in following chapters of this dissertation. A list with the group of objectives is displayed below.

- **DSL Definition.** Definition of a Domain-Specific Language for specification of project plans.

- **IDE Development.** Development of an IDE to support the newly defined language.

- **Automatic Validation Mechanisms Implementation.** To automatically check the correctness of specifications written in the newly defined language on the new IDE.

- **Interoperability Features Implementation.** To allow transformations between formats and therefore different representations for the same project plan specification.

- **Customization of IDE Aspects.** To enhance the user experience and increase the chances of adoption among business and non-business stakeholders.

- **Validation of Results.** Test all implemented features to make sure the system is bug-free and that erroneous input does not make it crash.

### 1.3. Solution

This research proposes ProjectLingo, an initiative whose aim is the achievement of more rigorous specifications of project plans. To reach this goal, a set of tools and mechanisms were implemented or enhanced.

The first step was the definition of the Project Specification Language (PSL), a rigorous and intermediate textual DSL that allows the specification of multiple project-related aspects, such as the WBS, RAM, OBS, Risks, Issues, etc. This language was specified using the Xtext framework and is based on the PSL Excel Template. Both artefacts (PSL and PSL Excel Template) are described in later sections of this document and presented in Appendices A and B, respectively.

After the grammar definition, additional checks to the grammar had to be implemented using the Xtend programming language. These additional checks either could not be implemented directly on the grammar specification or would make this specification a lot more complex. Some of these validations were to make sure that PSL and PSL Excel Template were completely compliant, other were related to obliterating invalid scenarios, and so on. This topic is approached into more detail in section 5.4, and a complete list of the implement validations can be seen in Appendix D.
Then, it was necessary to develop an IDE where users could create and edit files of the new format. To this end, the ProjectLingo-Studio was developed. It is an Eclipse-based IDE that additionally to allowing the creation and editing of PSL files, also verifies the correctness of PSL specifications (by issuing errors when some validations fail) and provides import and export features between PSL and other Microsoft formats, namely Excel, Word and Project. To deal with Excel and Word files, Apache POI was used, while MPXJ was chosen for Project files. These features are available on the Main Menu bar of ProjectLingo-Studio, and even have shortcuts assigned to them.

An overview of this solution is shown in Figure 1.

As can be observed, it suggests that the Project Manager is responsible for creating a Project Plan specification on ProjectLingo-Studio IDE, using the PSL language which is based on the PSL Excel Template. Alternatively, the Project Manager can also fill the PSL Excel Template sheets, import it using Studio (that automatically translates it to PSL and checks its correctness), and then export it either back to Excel or to Word. This output is a validated Project Plan that can be delivered to the Business Stakeholder for further approval. If the Project Manager is only interested in the WBS, he can define it using MS-Project instead and then validate it and export it to other formats.
1.4. Work Schedule

This work took place between February and October of 2017, and in Table 1 is detailed what was achieved in each of these months.

<table>
<thead>
<tr>
<th>Month</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>February</td>
<td>- Learning how to use Xtext and Xtend technologies.</td>
</tr>
<tr>
<td>March</td>
<td>- Implementation of PSL’s initial version.</td>
</tr>
<tr>
<td></td>
<td>- Learning about Apache POI framework.</td>
</tr>
<tr>
<td>April</td>
<td>- Implementation of Export and Import features for Excel.</td>
</tr>
<tr>
<td></td>
<td>- Development of ProjectLingo-Studio’s menus</td>
</tr>
<tr>
<td>May</td>
<td>- Complete implementation of PSL constructs for all PSL Excel Template concerns.</td>
</tr>
<tr>
<td></td>
<td>- Improvement of ProjectLingo-Studio interface.</td>
</tr>
<tr>
<td></td>
<td>- Implementation of Export and Import features for Excel’s missing concerns.</td>
</tr>
<tr>
<td></td>
<td>- Major improvements to the PSL Excel Template.</td>
</tr>
<tr>
<td>June</td>
<td>- Implementation of Export to Word features.</td>
</tr>
<tr>
<td></td>
<td>- Creation of PSL Word Template.</td>
</tr>
<tr>
<td></td>
<td>- Learn about MPXJ framework.</td>
</tr>
<tr>
<td></td>
<td>- Implementation of validations to the PSL language.</td>
</tr>
<tr>
<td>July</td>
<td>- Implementation of Import from MS-Project feature.</td>
</tr>
<tr>
<td></td>
<td>- Customization of PSL’s syntax highlighting.</td>
</tr>
<tr>
<td>August</td>
<td>- Customization of ProjectLingo-Studio’s outline view, labeling and proposal provider.</td>
</tr>
<tr>
<td>September</td>
<td>- Thesis writing.</td>
</tr>
<tr>
<td>October</td>
<td>- Thesis writing.</td>
</tr>
</tbody>
</table>

1.5. Structure

The remainder of this dissertation is organized as follows:

- **Chapter 2.** This chapter provides an overview of the main concepts and background that underlies this research, particularly focusing on three main topics: Model-Driven Engineering, Domain-Specific Languages and Project Management.
- **Chapter 3.** This chapter states the research context of this dissertation, namely its antecedent approaches and brief explanations to tools developed during them.
- **Chapter 4.** This chapter provides descriptions for the technologies used for the execution of this work, particularly four of major importance: Xtext, Xtend, Apache POI and MPXJ; and Visual Basic for Applications.
- **Chapter 5.** This chapter describes the ProjectLingo approach, its artefacts, features and tools, and validations and customizations performed.
Chapter 6. This chapter describes how the developed work was validated and evaluated, and presents the different workflows that can be used for further testing.

Chapter 7. Lastly, this chapter presents the main conclusions of this work along with suggestions for future development.

Additionally, there are the following appendixes:

- **Appendix A.** Contains screenshots of all the sheets from PSL Excel Template.
- **Appendix B.** Contains screenshots of the cover and back page of the PSL Word Template, as well as a screenshot from one of the middle pages for example purposes.
- **Appendix C.** Contains a table with two columns: one with the names of the different data type rules of PSL and the other with the possible values for that certain rule.
- **Appendix D.** Contains a table with two columns: one with the description of each of the implemented validations to PSL and the other with the correspondent error message.
2. Background

In this chapter, three main concepts that serve as a background for this work will be thoroughly described and examples will be provided. These concepts are: Model-Driven Engineering, Domain-Specific Languages and Project Management. ProjectLingo initiative is within the scope of these, although, as far as known from the literature analysis, there is not any initiative that can be compared to it or that addresses the same objectives or provides the same features.

2.1. Model-Driven Engineering (MDE)

One of the hardest challenges during project iterations, is reaching a common vision and knowledge among technical and non-technical stakeholders. Models allow the surpassing of this challenge, facilitating and promoting the communication between them (6). In addition to these qualities, models also make the project planning more effective and efficient while providing more appropriate views of the system (6). For the above identified reasons, MDE is within the scope of this project.

During the last decades, several techniques and modelling languages have been proposed to support the design and the development of complex software systems (6). Various of these were defined in the context of methodological approaches, such as object-oriented or unified methodologies, with the goal of allowing a common and coherent vision of the system under study and, consequently, of easing the communication among stakeholders (7).

However, methodologies considering models not only as documentation artefacts, but instead first-class citizens in the software engineering process, have emerged during this last decade, meaning that these models might be used in all engineering branches and for all application domains (6). These proposals – such as Model-Driven Architecture (MDA) (8), Software Factories (6), or, most recently, DSL Engineering (9) – have been classified more generically as MDE or by related terms such as Model-Based Engineering (MBE), Model-Driven Development (MDD), Model-Driven Software Development (MDSD) (10) (11), or even Model-Based Testing (MBT) (12).
In addition to the benefits provided by earlier proposed methodologies, this new trend of approaches would also allow the creation or automatic execution of software systems based on models using complex techniques such as meta-modelling, model transformation, code generation, or model interpretation (6).

2.1.1. GP(M)L vs. DS(M)L

In terms of the proposed modelling languages – which are defined by metamodels and are sets of all possible models that are in accordance with its metamodel – they might be classified in two categories: General-Purpose (GPML) or Domain-Specific Modelling Languages (DS(M)L) (13) (14) (15) (16) (17). GPMLs (such as UML or SysML) have a great number of generic constructs. On the other hand, DSLs use fewer constructs or concepts close to its application domain.

More importantly, some authors argue that DSLs – described in more detail in section 2.2 – can improve productivity, reliability, maintainability, and portability (13) (18). However, the usage of a DSL can also raise problems such as the cost of learning, and of implementing and maintaining a new language (as well as its support tools), proving that choosing to adopt them rather than a more widely accepted GPML is not a straightforward decision and should be carefully considered.

2.1.2. M2T vs. M2M

Apart from the modelling language, there are also two main types of transformations that tend to be considered when using an MDE approach: Model-to-Text (M2T) and Model-to-Model (M2M) (6). The former transformations generate or produce software artifacts (source code, XML, and other text files) from models, by, most commonly, using code generation techniques (6). The latter, on the contrary, allow the translation of models into another set of models typically more specific or that satisfy specific needs for different stakeholders (6).

2.1.3. MDE Overview

Figure 2 displays an overview of MDE. The process is the following: a functional specification (e.g. an Xtext grammar) specified using a DSL, in its turn specified in a meta language (e.g. Xtext), is translated into application code (e.g. Java code) by a generator (e.g. Xtend class). This application code uses a framework (e.g. Eclipse) to execute, and both (application and framework) conform to some architecture.
It is also worth mentioning that, to be effectively used in the context of MDE, models must be defined in a consistent and rigorous way. With this end, there are sets of features that modelling tools should provide – such as model analysis, validation and simulation (as discussed in (19)) – to assure a certain quality level (6).

2.1.4. From Abstract to Concrete MD Approaches

According to (6), MD approaches can be divided in three layers: top, middle and bottom.

On the top, and most abstract layer, there is MDE, MDD and MBT which have already been introduced in the beginning of this chapter.

Middle-level approaches (commonly known as “language workbenches”), constitute the middle layer and are usually supported by complex tools referred as “MD MetaTools” (6). Most of these tools help users on the definition of DS(M)Ls, by providing features such as specific editors, model validation, model transformation, or others (6). Xtext and EMF (Eclipse Modeling Framework) are textual and graphical, respectively, middle-level approaches that are within the scope of this project and that were used during its execution.

Finally, the bottom layer comprises concrete-level approaches. XIS-Mobile, WebService Software Factory, SilabMDD, PGBT, or, the more familiar, RSLingo’s RSL are examples of approaches included in this layer. Given that XIS-Mobile uses EMF, and that SilabMDD encloses a textual DSL specification, both aspects of interest for this project, a brief description of these initiatives will be presented below.

- **XIS-Mobile** (20). Developed within the context of MDDLingo, a “cousin” approach of RSLingo, with the purpose of increasing the productivity of developing cross-platform mobile applications.

---

3 http://www.theenterprisearchitect.eu/blog/2009/02/18/model-drivenengineering-tools-compared-on-user-activities
It has a supporting framework which intends to generate source code for multiple platforms from a single platform-independent model specification, through M2M and M2T transformations. This framework suggests the development of a mobile application by following four steps: (i) definition of the required views using the Visual Editor, (ii) validation of defined views using the model validator, (iii) generation of the User-Interfaces View models with the model generator, and, finally, (iv) generation of source code through the code generator. Following these steps, only a single specification of the system is used which will in its turn increase the developer’s productivity by avoiding implementation of boilerplate code, and reducing errors.

- **SilabMDD** (21). Particularly focused on the requirements discipline. It is constituted by the SilabReq language, a textual DSL implemented using JetBrains MPS, that allows definition and management of requirements based on rigorous use-cases specification. The goal of SilabMDD is to provide a complete software development workbench (by extending JetBrains MPS) to be used by all types of technical stakeholders, from requirements engineers to developers, and non-technical stakeholders.

### 2.2. Domain-Specific Languages (DSLs)

DSLs are also called application-oriented, special purpose, specialized, task-specific, or application languages (16).

There are also two types of DSLs: internal and external. Internal DSLs rely on the powers of a GPL host (e.g. RSpec or Rake with Ruby as a host). On the other hand, external DSLs have their own syntax instead of being built on top of a language (e.g. CSS) (9).

It is also worth mentioning that DSLs are becoming more and more important in the software engineering field, and that the tools to develop them are being improved as well (9). They trade generality for expressiveness in a limited domain, which means they do not provide features for solving all kinds of problems but offer an easier and faster solution if the domain is covered by a specific DSL (9). By providing notations and constructs tailored towards a certain application domain, they also offer substantial gains in expressiveness and ease of use compared with widely-known GPLs (e.g. Java, C++, and others) for the concrete domain being considered, with corresponding gains in productivity and reduced maintenance costs (16).

DSLs are defined by three main aspects: Abstract Syntax, Concrete Syntax, and Semantics (6) (9).

- **Abstract Syntax.** Describes the structure of the language, and how different primitives can be combined, based on concepts and relationships, and well-formed rules (6) (9).

- **Concrete Syntax.** Describes specific representations of the modelling language, covering textual and/or graphical representations (6) (9).
- **Semantics.** Describes the meaning of the elements defined in the language, as well as the meaning of their connections. Semantics can be of four types: denotational, pragmatic, translational and operational (6) (9).

Even though all GPLs are interchangeable (9), each provides different features tailored to specific tasks at hand and that is the reason why there are multiple and not only one (9). The more specific the tasks get, the bigger is the need to use specialized languages rather than general purpose. For instance, SQL was created due to the necessity of having a language to deal with relational databases that use tables, rows, columns and so on, as their core abstractions.

On the uttermost state of specialization, there are DSLs. A DSL is a language that is optimized for a determined domain, and is, usually, more specific than a GPL (9).

Besides these, there are also other benefits of using DSLs that will now be described.

- **Productivity.** Once the language implementation phase is completed and the execution engine for a particular aspect is ready, work becomes highly efficient and no longer manual. This is of highest importance if a few lines of “DSL code” can replace tons of “GPL code” (9).

- **Quality.** The consistent automation of repetitive work by the execution engine, among other things, are behind the increasing of quality inherent to DSL usage. This occurs because the language adopts a correct-by-construction policy, which means it only allows the construction of correct programs (9).

- **Validation and Verification.** Analyses are much easier to implement, and error messages can use more meaningful wording, since they can use domain concepts. In addition to this, since the domain-specific aspects are uncluttered, domain experts can be involved in manual reviews and validation tasks which is another great advantage compared to common GPLs (9).

- **Data Longevity.** Models, if done right, are independent of specific implementation techniques. They can be transformed into other representations, for example, due to a migration to a new DSL technology. The implementation, however, is specific to an individual tool and therefore not reusable (9).

- **No Overhead.** Code generators can remove the abstractions of a DSL source code and generate efficient and low-overhead code, every time, automatically. This can represent a big advantage in cases where performance is a concern (9).

- **Platform Independent.** DSLs support separation of concerns. That is, concerns expressed in the DSL are independent of target platform. Therefore, it is absolutely feasible to change the execution engine and the target platform, to execute the code on a new platform (9).
Despite all these benefits, usually only the sum of a subset of them is worth the investment in learning and adopting this methodology. Besides that, there is also a set with the same amount of challenges such as the effort of building the DSL, lack of experience, non-involvement of domain experts and, evolution and maintenance restrictions (9).

In terms of the development process of a DSL, it consists of five main steps: (i) Decision, (ii) Analysis, (iii) Design, (iv) Implementation, and (v) Validation. Each of these steps is briefly described below.

- **Decision.** Decide if it is to create a new DSL from scratch or simply reuse an existing one. Non-technical users can and should take part during this phase (16).

- **Analysis.** After the analysis phase, the problem domain should be perfectly identified, the domain knowledge should have been gathered from end-users, and a domain model should have been designed (16).

- **Design.** Definition of the DSL components (Abstract Syntax, Concrete Syntax and Semantics), and resolve which approach should be used to implement the language (16).

- **Implementation.** Develop the abovementioned components (using Xtext, or other similar tools), and the corresponding interpreter/compiler if it is an executable DSL (16).

- **Validation.** Test the language and validate its features with the end-user (16).

Finally, to validate all the arguments presented during this section favoring the adoption of DSLs, a few examples of widely used DSLs are listed on Table 2 as a proof of concept.

<table>
<thead>
<tr>
<th>DSL</th>
<th>Application Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNF</td>
<td>Syntax specification</td>
</tr>
<tr>
<td>Excel</td>
<td>Spreadsheets</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper</td>
</tr>
<tr>
<td>LaTeX</td>
<td>Typesetting</td>
</tr>
<tr>
<td>Make</td>
<td>Software building</td>
</tr>
<tr>
<td>MATLAB</td>
<td>Technical computing</td>
</tr>
<tr>
<td>SQL</td>
<td>Database queries</td>
</tr>
</tbody>
</table>
2.3. Project Management (PM)

Project Management (PM) arrived due to the increase of demanding for complex and sophisticated systems and services and due to the expansion of human knowledge. It is a set of principles, methods and techniques for achieving effective planning of work, and thereby establishing a basis for scheduling, controlling and re-planning in the management of projects. PM provides tools to improve organizations’ ability to plan, organize, implement and control its activities and its human resources and assets.

A project is a non-repetitive activity that usually takes place during a well-defined period, and that normally has time, financial and technical performance goals. Even though PM fundamentals can be applied to any range of projects, from very small to very complex, the provided tools are more appropriate for rather large projects. PM techniques are also applicable for several distinct businesses, such as construction, manufacturing, servicing, software development, etc. since their implementation can have significant results such as cost and time reduction, resources allocation and even better final-product quality.

According to the PMBOK (1), a project that follows PM basics is divided in five phases: (i) Initiating, (ii) Planning, (iii) Executing, (iv) Monitoring & Controlling, and (v) Closing.

- **Initiating.** The goal of this phase is to define the project at a broad level. It is also during this phase that it is decided if the project is feasible and therefore be undertaken. If it given a green light to the project, a project charter or a Project Initiation Document (PID) must be created. The latter should include the purpose and requirements of the project and it is advisable that the used template follows PMBOK guidelines.

- **Planning.** This phase is key for a successful PM implementation and focuses on developing a roadmap that should be followed by everyone involved. It usually begins with setting goals for the project in hands. Most importantly, is during this phase that the scope of the project is defined and the project plan developed by the project manager.

- **Executing.** This phase starts with a kick-off meeting where the teams and individuals are informed about their responsibilities. During this phase, the deliverables are developed and completed and the plan specified in the previous phase can be modified as needed. It also involves status meetings, and production of status report and performance reports.

- **Monitoring & Controlling.** This phase often occurs simultaneously with the previous one. The main goal of this phase is to measure the project performance and progression to ensure everything is aligned with the plan. Key Performance Indicators (KPIs) are used by project managers to determine if the project is on the right track and, if not, it may be needed to adjust schedules and/or resources to avoid major deviations.
- **Closing.** This phase begins when the project is completed. Valuable team members are recognized, contractors hired to work on the project are disengaged, and a closing meeting is often organized by the project manager to evaluate what went well during the project execution and identify possible flaws. Then, the project manager creates a report with a list of goals that were not accomplished during the project, and prepares a final project report.

Despite being applicable to several industries, PM is becoming increasingly specialized and numerous distinct methodologies have emerged to better accommodate the specific project scope in hands. However, the traditional five phases described above still conform with most of these methodologies, especially with the Waterfall methodology.

### 2.3.1. Waterfall vs. Agile

Two of the most prominent methodologies are Waterfall and Agile. Waterfall is the most widely applied and relies on a clearly defined set of linear processes that are strictly conducted according to an established schedule. On the other hand, the Agile methodology relies on rapid and cyclical product deliveries, and on anticipating contingencies during the project execution. These two methodologies will now be described for the software development paradigm.

#### 2.3.1.1. Waterfall

Similarly to construction or manufacturing workflows, this methodology is a sequential design process: after each of the six phases are completed, a new step begins. These phases can have several different names, but one possible set is: (i) Requirement Gathering and Analysis, (ii) System Design, (iii) Implementation, (iv) Testing, (v) Deployment, and (vi) Maintenance.

Considering this sequential nature, once a step has been completed, it is not possible to go back without scratching the project and starting from the beginning. Therefore, since there is less or no room for changes or errors, an extensive plan must be set in the beginning and carefully followed for the rest of the project. Another disadvantage of this methodology is that it heavily relies on the initial requirements specification, and if those change over time, or are faulty, the project is doomed.

Also, testing is only done at the end of the implementation which means that if bugs are written early but discovered later in the process, the amount of affected code will most likely be higher than if those bugs were discovered earlier.

Despite these constraints, there are certain reasons that make this methodology the most widely chosen by companies. For example, since there is an extensive project plan since the beginning, the client knows exactly what to expect and has a deep idea of the size, cost, and timeline for the project.
Additional advantages are related to the immense documentation produced with this methodology, which can lead to improvement in future projects, or to minimal impact in case of an employee turnover.

Thus, this methodology should be used when: (i) there is a clear picture of what the final product should be, and a tuned symphony between the project manager and the client; and (ii) clients are not allowed to change the scope of the project after its beginning.

### 2.3.1.2. Agile

Agile is a methodology that emerged as a solution to the disadvantages of the Waterfall methodology identified in the previous section. Instead of following a sequential design process, it follows an incremental approach.

The development teams start with an oversimplified project design, and then start working on small modules as increments to it. These increments are done in weekly or monthly sprints, and tests are run at the end of each.

One of the commonly identified flaws of this methodology, is its collaborative nature that focuses on principles rather than on the process. This methodology relies on the project manager to avoid at all costs the project from becoming a series of code prints, which will most likely lead to a belated and over budget project. Also, since the initial project lacks an extensive initial plan, the final product can be extremely different than what was initially expected which can affect the relationship with the client if cost or time deviations occur.

However, agile is useful in the sense that changes are easily accepted and made to the initial planning. Since there is a constant contact with the client, feedback about changes to be implemented is almost instantly received and executed during the next sprint(s). This will also ensure that the client accepts the final product considering that he was part of its development. Additionally, due to tests being run after each sprint, bugs are identified and fixed as early as possible contributing to fewer schedule adjustments.

Scrum, Extreme Programming (XP) and Kanban are some examples of widely-used agile frameworks that share much of the same philosophy, characteristics and practices but that differ in terms of terminologies and tactics.

### 2.3.2. Standards

PMBOK, PRINCE2 and ICB are the most recognized standards for project management. The first is mostly used in America, Asia and Africa, while the others are especially accepted in Europe and Australia. It is of utmost importance to establish a common agreement on which standard will be used
with all involved team members before the beginning of the project activities. The structures for each of these standards will be provided in the remainder sub-sections of this section.

### 2.3.3. PMBOK

PMBOK stands for Project Management Body of Knowledge and is a knowledge-based standard published by the Project Management Institute (PMI).


Additionally, it also considers 5 process groups: (i) Initiating, (ii) Planning, (iii) Execution, (iv) Monitoring & Controlling, and (v) Closing. These process groups englobe 47 processes and each consists of a number of Inputs, Tools & Techniques and Outputs.

### 2.3.4. PRINCE2

PRINCE2 stands for Projects in Controlled Environment and is a methodology-based standard published by the Office of Government Commerce (OGC).

It is based on 7 principles that are mandatory when using this methodology, namely: (i) Continued business justification, (ii) Learn from experience, (iii) Defined roles and responsibilities, (iv) Manage by stages, (v) Manage by exception, (vi) Focus on products, and (vii) Tailor to suit the project environment.

PRINCE2 also relies on 7 themes, which are areas of PM that must be continuously addressed throughout the project. These themes are: (i) Business Case, (ii) Organization, (iii) Quality, (iv) Plans, (v) Risk, (vi) Change, and (vii) Progress. For each of these themes, PRINCE2 provides explanations for their purpose, definition, PRINCE2 approach, and Responsibilities based on the roles.

Ultimately, PRINCE2 is also defined on the basis of 7 processes: (i) Starting up a project, (ii) Directing a project, (iii) Initiating a project, (iv) Controlling a stage, (v) Managing product delivery, (vi) Managing a stage boundary, and (vii) Closing a project. Similar to themes, an explanation for the purpose, objective, context, and needed activities is provided for each process.

### 2.3.5. ICB

ICB stands for IPMA Competence Baseline and is considered as a baseline for PM competencies published by the International Project Management Association (IPMA). It has three main competence
elements that together constitute the Eye of Competence. These competence elements are either Technical, Behavioral or Contextual.

Technical competence covers 20 elements, behavioral covers 15 and contextual covers 11. ICB provides distinct descriptions for the most important sections of each of these different competences based on their type. For the sake of brevity, all the elements as well as the description criteria for the most important sections are omitted from this section.
3. Research Context

This chapter provides an overview of the context that underlies this dissertation, namely the RSLingo research initiative, which defines a set of processes and languages towards the definition of more formal and errorless requirements specifications. The work developed during the last years within this approach, has originated a set of languages and other artefacts such as templates that were of major relevance for the work described in this document. Some of these outputs will also be described during this section. Finally, the chapter ends with a brief description of REBox/ITBox which, likewise ProjectLingo, is a ‘son’ approach of RSLingo.

3.1. RSLingo (Initial Approach)

A great deal of RE effort is about effective communication and to achieve this, it is fulcrum that everyone communicates by means of a common language (5).

The RSLingo approach supports the usage of natural language for writing requirements specifications. This approach is not disruptive with regard to traditional RE approaches since despite being particularly focused on the writing process of requirements specifications, it still complies with the common RE process, where other activities such as elicitation and negotiation are indispensable (5).

Natural language provides enough expressiveness and avoids problems related to familiarity or proficiency. These problems can arrive especially due to some approaches that force business stakeholders to learn and use new notations, mainly those with a higher level of formality and that are preferred by computer-savvy stakeholders such as developers (5).

However, the usage of natural language for RE purposes gives rise to requirements specifications with well-known problems such as ambiguity, incompleteness, inconsistency, and incorrectness (4) (22). In addition to the propensity for errors while using this notation, activities related to writing and validating requirements written in natural language are usually also very time consuming and error-prone (5).
In order to circumvent the abovementioned problems, it would be helpful to automate some of the manually performed tasks of requirements specifications for (i) domain analysis, such as identifying relations between relevant concepts; (ii) verification and consistency check, to prevent delays and extra costs due to rework arising from discovered defects in later software development phases; and (iii) transformations, for automatically generate other requirements representations, such as diagrams, or reports (5).

To this end, RSLingo initially proposed an information extraction approach (23), based on linguistic patterns that are frequently used in RE-specific concerns and that could be followed to improve the quality and rigor of requirements specifications written in ad-hoc natural language.

However, the full processing of an ad-hoc natural language text was still too complex to be automatically performed by computers in a general-purpose context (5). To overcome this limitation, RSLingo switched to using some simplifications that could be applied to Natural Language Processing (NLP) techniques when only relevant information was to be extracted from the original text (5).

To capture and translate this filtered information, the RSLingo approach relied on the usage of two languages: RSL-PL (‘PL’ standing for Pattern Language) (24) and RSL-IL (‘IL’ standing for Intermediate Language) (25). The former’s aim was to encode RE-specific linguistic patterns found in the natural language-written requirements specifications (8). The latter, in its turn, could be considered as a DSL for RE, that enabled the formal specification of requirements through requirements representations that followed the previously defined (using RSL-PL) linguistic patterns (9).

Then, RSLingo considered two distinct stages: definition at process-level and usage at project-level. Process-level is depicted in Figure 3 and consisted of the definition (or adaptation) of the linguistic patterns encoded in RSL-PL, and of the mappings between these patterns and the semantically equivalent RSL-IL formal structures (5). The set of these mappings is represented as ‘(RSL-PL) => (RSL-IL)’ and was the main asset to be produced during this stage.

![Figure 3. Definition phase of RSLingo at process-level (5)](image-url)
On the other hand, project-level, as exhibited in Figure 4, consisted on the application of the previously defined concepts and languages, as well as on the usage of a set of tools, during the execution of a specific software project (5).

![Figure 4. Overview of the RSLingo approach: usage at project-level (5)](image)

This stage considered two main roles: the ‘Requirements Engineer’ and the ‘Business Stakeholder’. Both would contribute for the initial requirements specification which would be written in natural language. This initial specification would be accompanied by a ‘Glossary’ that established a common vocabulary for key domain-problem terms. These two artefacts, along with the ‘RSL-PL => RSL-IL Mapping’ produced in the previous stage, and a set of ‘General Lexical Resources’ for supporting disambiguation tasks, were then used by the RSLingo toolset (4) to supply a repository of RSL-IL expressions that represented a formal subset of the original requirements specification. These expressions, could then be used to execute automatic verifications, or transformations to ‘Controlled NL’ requirement specifications or ‘Requirements-derived Models’.

### 3.2. RSLingo (Current Approach)

In the previous section, the initial RSLingo approach was described. This approach was followed until mid-2016 being then replaced by the approach described during this section.

Currently, RSL-PL can still be used but it is not mandatory anymore and RSL-IL evolved into a new language. This language, RSL (Requirements Specification Language), was defined based on the RSL
Excel Template\textsuperscript{4}: an Excel document constituted of multiple sheets each covering an aspect related to requirements and with cross-references between sheets.

RSL is defined in Xtext and uses the Xtend programming language for the implementation of other related aspects, such as constraints checks (validations) to the grammar. To take advantage of this new language, the RSLingo-Studio, an Eclipse-based IDE, was also developed. This IDE works as an external tool and therefore can be downloaded and used by anyone interested. It also provides some features that allow users to import or export between the RSL format and other formats such as .xls (Microsoft Excel) format, by meticulously filling the RSL Excel Template which will be described in the next section.

\subsection*{3.2.1. RSL Excel Template}
RSLingo’s former approach considered the whole process of knowledge extraction and conversion to a more rigorous representation, as a way of helping business stakeholders without deep knowledge about requirements engineering to gain a better understanding on the problem at hand. It also argued that this process would favour the business stakeholders in better comprehending the implications of the natural language-written statements, that represented their requirements.

However, RSLingo’s current approach considers that, for a technical user that is already familiar with most of the concepts of RE, the process-level phase (Figure 3) can be omitted. RSL defines several constructs that are logically arranged into viewpoints according to the RE-specific concerns they address. For instance, the people and organizations that can influence or be affected by the system are represented in the Stakeholders viewpoint. Likewise, the objectives of business stakeholders regarding the expected value from the system are expressed in the Goals viewpoint (26) (27). This multi-view architecture defined in RSL has been replicated to an Excel template, namely the RSL Excel Template\textsuperscript{3}, formerly known as RSL-IL Excel Template or as RSL-IL SRS Excel Template (28).

Finally, a simplified screenshot of the ‘stakeholders’ sheet from the template is shown in Figure 5, with the name of some other sheets on the bottom, filled for a specific example.

\begin{footnotesize}
\begin{itemize}
\item[4] \url{https://www.github.com/RSLingo/RSL-Excel-Template/blob/master/RSLingo-RSL-Excel-Template-v3.3.xlsx}
\end{itemize}
\end{footnotesize}
3.2.2. RSLingo-Studio

RSLingo-Studio is an Eclipse-based IDE whose main feature is a text editor to create and edit RSL files. This editor was developed using the Xtext⁵ framework which allows the definition of a DSL using only a grammar specification. From the grammar specification, the framework will automatically generate the whole language infrastructure and a fully customizable Eclipse plugin containing an editor for the specified DSL with the typical Eclipse features such as syntax highlighting, error checking, or error-completion.

Despite the editor, RSLingo-Studio also has import and export features to support multiple representations of a requirements document. RSL is an intermediate language between the different transformations and therefore any supported imported file is translated to RSL and validated by the editor, and can be further exported back to the same or other format. This multi-transformation approach involves Model-to-Text and Model-to-Model transformations and supports the following formats: Excel, Word, UML, LegacyIS and NewIS. These transformations are described below:

- **T1: Excel-to-RSL.** Implemented in RSLingo-Studio as a “Import from Excel” option from the menu, this transformation requires the source Excel file to comply with the RSL Excel Template to be read and translated to RSL.

---

⁵ https://eclipse.org/Xtext
- **T2: RSL-to-Excel.** Similarly to T1, this is implemented as a “Export to Word” feature from RSLingo-Studio’s menu and uses the RSL Excel Template to generate the target Excel file from the source RSL file.

- **T3: RSL-to-Word.** Likewise T2, this is an Export feature that generates a Word file from a RSL source file and it should also use the RSL Word Template.

- **T4: RSL-to-UML.** This transformation appears on the RSLingo-Studio’s menu as a “Export to UML (PlantUML)” option and generates several UML diagrams based on the PlantUML format. These generated diagrams can be further included and published automatically in a Word format by using the RSL-to-Word transformation from above.

- **T5: LegacyIS-to-RSL.** This transformation was implemented in the scope of the XIS-Reverse approach (29) and is a model-driven reverse engineering approach that includes a tool for automatically extracting RSL specifications from legacy information systems, in particular from their legacy databases. It is of extreme good use when there is no prior or up to date knowledge about the legacy system.

- **T6: RSL-to-NewIS.** This feature will be implemented as a “Generate App…” option and will involve a set of model-to-text transformations that shall generate quasi-complete software business applications from a well-defined and validated RSL specification. It will allow users to select the intended target software architecture and technology to generate different types of software code written in common programming languages (such as Java, C#, JavaScript or SQL), for different types of frameworks (such as ASP.NET, PHP, Angular2, etc).

From the abovementioned transformations, the first four are mainly targeted for technical documentation purposes and, hence, to be especially used by requirements engineers and business analysts. On the other hand, the last two correspond to reverse and forward engineering approaches and therefore are particularly targeted to software developers (27).

### 3.3. REBox/ITBox

REBox is a web-based collaborative platform for managing requirements specification documents and other technical documentation. Users can author, review and validate their requirements written based on RSL’s constructs but using an easy to learn and clean interface (27).

Most recently, REBox has changed its name to ITBox since it now also provides the same features for the artefacts developed during this work, i.e., project plan specifications written based on PSL’s constructs and PSL Excel Template.
4. Technologies

This chapter provides thorough descriptions of all the technologies, frameworks or libraries used for the implementation and customization of the ProjectLingo artefacts (namely the ProjectLingo-Studio, PSL and PSL Excel Template). These technologies are: (i) Xtext, used for specifying the PSL grammar; (ii) Xtend, used to implement additional checks and customizations to PSL; (iii) Apache POI, used to deal with import and export of Excel and Word files; (iv) MPXJ, used to deal with import of Project files; and (v) VBA, used for defining a macro for Excel.

4.1. Xtext

Implementing a new DSL consists on developing a program that can read text written using that DSL, parse it, process it, and then interpret it or generate code in another language. This process may require several phases, depending on the purpose of the DSL, but most of these are typical of all implementations.

The first step of this process, is to check if the written program respects the syntax of the language. To this aim, the program is broken into tokens — which are keywords, identifiers, literals, operators, separators, or other elements of the language. This transformation from a sequence of characters into a sequence of tokens is called lexical analysis and is performed by a lexer.

Next, it is necessary to check if the generated sequence forms a valid statement, i.e., if it complies to the expected syntactic structure of the language. This checking is called parsing, or syntactic analysis, and is performed by a parser.

There are currently tools to perform parsing without having to implement a parser by hand, and there are also DSLs that, given the specification (grammar) of the language, automatically generate the code for both lexer and parser. These tools are called parser generators or compiler-compilers, and some of the most common are: Bison and Flex in the C context, or ANTLR in the Java context. The former

---

6 https://www.eclipse.org/Xtext/
automatically generates the parser and lexer in C given the specification of syntactic structure (Bison) and lexical structure (Flex), respectively. The latter allows the definition of both syntactic and lexical specifications (grammar) in a single file, and from there it automatically generates the parser in Java.

To exemplify what a grammar specification looks like in ANTLR, a code snippet of a construct ‘expression’ that represents recursive arithmetic expressions is displayed below:

1. expression
2.   :   INT
3. | expression '*' expression
4. | expression '+' expression
5. | expression '-' expression
6. | expression '/' expression
7. ;

After the parsing stage, and since the instructions are now classified as syntactically correct, it is necessary to check the overall correctness of the program in hands. One of the checks that cannot be performed during parsing is type checking (e.g. a string value cannot be assigned to an integer variable), and although it may be possible to embed some type checking in a grammar specification — not all, since other type checks can only be performed after additional parts have already been parsed — it would probably make that specification more complex and therefore take additional effort.

Type checking is part of the semantic analysis of a program, which often includes the management of the symbol table (handling the declaration of variables and their scope). To execute this step and to avoid the parsing of the same text multiple times, it is necessary to somehow build a representation of the previously parsed program and store it in memory. An adequate representation is, for instance, a tree structure called Abstract Syntax Tree (AST) where each node represents a construct of the program. Once the AST is stored in memory, all additional semantic checks are performed on it and therefore the program does not need to be parsed anymore. If these checks succeed entirely, the AST can also be used for the final stage of the implementation which can be either the interpretation of the program or code generation.

The implementation of the AST is not a straightforward process and includes a significant amount of effort. For instance, using Java, a Java class would typically have to be written for each construct of the language and supplementary abstract classes as well to, for example, deal with recursion.

Then, as a final step, it is required to ‘annotate’ the grammar specification with ‘actions’ (Java code blocks embedded in the grammar specification) that construct the AST during the parsing. The following snippet demonstrates this process but does not necessarily respect the correct ANTLR syntax.

```
-   expression
-     :   INT { $value = new Literal(Integer.parseInt($INT.text)); }
-     | left=expression '+' right=expression {
-         $value = new BinaryExpression($left.value, $right.value);
-         $value.setOperator("+");
-     }
```
Regardless of all the endeavor spent implementing the DSL and mechanisms to read, validate, and execute programs written in that DSL, it is not guaranteed that programmers will use it. This uncertainty comes from the fact that many programmers prefer to use powerful IDEs (such as Eclipse or IntelliJ) during their work. For this particular reason, a DSL should be shipped with good IDE support in order to be most likely adopted by end users. There are specific and powerful features that contribute for this preference and that make it easier for users to learn, use, and maintain their DSLs. The most important features concerning IDE integration are briefly described below.

- **Syntax Highlighting.** Allows different language elements to have different colors (and/or formats) which is not just a cosmetic ability. This feature gives immediate feedback concerning the syntactic correctness of what is being written, and helps to track missing or wrongly inserted closing elements such as quotation marks (if strings are displayed in red and if the whole text is red, then it is evident that a closing quotation mark is missing).

- **Background Validation.** The process of writing/editing a program with a text editor, saving it, switching to the command line, running the program, and repeating in the case of errors, is extremely exhausting and unproductive. A programming environment should warn the programmer about errors as fast as possible, and, preferably, while the programmer is writing, disregarding if the current file has been saved yet or not. This feature contributes for a prompter debugging phase, which will lower the cost in terms of time and mental effort to correct the triggered error.

- **Error Markers.** As a complementation of the previous feature, it is also necessary to highlight the parts of the program with errors directly in the editor by underlining it with a red line. In addition to this marker, a comprehensive message about the error should be displayed on the left of the editor (aligned with the lines containing errors) or when the programmer hovers the cursor over the marker. Additionally, the environment should also fill a table (‘Problems view’ in Eclipse) with all the errors to display information about all the parts of the program that need to be fixed.

- **Content Assist.** Provides, automatically or on demand, suggestions on how to complete a certain statement/expression being written. This proposal should make sense in the current program context in order to form valid instructions. An example of this feature in a Java file and
using Eclipse, could be the suggestion of Java class names after the insertion of the keyword 'new'. This feature can lead to a tremendous productivity boost, especially when using DSLs (not as common as GPLs like Java) because it allows users to learn the language while writing specifications instead of having to know all the syntax by heart.

- **Hyperlinking.** Allows the navigation between references in a program. For example, it is possible to jump from a variable to its declaration, or from a function call to the function definition. Once again, this feature saves time by providing an alternative to Ctrl + F (Search shortcut) or by helping on fixing errors related to missing declarations or invalid scope. Hyperlinking helps hand-in-hand with hovering which is the feature that triggers the display of information about a program element in a pop-up window, just by hovering over that element. Additionally, hovering can be very useful to know the type of a specific variable or the return type of a function given a certain function call, for example.

- **Quickfixes.** There are some mistakes made by the programmer that can be fixed by the editor itself. The editor offers one or a set of suggested quickfixes and the programmer may choose one of them, or none. For example, the invocation of a specific method that does not exist will trigger some quickfixes such as the ability to create the given method, or invoke another method with a similar name in case the programmer just misspelled the method name. Although quickfixes are a great functionality, sometimes the proposals do not solve the problem in the right way despite solving the error itself and making the error marker disappear.

- **Outline.** When dealing with big and complex programs, it surely helps to have an outline of it showing only the main components. This outline should allow the programmer to click on an element and directly jump to the corresponding source line in the editor. Furthermore, the outline often includes other important pieces of information such as types (e.g. variable types and return types), and structure (e.g. which variables are declared within the scope of this method) that may not be immediately understood by just looking directly to the code.

- **Automatic Build.** Last but not least, automatic build is a common feature amongst famous IDEs. Eclipse automatically compiles a file, and all dependent files, if that certain file is modified and saved. This not only helps programmers know about errors earlier, but also avoid the painful process of manually calling the compiler each time a possible disruptive change is introduced.

Taking into account all the aforementioned information, it is straightforward that (i) having to implement by hand all aspects of a DSL is an immensely arduous process; and that (ii) without support from an IDE full of powerful features, the DSL will not be adopted by users.

Xtext is an Eclipse framework for implementing programming languages and DSLs. It allows the quick implementation of new languages, and, most importantly, it covers all aspects of a complete
language infrastructure such as the parser, code generator, and interpreter. This means that all the pitiful process of implementing these aspects by hand, is single-handedly carried out by the framework itself.

Besides this enormous advantage, Xtext also provides a complete Eclipse IDE integration, which will increase the likelihood of its adoption, with all the common IDE features described above-mentioned.

Therefore, Xtext provides an alternative for the painful process of implementing all the aspects of a DSL, and, due to its Eclipse integration, also allows the usage of powerful features.

Furthermore, to start a DSL implementation only a grammar specification (similar to ANTLR) is needed and it is not even necessary to annotate the rules with actions to build the AST (as exemplified on the ANTLR code snippet above), since the creation of the AST and definition of all the Java classes to store it, are automatically handled by Xtext. From the grammar, Xtext will generate the lexer, the parser, the AST model, the representation of the parsed program on the AST, and launch an Eclipse editor with all its intrinsic features.

Despite providing decent default implementations for all these aspects, Xtext also allows the customization of each of them. To this end, Xtend, a Java-like language which will be presented in the following section, is used.

To conclude it is worth mentioning that since version 2.9.0, it is also possible to use Xtext with IntelliJ and to embed the DSL editor in a web application, which will certainly increase its popularity and adoption amongst users.

### 4.2. Xtend

Xtend is a fully featured general purpose Java-like programming language, implemented in Xtext, that is completely interoperable with Java. It can be used independently from Xtext for example as an alternative to Java or even used simultaneously with it for any type of Java application development, from web applications to Android applications. It has a more concise syntax than Java and provides distinct powerful features such as type inference (automatic deduction of the data type of an expression), extension methods (enhance closed types with new functionality), dispatch methods (selecting which implementation of a polymorphic method to call at run time), lambda expressions (anonymous functions to create delegates), and multiline template expressions which are extremely useful for defining code generators.

Instead of using Java to implement all the aspects of a DSL implemented in Xtext, users can implement those aspects with Xtend since it is easier and allows the writing of better readable code.

---

7 https://www.eclipse.org/xtend/
Besides that, it is possible to reuse all the Java libraries, since these two languages are completely interoperable, as well as work with all the Eclipse JDT (Java Development Tools) seamlessly.

Xtend can be used to write all parts of a DSL implementation, and all the stub classes generated by Xtext for a certain DSL are Xtend classes (despite being possible to customize the MWE2 workflow file for generating Java classes instead). Namely, to customize UI features, to write tests, to implement constraint checks, or even to write code generators or interpreters.

Besides providing useful mechanisms such as multiline template expressions for writing code generators, Xtend also provides features that make model visiting and traversing easier, more straightforward and simple to read and maintain. After the definition of the grammar using Xtext, the rest of the time when creating a new language will be spent visiting the AST model to customize all the other aspects of a language.

In terms of the language itself, Xtend can be considered as a statically typed language that uses the Java type system but is also commonly described as a less “noisy” version of Java. Java linguistic concepts such as classes, interfaces, and methods are similarly used in Xtend, and even the inclusion of Java generics and Java annotations was not despised during Xtend definition. However, considering the goal of having a better Java with less “noise”, Xtend developers identified redundant linguistic features present in Java that only make .java programs more verbose. Therefore, some dissimilarities between the syntaxes of the two languages came from the removal of this syntactic noise from Xtend, and an example of it is the non-mandatory usage of terminating semicolons (;).

Also, as already mentioned, Xtend is completely interoperable with Java and therefore it provides the same IDE tooling of the Eclipse JDT. In addition to this, since it is implemented with Xtext, its integration into Eclipse provides rich tooling features such as: (i) content assist for all the existing Java libraries types and methods; (ii) automatic insertion of imports when using a certain class/method; (iii) shortcut (Ctrl + Shift + O) for organizing imports and deleting unused; (iv) call hierarchy view; or (v) refactoring mechanisms including renaming, or extraction of variables and methods.

Despite the versatility of Xtend, for this project it was only used for writing additional checks for the grammar (validation), for customizing the labels of the Outline View and for customizing the outline structure of the specifications, while Java was used for the syntax coloring although Xtend could also have been chosen for this task. Therefore, Xtend was not used to its full potential since some of its most notable advantages in comparison to Java, such as multiline template expressions for writing code generators, were not used (30).
4.3. Apache POIo

Apache POI (or Apache Poor Obfuscation Implementation) is a Java library for reading and writing Microsoft Office files using Java programs. It is the most popular API amongst programmers to create, modify and display Excel, Word and even Powerpoint, Outlook, Visio or Publisher files. Furthermore, it is totally open source and maintained by Apache Software Foundation, and most importantly has a huge number of users and therefore a tremendous amount of documentation, tutorials and active forums.

It contains classes and methods to treat data from MS Office files, and a complete list of the API components is given below.

- **POIFS (Poor Obfuscation Implementation File System).** Basic factor of all other POI elements, and is used to read different files explicitly.
- **HSSF (Horrible Spreadsheet Format).** Used to read and write .xls format of MS Excel files (1997-2007).
- **XSSF (XML Spreadsheet Format).** Used to read and write .xlsx format of MS Excel files (2007-present).
- **HPSF (Horrible Property Set Format).** Used to extract property sets of the MS Office files.
- **HWPF (Horrible Word Processor Format).** Used to read and write .doc format of MS Word files (1997-2007).
- **XWPF (XML Word Processor Format).** Used to read and write .docx format of MS Word files (2007-present).
- **HSLF (Horrible Slide Layout Format).** Used to read, create, and edit .ppt format of MS PowerPoint files (1997-2007).
- **XSLF (XML Slide Layout Format).** Used to read, create, and edit .pptx format of MS PowerPoint files (2007-present).
- **HSMF (Horrible Stupid Mail Format).** Used to read Outlook MSG files.
- **HDGF (Horrible DiaGram Format).** Contains classes and methods for MS Visio binary files.
- **XDGF (XML DiaGram Format).** Contains classes and methods for MS Vision XML files.
- **HPBF (Horrible Publisher Format).** Used to read and write MS Publisher files.

However, the only components used for the purpose of this project were: (i) XSSF, for dealing with Excel files; and (ii) XWPF, for dealing with Word files. In the case of the former both read and writing features were used, while for the latter only classes and methods for writing were utilized. Despite the library already coming with innumerous methods, some auxiliary methods had to be defined in order to perform searches of cells containing a specific string or being of a specific type. Apache POI forums were also useful and allowed the surpassing of the existing uncertainties and implementation of complex and unusual features such as copying the cell style of a cell and reuse it on others.

---

8 https://poi.apache.org/
4.4. MPXJ (Microsoft Project Exchange in Java)\(^9\)

MPXJ is a library that provides a set of facilities to allow MS-Project information to be manipulated in Java and .NET. It supports the following set of data formats:

- **MPX (Microsoft Project Exchange)**. This format can be read by versions of Microsoft Project up to and including Project 2010, and written by all versions up to Project 98.
- **MPP (Microsoft Project)**. This format belongs to Microsoft and MPXJ only supports reading access to files produced by multiple versions of Microsoft Project, including the most recently one Project 2016. MPP template files, i.e. MPT files, are also supported.
- **MSPDI (Microsoft Project Data Interchange)**. This format is Microsoft’s XML file format for storing project data. MPXJ allows MSPDI files to be created, read, and written for some versions until Microsoft Project 2010.
- **MPD (Microsoft Project Database)**. This format is an Access database file format to store one or more projects. MPXJ can only read projects stored in an MPD file.
- **Planner**. Planner is a project management tool which uses an XML file format to store project data, and MPXJ can both read and write this format.
- **Primavera**. High end project planning tool for users with complex planning requirements. It exports data in both XER or PM XML files, which are supported by MPXJ for readings.
- **Powerproject**. Asta Powerproject is a planning tool that can save data to PP or MDB database files, and both can be read by MPXJ.
- **Phoenix**. Easy-to-use critical path method scheduling tool that generates PPX files which can be read by MPXJ.
- **FastTrack**. General purpose planning tool that exports to the FTS format which can be read by MPXJ.

For this work, only the second item from this set of formats was used, i.e. MPP. Unfortunately, MPXJ only provides methods for reading MPP files and not to write it. Their answer to this lack of functionality is:

“...Although it is technically feasible to generate an MPP file, the knowledge we have of the file structure is still relatively incomplete, despite the amount of data we are able to correctly extract. It is therefore likely to take a considerable amount of development effort to make this work, and it is conceivable that we will not be able to write the full set of attributes that MPXJ supports back into the MPP file - simply because we don't understand the format well enough…”

MPXJ also has a decent number of active users, posting in forums and a good documentation with the concepts explained in a very easy-to-understand manner. It also has an active team working on

improvements to the library that provides quick feedback and fixing of errors. During the implementation of the functionality for importing MPP files, a bug was found when importing Project 2016 files but that could not be replicated when using 2013 version. I have sent an email to one of their developers showing the error and how to trigger it, and, after an instant response, the bug was fixed in four days.

4.5. VBA (Visual Basic for Applications)

VBA was only used for implementing a macro to be run on Excel. The problem existed only on the RAM (Responsibility Assignment Matrix) sheet, where each cell contains a drop-down list with all the possible responsibilities: X, D, d, P, T, I, C and A. However, even though stakeholders can have multiple responsibilities for a specific task in hands, Excel does not allow the selection of two items from a drop-down list or implements this feature by default.

Therefore, implementing this feature by means of a macro written in VBA was an indispensable task, since PSL supports RAM validation by providing checks for each of its rules (e.g. only one X or P per line, etc.). The following code snippet represents the complete macro definition for allowing the selection of multiple items from the drop-down list without allowing repetition.

```vba
1. Dim old_value As String
2. Dim new_value As String
3. Application.EnableEvents = True
4. On Error GoTo Exit
5. If Target.Address = "$C$2" Then
6.   If Target.SpecialCells(xlCellTypeAllValidation) Is Nothing Then
7.     GoTo Exit
8.   Else: If Target.Value = "" Then GoTo Exit Else
9.     Application.EnableEvents = False
10.    NewValue = Target.Value
11.    Application.Undo
12.    Old_value = Target.Value
13.    If Old_value = "" Then
14.       Target.Value = New_value
15.    Else
16.       If InStr(1, Old_value, New_value) = 0 Then
17.          Target.Value = Old_value & "", " & New_value
18.    Else:
19.       Target.Value = Old_value
20.   End If
21. End If
22. End If
23. Application.EnableEvents = True
24. Exit:
25. Application.EnableEvents = True
26. End Sub
```
5. ProjectLingo Approach

This chapter describes the ProjectLingo approach and all its tools, as well as work developed towards their implementation. Thus, the remaining of this chapter is as follows: (i) section 5.1 provides an extensive description of PSL and PSL constructs and views; (ii) section 5.2 identifies and describes ProjectLingo-Studio’s components; (iii) section 5.3 provides insights about the PSL Excel and Word templates and their importance for the import and export features of ProjectLingo-Studio; (iv) section 5.4 introduces validations and examples of implemented validations for PSL; and finally, (v) section 5.5 lists the customized aspects of ProjectLingo-Studio and their relevance.

5.1. Project Specification Language (PSL)

Project Specification Language (PSL) is a domain-specific language whose grammar was defined using Xtext. This language was specified using the PSL Excel Template as a guide and covers all the aspects included on it, and consequently most of the main concepts of a project plan document. This template is divided in 18 sheets, each referring to an aspect (see Appendix A), plus an index and a configuration sheet.

PSL reuses another grammar (‘grammar mixin’) called Terminals, which already includes a default set of predefined and often required terminal rules (usually referred as token or lexer rules) such as INT, STRING and ID rules, but also hidden terminal rules that define the concepts of white spaces (WS), and multi and single-line comments (ML_COMMENT and SL_COMMENT, respectively). A terminal rule produces a single atomic terminal token and it is possible to define new terminal rules, always considering that their order is crucial as one may shadow another, or overwrite rules from the grammar being used, considering that declared rules will have a higher priority than imported ones.

To avoid the abovementioned problems, it is usually recommended to use data type rules instead of defining new terminal rules. Following this recommendation, PSL has a data type for qualified names and for qualified names with a wildcard:

1. QualifiedName hidden():
2. ID (\'\' ID)*;
If instead it was defined as a terminal rule, and if a qualified name with only one segment was given as input, the introduced qualified name would look like a plain ID and therefore this rule would be shadowed by the terminal rule ID. Data type rules are also used to describe project-related aspects such as, for example, the type of a Project (ProjectType) or the role of a Stakeholder (StakeholderRole). A complete list of all the data type rules for PSL can be seen in Appendix C.

Back to the previous code snippet, the hidden method call, without arguments, prevents the user from inputting compiler-ignored symbols (namely, white spaces and comments) between fragments of the rule (e.g. between ID and ‘.’). This method can receive the names of hidden terminal rules (such as WS, ML_COMMENT and SL_COMMENT) as arguments separated by commas, meaning that those arguments are allowed between fragments for the rule invoking it. Finally, the ‘?’ symbol, from the second rule, is the cardinality of the fragment attached to it (i.e. ‘.’’) and means ‘optional’. The other types of existing multiplicities in Xtext syntax are: not represented by default, which means exactly one (i.e. mandatory); ‘+’ meaning one or many; and ‘*’ that means zero or many.

However, and contrariwise to the previous recommendation, a terminal rule for DOUBLEs was defined:

```
1. terminal DOUBLE:
2.   INT \'.\' INT;
```

In this case, the shadowing problem cannot be replicated since the rule does not have any optional fragments that could overlap the terminal rule INT: a DOUBLE will always be an INT followed by a dot, followed by another INT. After having definitions for INTs and DOUBLEs a rule for accepting both types was defined and used by a newly created rule for percentages, as shown by the following snippet:

```
1. DoubleOrInt:
2.   DOUBLE | INT;
3.
4. Percentage hidden():
5.   DoubleOrInt \'%\';
```

Now that some basic concepts about Xtext and PSL have been introduced, a top-down journey through the most important constructs of the grammar will be provided. These constructs are not defined as either terminal or type rules, but parser rules instead. This new type of rules, contrarily to terminal rules, do not produce a single atomic terminal token but a tree of non-terminal and terminal tokens. As already mentioned, PSL was defined based on the previously created PSL Excel Template and therefore is completely aligned with it.
### 5.1.1. Packages

As illustrated in Figure 6, a project plan specified in PSL (i.e., a project plan model) is defined as a set of PSL packages. For each PSL specification, it is only possible to define a single package of each of the eight types. Then, one package can use or access constructs defined in other packages, which is represented by the ‘import’ package relation. This means that each package should be well identified, so it can be referenced by other packages. In PSL, each package has the special attribute ‘name’ of type QualifiedName. This attribute will hold the point of reference for other packages, i.e. it will univocally identify the package for future references. After making sure each package can be referenced, it was necessary to define a new rule for the ‘import’ relation which is simply the keyword ‘import’ followed by a name of type QualifiedName. Therefore, the rule for the package charter is as follows:

1. PackageCharter:
2. ‘package’ ‘charter’ name = QualifiedName
3. imports += Import*
4. project = Project
5. ;

For the sake of simplicity, the rules for the remaining packages will be omitted.

---

**Figure 6. PSL Model as a Set of Packages**
5.1.2. Charter

This rule has a single (besides name) mandatory attribute project of type Project (Figure 7) and, as its name suggests, depicts the concept of a project charter from a project plan.

![Diagram for PackageCharter rule](image)

5.1.3. Scope

The PackageScope is constituted by four optional ‘block’ features: requirements, WBS, deliverables and milestones (Figure 8).

![Diagram for PackageScope rule](image)

Block features are a type of rules introduced in PSL, due to the need of having multiple items of the same type together inside a block instead of having an enormous list of scattered items. These rules were defined for all constructs that could face this ‘issue’ and are simply as shown in the code snippet below for the Requirements construct:

```plaintext
1. Requirements:
2.   ('Requirements' '{'
3.     requirements += Requirement+
4.   '}' | requirements += Requirement
5. );
```
5.1.4. Time

This package only has an optional block feature of type *Timetable* (Figure 9), which in its turn is a set of *Tasks*.

![Figure 9. Diagram for PackageTime rule](image)

5.1.5. HR

HR, or Human Resources, is composed of the Organization Breakdown Structure (OBS) and Responsibility Assignment Matrix (RAM) blocks (Figure 10).

![Figure 10. Diagram for PackageHR rule](image)

5.1.6. Communication

According to the PMBOK, issues should be placed into the communication chapter along with communication flows and meetings. Following this standard, PSL has these three concepts inside the communication package (Figure 11).
5.1.7. Cost

The *PackageCost* holds blocks for the cost rates and for the expenses (Figure 12). These two concepts interrelate since each expense has a rate measure associated to it (such as Yearly or Monthly).

5.1.8. Risk

As its name indicates, it is within the *PackageRisk* that the identified risks for the project will be defined (Figure 13). This includes the identification, assessment and treatment phases.
5.1.8.1. Quality

Finally, the PackageQuality will store the required metrics for the project (Figure 14). These metrics are defined by a range with a minimum and maximum value, and can be, for example, used for evaluating the user satisfaction or to simply specify the percentage concept (i.e. range between 1 and 100). Metrics defined on this package can then be used to mathematically represent the acceptance criteria of the project (e.g. user satisfaction should be higher than 4 in a 0-5 scale).

![Package Quality Diagram](image)

Figure 14. Diagram for PackageQuality rule

5.1.9. Views

A layer below the Packages layer, PSL includes constructs logically arranged into views according to the specific project plan concerns they address. This section overviews PSL by introducing the views for Project, WorkComponent, Task, Requirement and Stakeholder, which are some of the most important PSL constructs.

5.1.9.1. Project View

The Project view defines the project that will take place just as a project charter of a project plan. A project should be classified by an ID (name), a more descriptive name (nameAlias), and by its own type (ProjectType) namely as a System Development, Design, System Deployment, System Maintenance, Auditing, Training, Research, Sales & Marketing, or Other. Optionally it can also be categorized by its nationality (ProjectNationalityType), i.e. either National or International. Additionally, it is also identified by its domain (ProjectDomain), i.e. Public Sector, Education, Health, Telecoms, Energy & Utilities, Finance & Banks, or Other.

Furthermore, it has four optional blocks for the involved organizations, planned schedules and budgets, and success items (success factors, success criteria and business benefits) and other optional fields that represent project-related concerns. Table 3 summarizes the fragments included in a Project specification.
Table 3. Summary of Project fragments

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Multiplicity</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>1</td>
<td>ID</td>
<td>Unique identifier of the element.</td>
</tr>
<tr>
<td>type</td>
<td>1</td>
<td>ProjectType</td>
<td>Type of the element.</td>
</tr>
<tr>
<td>nationality</td>
<td>0..1</td>
<td>ProjectNationalityType</td>
<td>Nationality type of the element.</td>
</tr>
<tr>
<td>nameAlias</td>
<td>1</td>
<td>STRING</td>
<td>Name of the element.</td>
</tr>
<tr>
<td>domain</td>
<td>1</td>
<td>ProjectDomain</td>
<td>Domain of the element.</td>
</tr>
<tr>
<td>organizations</td>
<td>0..1</td>
<td>Organizations</td>
<td>Set of organizations.</td>
</tr>
<tr>
<td>schedules</td>
<td>0..1</td>
<td>Schedules</td>
<td>Set of schedules (maximum 2).</td>
</tr>
<tr>
<td>budgets</td>
<td>0..1</td>
<td>Budgets</td>
<td>Set of budgets (maximum 2).</td>
</tr>
<tr>
<td>summary</td>
<td>0..1</td>
<td>STRING</td>
<td>Summary of the project.</td>
</tr>
<tr>
<td>purposeWhy</td>
<td>0..1</td>
<td>STRING</td>
<td>Purpose of the project, answering the ‘Why’ question.</td>
</tr>
<tr>
<td>purposeWhat</td>
<td>0..1</td>
<td>STRING</td>
<td>Purpose of the project, answering the ‘What’ question.</td>
</tr>
<tr>
<td>successItems</td>
<td>0..1</td>
<td>SuccessItems</td>
<td>Set of success factors, success criteria and business benefits relevant for the project.</td>
</tr>
<tr>
<td>progress</td>
<td>0..1</td>
<td>Status</td>
<td>Current status of the element.</td>
</tr>
<tr>
<td>version</td>
<td>0..1</td>
<td>DoubleOrInt</td>
<td>Current version of the artifact.</td>
</tr>
</tbody>
</table>

5.1.9.2. WorkComponent View

The WorkComponent view defines a work component, and a set of WorkComponents constitute the WBS. Analogously to the previous view, a WorkComponent is identified by its ID (name), name (nameAlias) and type (WorkComponentType). The latter can be one of the following: Project, Sub-Project, Phase, Workpackage, or Other. Additionally, WorkComponents can also be part of each other (partOf relation). Table 4 summarizes all the fragments of a WorkComponent.

Table 4. Summary of WorkComponent fragments

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Multiplicity</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>1</td>
<td>ID</td>
<td>Unique identifier of the element.</td>
</tr>
<tr>
<td>type</td>
<td>1</td>
<td>WorkComponentType</td>
<td>Type of the element.</td>
</tr>
<tr>
<td>nameAlias</td>
<td>1</td>
<td>STRING</td>
<td>Name of the element.</td>
</tr>
<tr>
<td>partOf</td>
<td>0..1</td>
<td>[WorkComponent]</td>
<td>Aggregation relation between elements.</td>
</tr>
</tbody>
</table>
### 5.1.9.3. Task View

The Task view defines a task, and a set of Tasks constitute the Timetable construct that can be compared to an activity-level WBS. Just as the WorkComponent, a Task is identified by an ID (name), name (nameAlias) and type (TaskType). The latter has the same values as the WorkComponentType, plus Milestone and Activity. It has a similar partOf relation, but also has a dependsOn relation that represents a dependency between Tasks instead of a composition. These and the remaining of the fragments for the Task view can be seen in Table 5.

**Table 5. Summary of Task fragments**

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Multiplicity</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>1</td>
<td>ID</td>
<td>Unique identifier of the element.</td>
</tr>
<tr>
<td>type</td>
<td>1</td>
<td>TaskType</td>
<td>Type of the element.</td>
</tr>
<tr>
<td>nameAlias</td>
<td>1</td>
<td>STRING</td>
<td>Name of the element.</td>
</tr>
<tr>
<td>quality</td>
<td>0..1</td>
<td>QualityType</td>
<td>Quality strategy associated to the element.</td>
</tr>
<tr>
<td>partOf</td>
<td>0..1</td>
<td>[Task]</td>
<td>Aggregation relation between elements.</td>
</tr>
<tr>
<td>schedule</td>
<td>0..1</td>
<td>Schedule</td>
<td>Planned period of time for the task.</td>
</tr>
<tr>
<td>duration</td>
<td>0..1</td>
<td>INT</td>
<td>Duration in days for the task.</td>
</tr>
<tr>
<td>dependency</td>
<td>0..1</td>
<td>DependencyType</td>
<td>Dependency type between this task and its previous/next task.</td>
</tr>
<tr>
<td>dependsOn</td>
<td>0..1</td>
<td>[Task]</td>
<td>Dependency relation between tasks.</td>
</tr>
<tr>
<td>lag</td>
<td>0..1</td>
<td>INT</td>
<td>Delay in the successor task.</td>
</tr>
<tr>
<td>status</td>
<td>0..1</td>
<td>Status</td>
<td>Current status of the element.</td>
</tr>
</tbody>
</table>

### 5.1.9.4. Requirement View

This view represents the project requirements but in less detail than RSL’s requirements (a reusability strategy for both is suggested in section 7.1). It is identified by an ID (name), a name (nameAlias), and by a type (RequirementType) that can either be Main, Minor, or Other. Additionally, it is possible (but not
mandatory) to specify the current status of the Requirement (Status), i.e. Not Plan, Plan, On Design, On Develop, On Test, On Deploy, or Concluded. Table 6 resumes these and other fragments of this view.

Table 6. Summary of Requirement fragments

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Multiplicity</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>1</td>
<td>ID</td>
<td>Unique identifier of the element.</td>
</tr>
<tr>
<td>type</td>
<td>1</td>
<td>RequirementType</td>
<td>Type of the element.</td>
</tr>
<tr>
<td>nameAlias</td>
<td>1</td>
<td>STRING</td>
<td>Name of the element.</td>
</tr>
<tr>
<td>status</td>
<td>0..1</td>
<td>Status</td>
<td>Current status of the element.</td>
</tr>
<tr>
<td>description</td>
<td>0..1</td>
<td>STRING</td>
<td>General description of the element.</td>
</tr>
</tbody>
</table>

5.1.9.5. Stakeholder View

Finally, the Stakeholder view defines who are the participants of the project and their interrelations. A Stakeholder is identified by an ID (name), a name (nameAlias), a type (StakeholderType: Organization, Business Unit, Team, Person, External System, Other), a role (StakeholderRole: Project Owner, Project Sponsor, Project Manager, Team Leader, Team Member, Functional Manager, Other) and by the type of the organization it belongs to (StakeholderOrganization: Customer, Performing, Sub-contractor, Other). Additionally it also has optional partOf and reportTo relations and description. Table 7 summarizes these fragments.

Table 7. Summary of Stakeholder fragments

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Multiplicity</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>1</td>
<td>ID</td>
<td>Unique identifier of the element.</td>
</tr>
<tr>
<td>type</td>
<td>1</td>
<td>StakeholderType</td>
<td>Type of the element.</td>
</tr>
<tr>
<td>nameAlias</td>
<td>1</td>
<td>STRING</td>
<td>Name of the element.</td>
</tr>
<tr>
<td>role</td>
<td>1</td>
<td>StakeholderRole</td>
<td>Role of the stakeholder on the project.</td>
</tr>
<tr>
<td>organization</td>
<td>1</td>
<td>StakeholderOrganization</td>
<td>Type of organization represented by the stakeholder.</td>
</tr>
<tr>
<td>partOf</td>
<td>0..1</td>
<td>[Stakeholder]</td>
<td>Aggregation relation between elements.</td>
</tr>
<tr>
<td>reportTo</td>
<td>0..1</td>
<td>[Stakeholder]</td>
<td>‘Many’</td>
</tr>
<tr>
<td>description</td>
<td>0..1</td>
<td>STRING</td>
<td>General description of the element.</td>
</tr>
</tbody>
</table>
5.2. ProjectLingo-Studio

ProjectLingo-Studio (or simply “Studio” for the sake of brevity) is a ready-to-use tool, built on top of the Eclipse IDE more specifically leveraging the Eclipse Modeling Framework (EMF) and Xtext technologies (see Figure 15). It materializes the ProjectLingo approach by providing the technological support for users being able to specify and validate PSL specifications, and export those to other formats. Studio relies on a multi-transformation approach where PSL acts as an intermediate language used to represent the different concepts of a project plan, and then from a PSL specification it is possible to generate more readable representations of these project plans by, for example, exporting it to Excel or Word formats.

![Figure 15. ProjectLingo-Studio](image)

As can be seen from Figure 15, Studio is divided in four main parts: (i) editor, (ii) project explorer, (iii) menu, and (iv) outline view. These parts will now be briefly described.
- **Editor.** Studio’s textual editor allows the creation and edition of PSL files. It was developed using the Xtext framework, and its syntax highlighting and proposal provider were customized for a better user experience. Error markers for errors issued by the implemented checks for PSL are displayed on Studio’s editor as well as a descriptive message about the error. Files containing errors are considered to be incorrect, and should be fixed before exporting to other formats in order to avoid possible wrong plan specifications or Studio crashes.

- **Project Explorer.** The project explorer is an Eclipse feature that was customized for Studio. When the tool is launched, the user needs to create a new general project and after creating a .psl file inside that project, or after importing an Excel or Project file with success, Studio will change the project type from general to PSL. After using one of the import or export features, Studio will also automatically create a ‘src’ and a ‘src-gen’ folder, and a ‘docs’ folder inside the latter. PSL files should be specified inside the ‘src’ folder, while the ‘src-gen’ folder will contain the generated PSL files after importing from Excel or Project. Finally, the ‘docs’ folder will contain the result from exporting a PSL file, i.e. Excel and Word files (export formats).

- **Menu.** Marked with a number ‘3’ in Figure 15, we have the ProjectLingo-Studio menu in the Main Menu bar. It is divided in five sub-menus, being the first three for the supported import/export formats, the fourth for a to-be implemented feature that will allow the generation of a WBS given a Timetable and vice-versa, and the fifth is to close the Studio window. Each of the first three sub-menus have sub-sub-menus for their implemented transformations, this means that for Excel there is both an import and export feature, that for Word only the export feature and for Project only the import feature. These features have a shortcut assigned: Ctrl+Shift+<Letter> for imports; and Ctrl+Alt+<Letter> for exports. For Excel Letter is equal to X, while for Word and Project it is W and P respectively.

- **Outline View.** On the bottom left of Figure 15 there is the Outline View which lists the elements from the PSL file opened on the editor. This view is always synchronized with the element where the mouse cursor currently is, which is good because users do not have to scroll in order to find information about the element where they currently are. Xtext’s default implementation for this view lacked utility and therefore it had to be customized to better represent PSL constructs. More information about this customization can be found in section 5.5.2.

In terms of transformations, Studio relies on M2M and M2T transformations. Excel and Project files are models, since both have a tabular and highly structured representation. Contrarily to these, a Word file is similar to a text file in the sense that it contains plain text with a low-level of formatting. It is not structured as an Excel or a Project file, despite all being internally organized in an archive of multiple XML files, and therefore is a text file instead of a model.
Thus, Studio is constituted by four transformations: (i) M2M-1, from PSL to Excel; (ii) M2M-2, from Excel to PSL; (iii) M2M-3, from Project to PSL; and a single (iv) M2T-1, from PSL to Word. Also, as stated before, Word and Excel transformations are performed using the Apache POI library while Project files are imported by using MPXJ.

5.3. Templates

The transformations from PSL to Word and Excel are performed using the Apache POI library and two companion template files (one for each of the formats). This library abstracts the complex XML structure that underlies Microsoft Office files, and was used with the Java programming language. Additionally, template files were used to allow the user to customize the style and formatting of the generated files.

Both Word and Excel templates have special tag annotations (e.g. @TaskName) that represent the dynamic part of the template, and identify which property should replace the tag during the generation. It is also possible to define the style of the tag annotations (e.g. font type, size or colour) and the generated value will have the same exact style.

The Word template is a document organized in sections and sub-sections: one section for each concept of the PSL language (e.g. Issues, Risks, Stakeholders, etc.) and a sub-section for each item of these concepts (e.g. Issue, Risk, Stakeholder, etc.). Each section and subsection is delimited by a start tag (@Start<Concept>) and an end tag (@End<Concept>). During the transformation, i.e. when exporting to Word, each sub-section is copied as many times as the existing number of items of that type, and the tags are replaced by the value of the respective property/attribute of each item. If the item lacks some property, the line containing the property is deleted. The cover page, back page and a regular page are presented in Appendix B.

The Excel template is a workbook organized in sheets, one for each concept of the PSL language (e.g. OBS, RAM, WBS, etc.) plus a configuration sheet with the values for drop-down lists (e.g. for PSL data type values) and cross-references between sheets (e.g. Milestones can be linked to a Workpackage). Each sheet name identifies the set of elements defined by it and contains a header row identifying the content of each column (e.g. Name, Type, Date, etc.) and an example row containing tags annotations (e.g. respectively, @Name, @Type, @Date, etc.). Analogously to Word’s process, during this transformation the example row is copied the same amount of times as the number of existing items for that element, and the tags replaced by the value of that specific attribute. Also, when the item does not have a value for a certain attribute it simply deletes the contents of the correspondent Excel cell. The complete template can be seen in Appendix A.

PSL Excel template is targeted for business stakeholders such as business analysts and project managers, because it should be easier for them to specify the project plan using Excel instead of having to learn a new tool which in this case is the PSL language. After specifying it, they can use ProjectLingo-Studio import feature to validate their project and after export it back to Excel or to Word.
5.4. Validation

Parsing a program is only the first stage when implementing a new programming language. In particular, it is usually not possible to determine the overall correctness of a program during this phase. However, trying to embed additional constraint checks in the grammar specification could increase its complexity or could simply be impossible as some analysis can only be performed when other parts of the program have already been parsed. In fact, the best practice is exactly the opposite: do as little as possible checks in the grammar specification phase and as much as possible in the validation phase. This approach not only allows the bypassing of the abovementioned problems, but is also better due to the possibility of providing better error messages and to more precisely detect problems that are eligible for quickfixes.

In Xtext, the mechanism responsible for implementing such validations is the validator. Validations allow the implementation of additional constraint checks to a DSL which cannot be done at parsing time. To implement such constraint checks, it is only needed to communicate to Xtext the possible errors or warnings messages, and it will automatically take care of generating the error markers, by underlining with a red line the defective parts, in the IDE. This validation always takes place in the background, providing immediate feedback to the user while he is typing. Additionally, it is also possible to implement quickfixes for each error or warning generated during validation to save the user some time while debugging.

There are two types of validators: default and custom. Default validators are those already implemented by Xtext, which deal with common errors present on most languages. Some of these standard validators are, for example, (i) checking if a name is unique within the program, or (ii) if cross-references between elements are correctly specified.

Despite being provided by Xtext, it is possible to implement quickfixes for the errors generated by these validators or even to replace the error message for something more intuitive. For instance, the error message for incorrect cross-references is “ Couldn’t resolve reference to…” which might not be easy to understand for a business stakeholder that probably does not even know (or care) about what a reference is. When using PSL, a business stakeholder only needs to know that a Stakeholder can be part of another and not that both are linked by a cross-reference. Therefore, an error message such as “Stakeholder A cannot be part of Stakeholder B because it does not exist” (for the foregoing scenario) could be adapted for each case where this error can be replicated.

Custom validators, on the other hand, are validators implemented by the programmer of the language. While the default validators can be very useful performing common validation tasks, most of the checks will have to be implemented for the developer of the language according to its desired semantics for the language.
In Xtext, this implementation of additional checks must be performed on the validation folder from the main package, more specifically on the file `<language.name>Validator.xtend` (e.g. PSLValidator.xtend). Thus, it is necessary some deep knowledge about the Xtend programming language or to change the configuration file to generate Java files instead.

Each validation is performed by a single method, and each method should have a single parameter of a specific type and be annotated with a `@Check` annotation. The name of the method is not important, but should describe in a concise manner the aim of the check (e.g. `CheckNoDuplicatePackages`). Contrarily to the name, the single parameter is of primary importance since it is necessary to access values of attributes for that specific type to see if it conforms with the expected. For the check identified above, `CheckNoDuplicatePackages`, it is required to have access to all the packages defined in the file. Hence, the parameter for this method must be the `Model` itself in order to iterate over the packages by means of its accessor method `getPackages()`.

One of the most ordinary checks that has to be implemented, deals with hierarchy cycles between elements of the language. This check was implemented in PSL, for example, for the `partOf` relation from the `Stakeholders` construct (`checkNoCycleHierarchyOnStakeholder`), and a violating scenario is represented in Figure 16.

![Figure 16. Hierarchy Cycle on Stakeholder's 'partOf' Relation](image)

In this case, B is part of A, which in its turn is part of C, which in its turn is part of B. This obviously represents an invalid statement in both PSL and real life. Despite the process being similar for each construct, this check had to be implemented for each possible instance where this error could occur and therefore multiple methods had to be defined.

In the preceding method, that receives a `Stakeholder` as an argument, the `partOf` hierarchy of a `Stakeholder` is traversed, while keeping track of all visited `Stakeholders` by storing them in a variable. Evidently, if a `Stakeholder` does not have a `partOf` relation, there is nothing to check. However, if during the traversing an already visited `Stakeholder` is found, it means that the hierarchy contains a cycle and
an error is issued. After the error being issued, it is critical to leave the traversing loop or otherwise it will traverse the hierarchy endlessly.

Nevertheless, a bug was found during the implementation of this validations: the editor (ProjectLingo-Studio) crashes whenever a ‘direct’ cycle hierarchy exists (Figure 17). This is a framework’s fault and should be resolved in its future released versions.

![Figure 17. Crash-causing Hierarchy Cycle Scenario](image)

Finally, the missing aspect about validation methods is the ‘error’ method that is called to emit an error message. It has many overloaded versions (consult Xtext documentation for further details, https://www.eclipse.org/Xtext/documentation/), but the recommended and used version for this project receives the following arguments:

- **Error Message.** Should meaningfully describes the error and might also provide vital information on how to resolve it (e.g. "Cycle in hierarchy of stakeholder <stakeholder.name>").

- **EObject.** The EObject being examined, i.e. the element of the language currently being examined, which can differ from the check method argument (e.g. current stakeholder being visited).

- **EFeature.** Attribute of the examined EObject, which the error should be reported against, i.e., which should be marked with an error marker (e.g. partOf feature).

Errors determine the validity of a model: a file with errors represents an invalid model. Warnings are an alternative to errors that can be issued by using a method with the same signature as ‘error’ but named ‘warning’ instead. They should be used when the severity of the ‘problem’ is low, and the model should not be considered invalid due to their existence. Examples of warnings could be related to conventions that do not have to be followed, such as that the nameAlias of a Stakeholder should start with a capital letter or that its name should start with the prefix ‘Stakeholder_’ (although none of these were implemented in PSL).

In addition to the checks for detecting cycles in hierarchies, several others were implemented such as: (i) checking dates correctness, both its validity (e.g. February having 29 days on a leap year) and if starting dates occur before or on the same day as ending dates for a certain Schedule (e.g. for work
components); (ii) checking if the used currencies (e.g. USD, EUR, etc) for an Expense’s cost is the same used for its price; (iii) checking if the program conforms with the Excel template (e.g. a Task only has a Schedule column on the template and is of type Planned); or even (iv) checking if the Responsibility Assignment Matrix’s (RAM) rules are applied (e.g. only an X or a P per line, etc). The complete list of performed validations and the error message issued for each of them is shown in Table 10.

To conclude the validation section, it is also worth mentioning the importance of quickfixes despite none have been implemented for the errors issued in PSL (is it one of the recommendations for future work from section 7.1). A quickfix is one or more proposals to solve a problem (error or warning) in a program. In Xtext, it is triggered by an issue code, i.e. a string that uniquely identifies the issue, associated with an error or warning market. To add a specific quickfix to an error or warning, it is necessary to provide an additional argument which represents that issue code when invoking the error or warning method. Similarly to validations, a method that implements a quickfix is annotated with a @Fix annotation and a reference to the issue code this quickfix applies to.

5.5. IDE Customization

In terms of IDE concepts, three aspects were customized: (i) syntax highlighting; (ii) outline view and labelling; and (iii) proposal providers. A thorough explanation on how this customization took place is provided in the rest of this chapter.

5.5.1. Syntax Highlighting

In section 4.1, a list with some of the most important features concerning IDE integration was given. These features help to understand why a DSL should be shipped with good IDE support to be generally accepted by its target audience. The first item of this list, not necessarily representing the most important feature, was syntax highlighting.

Syntax highlighting is about having different elements of the language with different styles (e.g. colour or font style). It is not just a cosmetic preference: despite giving immediate feedback, it also improves readability and comprehension of what is being written.

Xtext’s default highlighting uses a single colour for most of the elements, except for strings and comments. Therefore, it was essential to implement a custom syntax highlighting to give use to the aforesaid advantages of a good syntax highlighter. Table 8 summarizes the adopted colours and font styles for the different elements of the language.
Table 8. PSL elements and correspondent colours and font styles

<table>
<thead>
<tr>
<th>Element(s)</th>
<th>#RGB</th>
<th>Font Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type values (options).</td>
<td></td>
<td>bold</td>
</tr>
<tr>
<td>Attributes and strings.</td>
<td></td>
<td>bold</td>
</tr>
<tr>
<td>IDs and qualified names.</td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>Language-specific keywords.</td>
<td></td>
<td>bold</td>
</tr>
<tr>
<td>Numbers (integers and floats).</td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>Single and multi-line comments.</td>
<td></td>
<td>none</td>
</tr>
</tbody>
</table>

To better comprehend the differences a good syntax highlighting implementation can bring, the same code snippet is displayed twice in Figure 18: on the left side, the default or standard Xtext’s rendering; and on the right side, the improved syntax highlighting for PSL.

```plaintext
/* Expenses block */
Expenses {
  Expense C1 : HRs {
    Name: "Human resources (Salaries, etc.)"
    PartOf: C0
    Years {
      Year 2017 {Total: 28800 (EUR)}
      Year 2018 {Total: 43452 (EUR)}
    }
    // 28800 + 43452
    Total: 72252 (EUR)
  }
  Expense C0 : Total {
    Name: "Total Costs"
    Years {
      Year 2017 {Total: 60120 (EUR)}
      Year 2018 {Total: 72832.80 (EUR)}
    }
    Total: 132952.80 (EUR)
  }
}
```

Figure 18. Xtext's Standard Rendering vs. PSL Custom Syntax Highlighting

Each of the different implemented styles (each line from Table 8) corresponds to a distinct method that simply creates a new TextStyle object, then sets its colour and/or font style and finally returns it. After defining the styles, they are used by all of the applicable elements and keyword ‘collisions’ addressed (e.g. distinguish between ‘Project’ that is a construct from ‘Project’ that is a type of Task).
5.5.2. Outline View & Labeling

The Outline View (identified by the number 4 in Figure 15) displays an outline of the currently opened file in the editor area (number 1 in Figure 15) and lists structural elements. Since the opened file is a PSL file, the contents of this view will be PSL’s elements. These contents are editor specific, which means that for a Java file the structural elements would be classes, fields, and methods instead.

Xtext’s default Outline View provides toolbar buttons to keep it synchronized with the element currently selected in the editor (or where the mouse cursor currently is). This feature is also depicted in Figure 15, where the Timetable construct is being shown on the editor and its structural elements are listed on the Outline.

By default, this tree structure is built using the containment relations of the DSL’s metamodel. This strategy is not always optimal and its flaws can be denoted on the image below. On the top right side of Figure 19, it is shown the Outline View for the PSL code snippet of the left side. It is possible to check that the definition of both Organizations (O1 and O4) also contain the OrganizationType element (Customer and Other, respectively), as if this feature was a children node of the Organization nodes. The same scenario can be observed for the definition of the Project element: ProjectType, ProjectNationalityType and ProjectDomain features are represented as its children nodes. This default behaviour leads to a large amount of space in the Outline View being occupied with unnecessary elements, which can become a big problem when dealing with complex files with a considerable number of elements.

Nonetheless, it is possible to modify the way these elements are displayed on the Outline View by modifying the structure of the Outline tree. A possible solution to address the above-described issue, could simply be the elimination of these nodes. However, information about the elements would be lost which is not an optimal scenario considering the importance of having it displayed on the Outline View. Thus, the solution is (i) eliminating unwanted nodes from the tree; and (ii) represent the previously deleted information on the parent node. To eliminate the children nodes, it is only needed to make sure that the Organization node is a leaf node (i.e. a node without children). To achieve this, a method named _isLeaf needs to be defined with a parameter of the type of the element (in this case, Organization). This method should simply return true and Xtext will take care of the rest.
However, the same procedure cannot be applied to `Project` since the goal is not transforming it into a leaf node but instead only eliminate some of his children node (`ProjectType`, `ProjectNationalityType` and `ProjectDomain`) and keep all the remaining. To achieve this, a method named `_createChildren` needs to be defined with parameters of type `IOutlineNode` and type of the model element (in this case, `Project`). After, each of the desired children nodes (`Organizations`, `Schedules`, `Budgets` and `SuccessItems` blocks) must be manually created if it is defined (on the example above, only `Organizations` is defined which means it is the only child node that must be created), by invoking a `createNode` method.

Finally, to represent the missing information from the eliminated nodes (step 2 of the procedure) another IDE concept must be customized: labels. Labels are simply the text displayed for each element (e.g. ‘P_SystemS’ for `Project` or ‘O1’ and ‘O4’ for `Organizations`). To accomplish this task, a method `text` must be defined with a single parameter of the type of the element of the label (e.g. `Project` or `Organization`) and a string with the intended label returned. In Figure 19’s bottom right corner, is displayed the result of the above-described procedure: unwanted nodes were eliminated and the information contained on them is shown in their parent nodes. For instance, after changes, an `Organization`’s label has the following structure: ‘nameAlias (name, type)’ which preserves the information its children nodes contained (i.e. its `type`). Similarly, `Project`'s label is represented as ‘Project: nameAlias (name, type, nationality, domain)’ after redefinition of its children nodes and of its label.

Lastly, a special case associated with block elements (such as `Organizations`) also had to be addressed. As can be seen from the top right corner of Figure 19, `Organizations` was previously labeled as ‘<unnamed>’. This occurred because block elements do not have a name attribute and therefore cannot be automatically identified by Xtext and labeled on the Outline tree by default. To switch its label to the name of the block itself, the same method (`text`) was invoked and a string of the intended label returned (e.g. ‘`Organizations`’).
5.5.3. Proposal Provider

At last, it was mandatory to customize the proposal provider (or content assist) due to its faulty behavior. Analogously to the sections above, in Figure 20 there is a before and after scenario showing the proposals given by the editor when typing Ctrl+Space. As can be noticed, only two options (Concluded and Plan) out of seven were displayed for the progress of a Project before customization. This issue occurred due to the existence of keywords with spaces in PSL (e.g. Not Plan, On Deploy, etc), which led to those keywords not being proposed. To overcome this obstacle, it was required to override the method that is generated for each of the language constructs, complete_<construct> (e.g. complete_NotPlan), for all the missing keywords. Due to its complexity, implementation details will not be provided. The result after the redefinition of methods is exhibited on the right side of Figure 20.

![Figure 20. Content Assist before and after](image)
6. Evaluation

In order to evaluate and test the developed work, a ‘generic’ project plan specification was used. Despite the initial goal being the evaluation with a real-life project plan followed by a testing session with users, the project took another path. The reasons behind this deviation were: (i) implementation of all the concepts of a project plan in PSL; (ii) implementation of additional features such as those described in section 5.5; (iii) implementation of multiple security checks to the import and export handlers to prevent ProjectLingo-Studio from crashing when wrong inputs are inserted; and (iv) enhancement of the PSL Excel and Word templates to achieve a more executive style. In addition to these, ProjectLingo’s artefacts (PSL and templates) constant changing and incrementing nature also played a big role since this work will be continued and therefore can be validated afterwards.

The generic project plan example that was used, via filling of the PSL Excel Template’s sheets, had a medium level of complexity and was modified accordingly when it was required to test special cases such as erroneous input (e.g. date cells filled with strings) or after the implementation of the macro described in section 4.5.

Therefore, the lack of evaluation with a specific project was the result of a trade-off between implementing more functionality, with better quality and error-prone features, or defining a partial project plan that would only cover a few number of aspects and thereby having a lower level of complexity than the utilized.

Notwithstanding, a brief description of every possible workflow using ProjectLingo-Studio is given in the following sections of this chapter. These workflows can be reused during the testing sessions with users, to obtain feedback related to each of them and about possible improvements. This could be done by means of a guideline that users should follow, where they define or use a previously defined project plan and use the different import and export features with the multiple workflows. An analysis of the time spent performing each of the workflows could be done, and a staff member could take notes about the main difficulties faced by the users and write down all asked questions during the session.
6.1. Workflows

Considering all the possible transformations (both Model-to-Model and Model-to-Text), provided by means of import and export features available on ProjectLingo-Studio, four diverse workflows can be chosen. These workflows are described in the sections below and each is identified by a descriptive keyword. Each of them

6.1.1. Arduous

This workflow suggests the full (or partial) implementation of a project plan using the intermediate language PSL, and then export it to Excel and Word formats (Figure 21). Considering the quantity of constructs available in the language, this process can become an extremely time and effort consuming task for new users.

Notwithstanding, if only a partial project plan specification is needed, or if the user has already mastered the language and already has predefined templates or similar plans specifications, this task can become less toilsome. Another advantage of using this procedure is that it is guaranteed that the plan is correct by construction and does not need to be validated again.

![Figure 21. Workflow 1: From a PSL specification to an Excel or Word file](image)

6.1.2. Partial

Since Microsoft Project does not have features for all concepts of a project plan and that MPXJ does not have classes for a couple of others, only a partial specification can be obtained by using ProjectLingo’s Microsoft Project import functionality.

However, if only a WBS specification is wanted or if the user is an expert using Microsoft Project, the workflow exhibited in Figure 22 might be the wisest choice.
6.1.3. Complete

PSL was defined on the basis of the PSL Excel Template. Therefore, both artefacts are seamlessly aligned in terms of existing constructs and covered aspects of a project plan. This correlation leads to a certainty: a completely defined project plan using the Excel template will generate a complete project plan specification in PSL when imported.

After importing the Excel file, it is possible to verify if there are any errors issued on ProjectLingo-Studio by PSL’s validator for the generated PSL file and directly fix them before exporting it to Word or back to Excel (Figure 23).

6.1.4. Recommended

Finally, and as the best comes last, this workflow is considered to be the most efficient in order to achieve a validated project plan specification with the existing tools and artefacts. Contrarily to the other workflows, and as shown in Figure 24, this procedure receives two files as input: a Microsoft Project and an Excel file.
In fact, this workflow is a combination of the two previous ones. A Microsoft Project file containing a WBS definition and an Excel file containing the rest of the aspects are imported and merged together and then validated and exported.
7. Conclusion

There are currently dozens of PM tools and softwares available on the market, for small and big companies, following agile or waterfall approaches. These tools provide all types of features either directly or by means of interoperability with other tools. However, none of these provide features for all the concepts that constitute a project plan. Also, these tools despise the semantics of natural language and lack any type of natural language processing. This leads to several problems due to humans being more prone to error than machines, and a lot of ambiguity can arise on project plan specifications.

The ProjectLingo is an initiative with a set of tools and mechanisms that allow a more rigorous specification of project plans. First, a rigorous domain-specific language for project plan specifications, PSL, was defined using Xtext based on the PSL Excel Template. After PSL grammar specification, validation checks were implemented using Xtend. Then, an IDE for creating and editing this new type of files, ProjectLingo-Studio (Studio, for the sake of brevity), was developed. Additionally, Studio verifies the correctness of PSL specifications by means of checking the compliance with the implemented validations. Some aspects of this eclipse-based IDE were customized, namely the syntax highlighting, outline view and labels, and proposal provider. Finally, import and export features were implemented on top of Studio for the Microsoft Office formats Excel, Word and Project.

This solution mitigates some of the problems that can occur due to human errors, and provides an extensive Excel template that allows the specification of most concerns that compose a project plan. Furthermore, the provided Word template has a professional appearance that can be used after specifying the plan on top of the Excel template and validate it with Studio. PSL Excel Template is especially targeted for business stakeholders that do not have want to learn a new language, and that already have deep knowledge on project plan concerns.

Therefore, given that Studio has import features for Excel and Project (partially) and export features for Excel and Word, four workflows can be chosen by the project manager: (i) specify the plan using PSL and export it to output formats; (ii) specify the WBS using Project, validate it using Studio and export it to other formats; (iii) specify the whole project plan by filling the PSL Excel Template, validate it using Studio and export it back to Excel or to Word; and (iv) specify the WBS using Project and the remaining
Despite not solving all the problems related to the usage of natural language, this approach provides interesting tools with useful features towards that goal such as the automatic validation and interoperability mechanisms that are of utmost importance for producing better project plan specifications.

The remainder of this chapter is focused on future work suggestions to enhance the quality of the developed tools and artefacts produced.

7.1. Future Work

This section presents some of the main features that can be implemented in a future stage of this work. Either due to time restrictions, because of their complexity, or simply because they were out of the scope of this research, these proposals were conceptualized but not (fully) implemented. Nevertheless, it is important to emphasize that none of them undermine the proposed goals for this work.

- **Generate WBS from Timetable and vice-versa.** In PSL, the *WBS* is a set of *WorkComponents* while the *Timetable* is a set of *Tasks*. *WorkComponents* and *Tasks* relate in so many aspects, that is worthwhile to implement a functionality to generate one given the other as input. The attributes **nameAlias**, **partOf**, **schedules** and **status** features can all be entirely reused, and type feature can be partially reused since *WorkComponentType* is a subset of *TaskType*.

To this end it should be enough to click on a button and if the input file contains a *WBS* or a *Timetable* definition, a *Timetable* or a *WBS*, respectively, is then automatically added to the end of that same file. The buttons for this functionality are already present on the current interface of ProjectLingo-Studio, as well as shortcuts assigned to them, as shown in Figure 25, and the handler classes for each of these buttons have already been created.

![Figure 25. ProjectLingo's 'Generate' Sub-Menu](image)
- **Integrate ProjectLingo-Studio with RSLingo-Studio.** ProjectLingo and RSLingo are two strongly linked approaches. Beyond that, ProjectLingo-Studio and RSLingo-Studio interfaces are also quite similar and share related features. Thus, the implementation of an integration between both should be considered by creating a new Eclipse-based IDE, containing all functionalities, and giving it a more appropriate name – namely ITLingo-Studio – for further expansion.

- **Allow reusability between RSL and PSL requirements.** Currently, RSL and PSL requirements relate and diverge on the same number of aspects. This raises a consistency problem since both are supposed to be equal. An example of this, is that both share a *status* feature, that has a value from the same set of options, and a *description*, but they do not share the same set of options for the *type* feature. That is, RSL classifies a requirement as a *Goal, Functional, Quality, Constraint or Use Case*, while in PSL a requirement is categorized into *Main, Minor or Other*.

  Ergo, it is mandatory to solve this consistency and coherence problem by restructuring one/both syntax/syntaxes to allow reusability between .rsl and .psl formats. A possible solution could be the definition of an additional feature, e.g. *subtype*, to hold the current set from PSL (*Main, Minor and Other*) and use the current *types* from RSL for the existing *type* feature of RSL and PSL. Other dissimilarities amongst syntaxes, is the inexistence of some RSL features on PSL. For example, the concept of composition between requirements (i.e. requirement A is part of requirement B), and a *priority* feature are missing. The easiest and most effective way of solving these divergences, could be the inclusion of the missing features on future PSL versions.

- **Standardize RSL and PSL syntaxes.** During the execution of the work described in this document, the syntax of RSL suffered minor changes. The curly brackets ({} ) gave place to square brackets ([ ] ) for code blocks, and some blocks were completely removed from a few constructs. These changes took place due to the association between curly brackets and programming languages, such as Java or C. Therefore, the main goal was to differentiate RSL from common programming languages such as these. Despite not personally agreeing with this fundament, it is surely mandatory to keep RSL and PSL syntaxes as similar as possible because it is most likely that the set of users from each will be the same.

  In addition to these small changes, it will also be required to extend RSL to accommodate new constructs introduced on PSL. The most notorious of these constructs, are the parser rules for blocks containing multiple items of the same type, and non-obligatory usage for a single item. A practical example of this improvement can be seen in the following code snippet, where a set of *stakeholders* is defined inside an *OBS* block instead of directly inside the scope of *PackageHR*. 


1. package HR ProjectP.HRPkg
2. import ProjectP.ScopePkg.*
3. 
4. OBS {
5.   Stakeholder Stk_CG_1 : Team {
6.     Name: "Steering Committee"
7.     Role: Other
8.     Organization: Customer
9.     Description: "TheProjectOwner"
10.   }
11. 
12.   Stakeholder Stk_C_PO : Person {
13.     Name: "TheProjectOwner"
14.     Role: Project Owner
15.     Organization: Customer
16.     PartOf: Stk_CG_1
17.     Description: "TheProjectOwner"
18.   }
19. 
20.   Stakeholder Stk_C_FM1 : Person {
21.     Name: "TheMarketingDirector"
22.     Role: Functional Manager
23.     Organization: Customer
24.     PartOf: Stk_CG_1
25.     Description: "TheMarketingDirector"
26.   }
27. }

Finally, it should also be discussed the usage of white spaces on multi-word keywords, such as on Project Owner or Functional Manager from the snippet above, since RSL have not adopted this alternative yet.

- **Add quickfixes to errors triggered by validators.** Xtext allows the definition of quickfixes for all triggered errors. This includes errors triggered by both default and custom validators. For almost all the defined validators, it is possible to define one or more quickfixes to help the user. For example, if price and cost from Budget do not use the same currency, the suggested quickfix should be changing the current cost currency to the current price currency. Another example could be the existence of duplicate packages of the same type, and in this case the provided quickfix would simply be the deletion of the second definition line.

- **Enhance error messages triggered by default validators.** Default validators take care of common validations provided by other programming languages such as solving variable names, scopes or references to inexistent entities. However, some of the error messages are not sufficiently suitable for the error in hands and therefore should be redefined. An example of this problem is replicated when the interpreter is expecting something of a certain type (for example a String) and the user introduces anything else.

  Additionally to these, there is always the possibility of implementing additional import and export features to other formats or to generate diagrams as a means of representing some concepts (e.g. OBS as a hierarchy tree).
8. References


16. MERNIK, Marjan; HEERING, Jan; SLOANE, Anthony M. When and how to develop domain-specific languages. ACM computing surveys (CSUR), 2005, 37.4: 316-344.


# Appendix A

## PSL Excel Template

### Copyright

<table>
<thead>
<tr>
<th>Copyright Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version:</strong> 4.0</td>
</tr>
<tr>
<td><strong>Date:</strong> 2017/October</td>
</tr>
<tr>
<td><strong>Author:</strong> Alberto Rodrigues da Silva</td>
</tr>
<tr>
<td><strong>Co-Author:</strong> João Francisco Vieira Gonçalves</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ProjectLingo’s Project Excel Template (ProjectLingo-Project-Excel-Template)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Copyright:</strong> Copyright 2016-2017, Alberto Rodrigues da Silva (Coordinator) et al., INESC-ID, Instituto Superior Técnico, Universidade de Lisboa. All rights reserved.</td>
</tr>
<tr>
<td><strong>Description:</strong> This repository publishes the “ProjectLingo’s Project Excel Template” that is a Project template based on the multi-view architecture and inspired on Project Management standards and frameworks such as ISO 21500, PMI PMBoK, IPMA ICB.</td>
</tr>
<tr>
<td><strong>This template includes the following files:</strong></td>
</tr>
<tr>
<td>ProjectLingo-Project-Excel-Template-vX.Y.xlsx: the “PSLingo Project Excel template” that can be customized or directly used in a project basis (please use its most recent version)</td>
</tr>
<tr>
<td>ProjectLingo-DemoProject-vX.Y.xlsx: an Excel file with a simple (and not complete) application example of this “Project Excel template”; Use its last version as a starting point for your learning and use purpose.</td>
</tr>
<tr>
<td><strong>Feedback:</strong> Feel free to use it! We would appreciate your feedback, <a href="mailto:alberto.silva@tecnico.ulisboa.pt">alberto.silva@tecnico.ulisboa.pt</a></td>
</tr>
<tr>
<td><strong>Licence:</strong> This material is released under the Eclipse Public License (EPL) - Version 1.0. <a href="http://www.eclipse.org/legal/epl-v10.html">http://www.eclipse.org/legal/epl-v10.html</a></td>
</tr>
<tr>
<td><strong>Further information:</strong> <a href="https://github.com/ITLingo">https://github.com/ITLingo</a></td>
</tr>
</tbody>
</table>
## Project Identification

<table>
<thead>
<tr>
<th>Project #</th>
<th>P_SystemS</th>
<th>Name</th>
<th>Project_SystemS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Progress</td>
<td>Not Plan</td>
<td>Version</td>
<td>1.0</td>
</tr>
</tbody>
</table>

### Types

<table>
<thead>
<tr>
<th>Project Type</th>
<th>Nationality Type</th>
<th>Application Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Development</td>
<td>National</td>
<td>Public Sector</td>
</tr>
</tbody>
</table>

### Entities

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Sponsor</th>
<th>Project Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>OrgName</td>
<td>OrgSponsor</td>
<td>OrgManager</td>
</tr>
</tbody>
</table>

### Time

<table>
<thead>
<tr>
<th>Time</th>
<th>Planned</th>
<th>Actual</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Nº Months</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Budget

<table>
<thead>
<tr>
<th>Price</th>
<th>Planned</th>
<th>Actual</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 EUR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cost</td>
<td></td>
<td></td>
<td>0 EUR</td>
</tr>
<tr>
<td>Profit</td>
<td>0 EUR</td>
<td>0 EUR</td>
<td>0 EUR</td>
</tr>
</tbody>
</table>

### Executive Summary

- **TBD_Summary**

### Purpose: Why?

- **TBD_Why**

### Purpose: What?

- **TBD_What**

### Business Benefits

- **BB_1**
  - TBD_BB_1

### Success Criteria

- **SC_1**
  - TBD_SC_1

### Success Factors

- **SF_1**
  - TBD_SF_1
Requirements

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_ID</td>
<td>R_Name</td>
<td>Main</td>
<td>Not Plan</td>
<td>R_Description</td>
</tr>
</tbody>
</table>

WBS

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>PartOf</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WC_ID</td>
<td>WC_Name</td>
<td>Project</td>
<td>WC_MA</td>
<td>Not Plan</td>
<td>WC_Description</td>
</tr>
</tbody>
</table>

Milestones

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>WorkPackage</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_ID</td>
<td>M_Name</td>
<td>Main</td>
<td>M_WP</td>
<td>Not Reached</td>
<td>M_Description</td>
</tr>
</tbody>
</table>

Deliverables

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>WorkPackage</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_ID</td>
<td>D_Name</td>
<td>Documentation</td>
<td>D_WP</td>
<td>Not Submitted</td>
<td>D_Description</td>
</tr>
</tbody>
</table>
### Timetable

#### Project Plan

**Time :: Timetable**

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>Quality Type</th>
<th>PartOf</th>
<th>Start</th>
<th>End</th>
<th>Duration</th>
<th>Type</th>
<th>Activity</th>
<th>Lag</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_ID</td>
<td>T_Name</td>
<td>Project</td>
<td>QP</td>
<td>T_PO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Not Plan</td>
</tr>
</tbody>
</table>

**Planned Dates**

**Dependency**

### OBS

**Project Plan**

**Organization Resources :: Organization Breakdown Structure**

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>Role (*)</th>
<th>Organization (*)</th>
<th>PartOf</th>
<th>ReportTo</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stk_ID</td>
<td>Stk_Name</td>
<td>Organization</td>
<td>Project Owner</td>
<td>Customer</td>
<td>Stk_PO</td>
<td>Stk_RT</td>
<td>Stk_Description</td>
</tr>
</tbody>
</table>

### RAM

**<P_SystemS> - Project Plan**

**Organization Resources :: Responsibility Assignment Matrix**

Legend:
- **X** - Executes the work
- **D** - Takes the Decision solely or primarily
- **d** - Takes the decision jointly or partly
- **P** - Controls Progress
- **T** - Provides Tutorial on the job
- **I** - Must be Informed
- **C** - Must be Consulted
- **A** - Available to Advise

<table>
<thead>
<tr>
<th>WP-ID</th>
<th>WP-Name</th>
<th>O</th>
<th>D</th>
<th>P</th>
<th>T</th>
<th>I</th>
<th>C</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_ID_1</td>
<td>T_Name</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>T_ID_1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_ID_2</td>
<td>T_Name</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>T_ID_2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_ID_3</td>
<td>T_Name</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>T_ID_3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Issues

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Issue (*)</th>
<th>Type (*)</th>
<th>Priority</th>
<th>Date</th>
<th>By Whom</th>
<th>Date</th>
<th>By Whom</th>
<th>Status</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_ID</td>
<td>I_Name</td>
<td>Bug</td>
<td>Very High</td>
<td>Opened</td>
<td></td>
<td></td>
<td></td>
<td>Opened</td>
<td>I_Comments</td>
</tr>
</tbody>
</table>

### Metrics

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>Min Value</th>
<th>Max Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QM_ID</td>
<td>QM_Name</td>
<td>HR</td>
<td></td>
<td></td>
<td>QM_Description</td>
</tr>
</tbody>
</table>

### Flows

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>Medium</th>
<th>Frequency</th>
<th>Owner</th>
<th>Audience</th>
<th>Deliverable(s)</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF_ID</td>
<td>D_ID</td>
<td>D_Name</td>
<td>Technical</td>
<td>Once</td>
<td></td>
<td></td>
<td>CF_Deliverables</td>
<td>CF_Format</td>
<td>CF_Description</td>
</tr>
</tbody>
</table>

### Meetings

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>Medium</th>
<th>Frequency</th>
<th>Owner</th>
<th>Audience</th>
<th>Deliverable(s)</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM_ID</td>
<td>CM_Name</td>
<td>Technical</td>
<td>Face-To-Face</td>
<td>Once</td>
<td>CM_Deliverables</td>
<td>CM_Format</td>
<td>CM_Description</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Risks (Identification)

Risks (Assessment)

Risks (Treatment)

Acceptance Criteria (Deliverables)
### Rates

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>Metric (*)</th>
<th>Year-1</th>
<th>Year-2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_ID</td>
<td>R_Name</td>
<td>HR</td>
<td>Monthly</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Expenses

<table>
<thead>
<tr>
<th>Id (*)</th>
<th>Name (*)</th>
<th>Type (*)</th>
<th>PartOf</th>
<th>Id</th>
<th>Metric</th>
<th>Unit Cost</th>
<th>NP</th>
<th>Value</th>
<th>Unit Cost</th>
<th>NP</th>
<th>Value</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_ID</td>
<td>E_Name</td>
<td>MU</td>
<td>E.PO</td>
<td>E_ID</td>
<td>E_Measure</td>
<td>0.00 EUR</td>
<td></td>
<td>0.00 EUR</td>
<td></td>
<td></td>
<td>0.00 EUR</td>
<td>E_Total</td>
</tr>
</tbody>
</table>
Appendix B

PSL Word Template

Cover Page
Example Page (Project Details page)

Project Details

@ProjectStart

Name: @Name
Type: @Type
Nationality: @Nationality
Domain: @Domain

Organization(s):
  - @OrgName (@OrgType):
    - Sponsor: @Sponsor
    - Manager: @Manager

Data(s):
  - @DataType: @Start (Start) - @End (End)

Budget(s):
  - @BudgetType: @Price (Price) - @Cost (Cost)

Summary: @Summary
Purpose Why: @PurposeWhy
Purpose What: @PurposeWhat
Business Benefits:
  - @BBDescription

Success Criteria:
  - @SCDescription

Success Factors:
  - @SFDDescription

Progress: @Progress
@ProjectEnd
# Appendix C

## PSL Data Types

Table 9. PSL Data Type Rules and its Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month</td>
<td>Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec</td>
</tr>
<tr>
<td>ProjectType</td>
<td>System Development, Design, System Deployment, System Maintenance, Auditing, Training, Research, Sales &amp; Marketing, Other</td>
</tr>
<tr>
<td>ProjectNationalityType</td>
<td>National, International</td>
</tr>
<tr>
<td>ProjectDomain</td>
<td>Public Sector, Education, Health, Telecoms, Energy &amp; Utilities, Finance &amp; Banks, Other</td>
</tr>
<tr>
<td>SimpleTypes</td>
<td>Planned, Actual, Estimated</td>
</tr>
<tr>
<td>SimpleDateSubType</td>
<td>Submitted, Approved</td>
</tr>
<tr>
<td>OrganizationType</td>
<td>Customer, Performing, Sub-Contractor, Other</td>
</tr>
<tr>
<td>SuccessItemType</td>
<td>Business Benefit, Success Criterion, Success Factor</td>
</tr>
<tr>
<td>WorkComponentType</td>
<td>Project, Sub-Project, Phase, Workpackage, Other</td>
</tr>
<tr>
<td>TaskType</td>
<td>Project, Sub-Project, Phase, Workpackage, Milestone, Activity, Other</td>
</tr>
<tr>
<td>RequirementType</td>
<td>Main, Minor, Other</td>
</tr>
<tr>
<td>DeliverableType</td>
<td>Documentation, Software Application, Software Infrastructure, Hardware, Network Infrastructure, Other</td>
</tr>
<tr>
<td>DeliverableStatus</td>
<td>Not Submitted, In Review, Approved, Rejected, Other</td>
</tr>
<tr>
<td>MilestoneType</td>
<td>Main, Minor, Other</td>
</tr>
<tr>
<td>MilestoneStatus</td>
<td>Reached, Not Reached</td>
</tr>
<tr>
<td>StakeholderType</td>
<td>Organization, Business Unit, Team, Person, External System, Other</td>
</tr>
<tr>
<td>StakeholderRole</td>
<td>Project Owner, Project Sponsor, Project Manager, Team Leader, Team Member, Functional Manager, Other</td>
</tr>
<tr>
<td>StakeholderOrganization</td>
<td>Customer, Performing, Sub-Contractor, Other</td>
</tr>
<tr>
<td>QualityType</td>
<td>QA, QC, QP</td>
</tr>
<tr>
<td>DependencyType</td>
<td>Finish-to-Finish, Finish-to-Start, Start-to-Finish, Start-to-Start</td>
</tr>
<tr>
<td>RateType</td>
<td>HR, Equipment, Other</td>
</tr>
<tr>
<td>RateMeasure</td>
<td>Hourly, Daily, Monthly, Yearly, Other</td>
</tr>
<tr>
<td>Currency</td>
<td>USD, EUR, GBP, INR, AUD, CAD, SGD, CHF, MYR, JPY, CNY</td>
</tr>
<tr>
<td>IssueType</td>
<td>Bug, Enhancement, Change, Problem, Task, Other</td>
</tr>
<tr>
<td>PriorityType</td>
<td>Very High, High, Medium, Low, Very Low</td>
</tr>
<tr>
<td>IssueStatus</td>
<td>Opened, Approved, Rejected, On Progress, Cancelled, Closed</td>
</tr>
<tr>
<td>MetricType</td>
<td>HR, Equipment, Other</td>
</tr>
<tr>
<td>RiskType</td>
<td>Scope, Scheduling, Resource, Technology, Other</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>SWOTType</td>
<td>Strength, Weakness, Opportunity, Threat</td>
</tr>
<tr>
<td>ImpactLevel</td>
<td>Very High, High, Medium, Low, Very Low</td>
</tr>
<tr>
<td>Strategy</td>
<td>Avoid, Transfer, Mitigate, Accept, Exploit, Enhance</td>
</tr>
<tr>
<td>RiskStatus</td>
<td>Identified, Assessed, Planned, On Process, Closed, Not Occurred, Rejected</td>
</tr>
<tr>
<td>ResponsabilityType</td>
<td>X, D, d, P, T, I, C, A</td>
</tr>
<tr>
<td>CriterionAssessmentStatus</td>
<td>Yes, No</td>
</tr>
<tr>
<td>ExpressionOperator</td>
<td>&gt;, &lt;, =, &gt;=, &lt;=</td>
</tr>
<tr>
<td>FlowType</td>
<td>Technical, Management, Steering, Other</td>
</tr>
<tr>
<td>FlowFrequency</td>
<td>Once, Weekly, Bi-Weekly, Monthly, As-Needed</td>
</tr>
<tr>
<td>FlowMedium</td>
<td>Face-To-Face, Conference Call, Email, Other</td>
</tr>
<tr>
<td>MeetingType</td>
<td>Technical, Management, Steering, Other</td>
</tr>
<tr>
<td>MeetingFrequency</td>
<td>Once, Weekly, Bi-Weekly, Monthly, As-Needed</td>
</tr>
<tr>
<td>MeetingMedium</td>
<td>Face-To-Face, Conference Call, Email, Other</td>
</tr>
<tr>
<td>ExpenseType</td>
<td>HRs, Travels, Equipments, Materials, Services, Suppliers, General, Capital, Total, Other</td>
</tr>
</tbody>
</table>
# Appendix D

## Validations

Table 10. PSL Custom Validations and respective Error Messages

<table>
<thead>
<tr>
<th>Validations</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check if provided day is between 1 and 31.</td>
<td>Day should be between 1 and 31.</td>
</tr>
<tr>
<td>Check if provided day is below or equal to 29 for February on a leap year.</td>
<td>Feb YEAR only has 29 days.</td>
</tr>
<tr>
<td>Check if provided day is below or equal to 28 for February on a common year.</td>
<td>Feb YEAR only has 28 days.</td>
</tr>
<tr>
<td>Check if provided day is below or equal to 30 for April, June, September and November.</td>
<td>MONTH YEAR only has 30 days.</td>
</tr>
<tr>
<td>Check if price and cost currencies are the same for a specific Budget.</td>
<td>Price currency and cost currency should be the same.</td>
</tr>
<tr>
<td>Check if multiple packages of the same type do not exist.</td>
<td>Only one TYPE package per file is allowed.</td>
</tr>
<tr>
<td>Check inexistence of Schedules of type Estimated on a Project.</td>
<td>Project must have at maximum 2 schedules: planned and actual. Estimated schedules are not allowed.</td>
</tr>
<tr>
<td>Check if Project has at maximum 2 Schedules.</td>
<td>Project must have at maximum 2 schedules: planned and actual.</td>
</tr>
<tr>
<td>Check if Project has only one Schedule of type Planned and one of type Actual.</td>
<td>Project must only have one schedule of type TYPE.</td>
</tr>
<tr>
<td>Check inexistence of Budgets of type Estimated on a Project.</td>
<td>Project must have at maximum 2 budgets: planned and actual. Estimated budgets are not allowed.</td>
</tr>
<tr>
<td>Check if Project has at maximum 2 Budgets.</td>
<td>Project must have at maximum 2 budgets: planned and actual.</td>
</tr>
<tr>
<td>Check if Project has only one Budget of type Planned and one of type Actual.</td>
<td>Project must only have on budget of type TYPE.</td>
</tr>
<tr>
<td>Check if starting date is before ending date.</td>
<td>End date must be after (or equal to) start date.</td>
</tr>
<tr>
<td>Check if value and total cost use the same currency for a specific Year</td>
<td>Value currency and Total Cost currency should be the same.</td>
</tr>
<tr>
<td>Check if Schedule is of type Planned for a specific Task.</td>
<td>Task schedule must be of type Planned.</td>
</tr>
<tr>
<td>Check if only Workpackages are being used on the RAM.</td>
<td>Only Workpackages are considered for the RAM.</td>
</tr>
<tr>
<td>Check if workComponent attribute from Deliverable is of type Workpackage.</td>
<td>Workpackage expected.</td>
</tr>
<tr>
<td>Check if workComponent attribute from Milestone is of type Workpackage.</td>
<td>Workpackage expected.</td>
</tr>
<tr>
<td>Check if opened date is before validated date for a specific Issue.</td>
<td>Validated date must be after (or equal to) opened date.</td>
</tr>
<tr>
<td>Check if validated date is before closed date for a specific Issue.</td>
<td>Closed date must be after (or equal to) validated date.</td>
</tr>
<tr>
<td>Check if maximum value is higher than minimum value for a Range.</td>
<td>Maximum value should be higher than minimum.</td>
</tr>
<tr>
<td>Check if created date is before lastUpdate date for a specific Risk.</td>
<td>Last updated date must be after (or equal to) created date.</td>
</tr>
<tr>
<td>Check if impact is equal to consequence times probability for a Risk.</td>
<td>Impact should be equal to consequence * probability.</td>
</tr>
<tr>
<td>Check if probability is greater than 0 and less than or equal to 100% for a Risk.</td>
<td>Probability should be greater than 0 and less than or equal to 100%.</td>
</tr>
<tr>
<td>Check if consequence value is between 0 and 10 (both included).</td>
<td>Consequence should be greater than or equal to 0 and less than or equal to 10.</td>
</tr>
<tr>
<td>Check if result is of type Percentage given that expression is of type Percentage on Criterion.</td>
<td>Result should be of type percentage since expression is of type percentage.</td>
</tr>
<tr>
<td>Check if duplicate Resources do not exist on RAM.</td>
<td>Resource RESOURCE is repeated.</td>
</tr>
<tr>
<td>Check if duplicate Spectators do not exist on an Audience.</td>
<td>Spectator SPECTATOR is repeated.</td>
</tr>
<tr>
<td>Check if no comma was introduced after last Spectator of an Audience.</td>
<td>Delete comma.</td>
</tr>
<tr>
<td>Check inexistence of hierarchy cycles on Expenses.</td>
<td>Cycle in hierarchy of expense EXPENSE.</td>
</tr>
<tr>
<td>Check if an Expense is not part of itself.</td>
<td>An expense cannot be part of itself.</td>
</tr>
<tr>
<td>Check inexistence of hierarchy cycles on partOf from Stakeholders.</td>
<td>Cycle in hierarchy of stakeholder STAKEHOLDER.</td>
</tr>
<tr>
<td>Check if a Stakeholder does not belong to himself.</td>
<td>A stakeholder cannot belong to himself.</td>
</tr>
<tr>
<td>Check inexistence of hierarchy cycles on reportTo from Stakeholders.</td>
<td>Cycle in hierarchy of stakeholder STAKEHOLDER.</td>
</tr>
<tr>
<td>Check if a Stakeholder does not report to himself.</td>
<td>A stakeholder cannot report to himself.</td>
</tr>
<tr>
<td>Check if a Milestone has only one Date of type Planned, one of type Actual and one of type Estimated.</td>
<td>A milestone must only have one date of type TYPE.</td>
</tr>
<tr>
<td>Check if a Milestone has at maximum 3 Dates.</td>
<td>A milestone must have at maximum 3 dates: planned, actual and estimated.</td>
</tr>
<tr>
<td>Check if Milestone dates do not have subTypes.</td>
<td>Milestone dates should not have subtypes.</td>
</tr>
<tr>
<td>Check if a Deliverable has only one Date of type Planned, one of type Actual and one of type Estimated.</td>
<td>A deliverable must only have one date of type TYPE.</td>
</tr>
<tr>
<td>Check if a Deliverable has at maximum 6 Dates.</td>
<td>A deliverable must have at maximum 6 dates of type planned, actual and estimated, and sub-types submitted and approved.</td>
</tr>
<tr>
<td>Check if Deliverable dates have subTypes.</td>
<td>Deliverable dates should have subtypes.</td>
</tr>
<tr>
<td>Check inexistence of hierarchy cycles on partOf from Tasks.</td>
<td>Cycle in hierarchy of task TASK.</td>
</tr>
<tr>
<td>Check if a Task is not part of itself.</td>
<td>A task cannot be part of itself.</td>
</tr>
<tr>
<td>Check inexistence of hierarchy cycles on Tasks of type Workpackage.</td>
<td>Cycle in hierarchy of workpackage WORKPACKAGE.</td>
</tr>
<tr>
<td>Check if a Task does not depend on itself.</td>
<td>A task cannot depend on itself.</td>
</tr>
<tr>
<td>Check inexistence of hierarchy cycles on Workpackage.</td>
<td>Cycle in hierarchy of task TASK.</td>
</tr>
<tr>
<td>Workpackages cannot be part of another workpackages.</td>
<td>Cycle in hierarchy of workpackage WORKPACKAGE. Workpackages cannot be part of another workpackages.</td>
</tr>
<tr>
<td>Check inexistence of hierarchy cycles on dependsOn from Tasks.</td>
<td>Cycle in hierarchy of work component WORKCOMPONENT.</td>
</tr>
<tr>
<td>Workpackages cannot be part of another workpackages.</td>
<td>Cycle in hierarchy of work component WORKCOMPONENT.</td>
</tr>
<tr>
<td>Check if a Task does not depend on itself.</td>
<td>A task cannot depend on itself.</td>
</tr>
<tr>
<td>Check if a Task does not depend on itself.</td>
<td>A task cannot depend on itself.</td>
</tr>
<tr>
<td>Check if a WorkComponent is not part of itself.</td>
<td>A work component cannot be part of itself.</td>
</tr>
<tr>
<td>Check if a WorkComponent is not part of itself.</td>
<td>A work component cannot be part of itself.</td>
</tr>
<tr>
<td>Check if a WorkComponent has only one Schedule of type Planned, one of type Actual and one of type Estimated.</td>
<td>A work component must only have one schedule of type TYPE.</td>
</tr>
<tr>
<td>Check if a WorkComponent has at maximum 3 Schedules.</td>
<td>A work component must have at maximum 3 schedules: planned, actual and estimated.</td>
</tr>
<tr>
<td>Check if Tasks and WorkComponents with the same ID, have the same nameAlias.</td>
<td>Task TASK and work component WORKCOMPONENT represent the same item, therefore they should have the same name.</td>
</tr>
<tr>
<td>Check if Tasks and WorkComponents with the same ID, have the same type.</td>
<td>Task and work component NAME represent the same item therefore they should have the same type.</td>
</tr>
<tr>
<td>Check if Tasks and WorkComponents with the same ID, are partOf the same WorkComponent.</td>
<td>Task and work component NAME represent the same item therefore they should be part of the same work component.</td>
</tr>
<tr>
<td>Check if Tasks and WorkComponents with the same ID, are partOf the same WorkComponent.</td>
<td>Task and work component NAME represent the same item therefore they should be part of the same work component.</td>
</tr>
<tr>
<td>Check if duration of a Task corresponds to the days between start and end dates of that Task.</td>
<td>Duration value should be DURATION instead.</td>
</tr>
<tr>
<td>Check if measure from an Expense is equal to the measure of the Rate referenced by that Expense.</td>
<td>Measure should be equal to RATE’s measure.</td>
</tr>
<tr>
<td>Check if totalCost of an Expense is equal to the sum of totalCosts of all the Years of that Expense.</td>
<td>Total value should be TOTAL TOTALCURRENCY instead.</td>
</tr>
<tr>
<td>Check if totalCost of a Year is equal to times quantity.</td>
<td>Total value should be TOTAL TOTALCURRENCY instead.</td>
</tr>
<tr>
<td>Check if all SimpleYears of a Rate use the same currency.</td>
<td>All currencies should be equal for each rate.</td>
</tr>
<tr>
<td>Check if all Years of an Expense use the same currency.</td>
<td>All currencies should be equal for each expense.</td>
</tr>
<tr>
<td>Check if only one X is used on each RAM line.</td>
<td>Only one X is allowed per line (i.e. per work package).</td>
</tr>
<tr>
<td>Check if only one P is used on each RAM line.</td>
<td>Only one P is allowed per line (i.e. per work package).</td>
</tr>
<tr>
<td>Check if an actor with responsibilities of type D, T or I, do not have other responsibilities.</td>
<td>If an actor has a responsibility of type TYPE he cannot have other responsibilities.</td>
</tr>
</tbody>
</table>