

UNIVERSIDAD POLITÉCNICA DE MADRID



POLITÉCNICA

ETS Ingenieros Informáticos

Exploiting software development methods and tools in ontology engineering

MASTER THESIS

Author:

Alba Fernández Izquierdo

Supervisor:

Oscar Corcho

Master in Artificial Intelligence

July 2016

Abstract

Ontological Engineering is the discipline that study the activities that have to be carried out during the ontology development, its life cycle and all the methodologies, tools and languages for building ontologies. Distributed ontology engineering early emerged to refer to those cases where such a development process was done collaboratively by experts working from geographically distributed locations.

In this thesis we propose an agile framework for distributed ontology engineering based on some of the current software development practices. We hypothesize that by applying these techniques, which have been widely used and tested by software developers over the years, the ontology development process can be improved and the results can be optimized. In this context, we have designed a development process which is build on specific technologies and describe the entire process of ontology creation. This development process incorporates the following ideas from agile software development: *continuous integration and evaluation*, *frequent releases*, *roles* and *continuous changes*. We have also designed an evaluation and a communication system based on agile strategies that help ontology developers to improve ontology development. This framework was validated using ontologies extracted from the GitHub account of the Spanish thematic network on Open Data for Smart Cities¹.

¹<https://github.com/opencitydata>

Resumen

La ingeniería ontológica se refiere al estudio de las actividades que conciernen el desarrollo de la ontología, su ciclo de vida y las metodologías, herramientas y lenguajes para construir las ontologías. La ingeniería ontológica distribuida ha surgido para referirse a ver estos casos en que un proceso de este tipo de desarrollo se lleva a cabo en colaboración por expertos que trabajan desde ubicaciones distribuidas geográficamente.

En esta tesis de máster, proponemos la creación de un framework para la ingeniería ontológica distribuida basado en algunas de las prácticas actuales en el desarrollo de software. Nosotros hemos supuesto que aplicando las técnicas de estas metodologías, las cuales ya están muy probadas y utilizadas desde hace años por los desarrolladores de software, el proceso de desarrollo de ontologías puede mejorarse y optimizar sus resultados. En este contexto, hemos diseñado un proceso de desarrollo que describe el proceso completo de creación de ontologías. Este proceso de desarrollo incluye las siguientes ideas ágiles del desarrollo ágil de software: integración continua, cambios continuos, roles y entregas frecuentes. También hemos diseñado un sistema de comunicación y otro de evaluación basado en estrategias ágiles que pueden mejorar el proceso de desarrollo de ontologías. Para validar este framework propuesto, hemos utilizado ontologías extraídas de la cuenta de GitHub de la red temática de Open Data para Smart Cities².

²<https://github.com/opencitydata>

Contents

Abstract	III
Resumen	V
List of Figures	IX
List of Tables	XI
1. Introduction	1
1.1. Motivation	1
1.2. Objectives	4
1.3. Structure of the document	4
2. State of the Art	7
2.1. Review of tools that support distributed development	7
2.2. Comparison criteria	10
2.3. Discussion	13
2.4. Limitations and improvements	17
3. Design	19
3.1. Tools to support the framework	19
3.2. Agile practices in ontology engineering	21
3.3. Modeling the agile scenario in ontology engineering	23
3.3.1. Roles	23
3.3.2. Ontology development process	23
3.4. Agile techniques adapted to ontology engineering	30
3.4.1. Communication system exploiting Github issues	30
3.4.2. Evaluation system	34
3.4.2.1. Unit test	35
3.4.2.2. Acceptance test	42
4. Implementation and experimentation	45
4.1. Implementation	45
4.1.1. Limitations	46

4.2. Experimentation	46
4.2.1. Example 1: “Turismo” ontology	47
4.2.2. Example 2: “Callejero” ontology	52
5. Conclusions and future work	59

List of Figures

3.1. OnToology Architecture, extracted from [1]	20
3.2. Ontology development process	28
3.3. Development process with roles	29
3.4. Development process with outputs	29
3.5. Modification procedure	30
3.6. Kanban example	33
3.7. OOPS! pitfalls classification, extracted from [26]	37
3.8. Query-Result example	44
4.1. Implementation structure	46
4.2. List of issues	47
4.3. Modeling issue	48
4.4. Inference issue	48
4.5. List of issues	51
4.6. Content of the issue	52
4.7. List of issues	52
4.8. Modeling issue	53
4.9. Inference issue	53
4.10. List of issues	57
4.11. Content of the issue	57

List of Tables

2.1. Comparison table (+ indicates that it is a supported feature, - indicates that it is not a supported feature)	16
3.1. Actions and roles in ontology development	31
3.2. Actions and labels in ontology development	32
3.3. Selected pitfalls	41
3.4. Enhancement pitfalls	42
4.1. User story 03. Influx of people in an area	49
4.2. User story 01. Information about a touristic area	49
4.3. User story 02. Tours in tourist areas	50
4.4. User story 18. Accessibility of a place	50
4.5. User story 04. Information about cultural events	51
4.6. User story 01. Information about streets	54
4.7. User story 12. Coordinates of a building	54
4.8. User story 07. Influx of people in a section of a street	55
4.9. User story 14. Information about the type of the buildings	55
4.10. User story 02. Information about streets and sections of streets	56

Chapter 1

Introduction

The Semantic Web introduces a new generation of the Web by establishing a layer of machine-understandable data. Ontologies play an important role in the Semantic Web. They aim at capturing domain knowledge in a generic way and provide a commonly agreed understanding of a domain, which may be reused or shared, as it describes the work in [12]. Due to the popularity of the Semantic Web, the interest in ontology engineering is increasing and ontologies are becoming more important for the development of information systems. In the last years, ontologies are not developed anymore by groups of reduced size working on their offices and having access to some domain experts, but in wider geographically distributed environments. This means that there are several experts, with different and complementary skills, involved in collaboratively building the same ontology. In order to promote distributed ontology engineering, methodologies are needed to organize the tasks that need to be performed.

1.1. Motivation

Ontology engineering refers to the study of the "activities that concern the ontology development process, the ontology life cycle, and the methodologies, tools and languages for building ontologies"[13]. In a distributed ontology engineering scenario process, methods and tools are explicitly designed to support a decentralized group of stakeholders or community of interest. A distributed ontology engineering process typically starts with an analysis of the domain to be captured by the ontology. Once the domain is clear, the conceptual model is implemented in a formal knowledge representation language such as OBO and OWL. In order to respond to changes, the community revises and extends the ontology continuously, and releases new versions of

it. In order to facilitate the evolution of the ontology through parallel work instruments for resolving conflicts are required.

Ontology engineering also includes complementary activities such as the documentation phase and the evaluation phase. The documentation phase produces a human-readable documentation that allows users understand the OWL or RDFs file produced during the implementation, and the evaluation phase evaluates the ontology in two different ways: by checking whether the requirements are answered properly and by checking whether the ontology follows design patterns and well established practices for its implementation or not. As the aim of the vocabularies and ontologies is normally to share the model for its reuse, the ontology needs to be published on the web with its documentation.

In addition to this, there are some observations about ontologies made by [6] that can help to understand how to build them:

- Ontology building as a learning process. The construction process is a learning process in which the involved individuals deepen their understanding of the real world and of a vocabulary to describe it.
- Formality and complexity of use as a barrier. The individual should take part in community activities by lowering the barriers: informal, lightweight, easy-to-use, and easy-to-understand.
- Continuous evolution in work processes. Ontology building is not supposed to be a one-time activity of a committee of experts, but rather a sustainable process of continuous evolution.

After analyzing how an ontology has to be built in a distributed environment, we have concluded that it has important analogies with the development of a product in software engineering (e.g documentation, evaluation, versioning phases). In fact, this was already suggested by early ontology engineering methodologies, such as Methontology[9], which were inspired by existing software development methodologies. Because of this, we think that the adoption of software development methods and tools, which are widely used and tested over the years, to this type of ontology engineering can help systematizing and structuring the development process by establishing a set of activities and guidelines that can support all the necessities mentioned before. There are two types of software methodologies that can be analyzed in order to integrate their strategies in ontology engineering:

Traditional Methodologies Traditional methodologies of software development[23] are characterized by a sequential series of steps like requirement definition, planning, building, testing and deployment. First, the client requirements are carefully documented to the fullest extent. Then, the general architecture of the software is visualized and the coding begins. Finally the various types of testing and the final deployment come in. These methodologies are very rigid hence such development does not work well in settings where requirements are uncertain and change frequently. Because of the fact that we need flexible processes and a continuous evolution of the work, these methodologies are not efficient in our environment.

Agile Methodologies Agile methodologies[3][7] are an alternative to traditional project management. They help teams respond to unpredictability through incremental, iterative work cadences known as sprints. They give more importance to adaptability and constant compatibility testing. Agile methodologies follow these principles:

- Individuals and interactions. In agile development, self-organization and motivation are important, as well as interactions like co-location and pair programming.
- Working software. Demo working software is considered the best means of communication with the customer to understand their requirements, instead of just depending on documentation.
- Customer collaboration. As the requirements cannot be gathered completely in the beginning of the project due to various factors, continuous customer interaction is very important to get proper product requirements.
- Responding to change. Agile development is focused on quick responses to change and continuous development.

With this flexibility, in particular the fast and efficient reactions to changing prerequisites, we consider this methodology useful for ontology construction. Its principles, which we consider to be helpful for ontology development, can also be integrated in ontology development in order to optimize development processes. There are already some agile software oriented approaches that have been adapted to the ontology development process, such as RapidOWL[2] or XP.K[19]. Both of these methodologies are lightweight agile methodologies that append agile practices to Knowledge Engineering,

but they only focus on implementation and leave aside the rest of ontology engineering phases (e.g. documentation, evaluation).

1.2. Objectives

The main objective of this work is to create an agile framework to support distributed ontology engineering. This framework will be built on specific technologies: OnToology[1] and Travis CI¹. OnToology addresses several steps of the ontology development process, including documentation, representation, evaluation and publication, and Travis CI provides continuous integration support. This framework will organize the activities and tasks to be performed by different distributed and collaborative actors during the ontology development process and will exploit agile practices and tools which, as it is mentioned in Section 1.1, can help in making the process more efficient and in obtaining better results in the vocabulary development process.

To reach this objective we will follow the next steps:

- Create a state of the art on tools that support ontology development, in order to compare their potential and limitations regarding the development process and agile tasks.
- Analyze agile software development strategies and approaches and identify the most useful processes and techniques for ontology development.
- Design an ontology development process, supported by OnToology, that includes all these agile strategies.
- Validate the development process using real vocabularies created.

1.3. Structure of the document

This document is organized as follows:

- Chapter 2 analyzes the state of the art in tools that support collaborative ontology development. First, a review of tools is presented and then the comparison criteria between these tools are described. Afterwards, we present the results obtained after comparing the tools using the criteria.

¹<https://travis-ci.org/>

-
- Chapter 3 describes the framework designed for ontology development based on agile techniques.
 - Chapter 4 describes the implementation of agile techniques extracted from the framework and the experimentation made with them in real use cases.
 - Chapter 5 draws some conclusions and presents ideas for future work.

Chapter 2

State of the Art

In the last few years different tools have been proposed to support the ontology development process. Each of these tools provide different elements for edition, version control or evaluation of the ontologies created, which are key tasks throughout the ontology creation. In this chapter we summarize the main tools that support collaborative development and we also compare them to identify their potential and limitations.

2.1. Review of tools that support distributed development

In this section, we present the most relevant ontology engineering tools that support the distributed edition of ontologies. Our primary focus is on those that support more activities related to agile development, because these are the tasks which we are interested in.

ContentCVS

ContentCVS [17] was developed by Universitat Jaume I and Oxford University Computing Laboratory as a downloadable Protégé 4 plugin. This tool is focused on version control and the changes made to the ontology, in addition to the error detection and solution among versions. ContentCVS uses the notion of structural equivalence to compare two ontologies. Because of that, in order to find the differences between two ontologies ContentCVS only uses the OWL structure. This tool also makes suggestions to the user so that he can resolve conflicts among versions.

Moki

Moki[11][10] was developed through a collaboration between the DKM Research Unit FBK and the Know-Center in Austria as a collaborative ontology development tool based on MediaWiki¹. This tool associates each entity of the ontology and process model to a wiki page, which contains both unstructured and structured information. The unstructured part contains text written following the standard MediaWiki markup format. On the other hand, the structured part contains knowledge stored according to the modeling language adopted. All the information can be edited by users. Another feature of Moki is the information access. This tool allows users to access information using different access modes, depending on what the user wants to know. Moki also integrates evaluation functionalities like models checklist and quality indicators.

Neologism

Neologism[8], developed by the Digital Enterprise Research Institute, is a web-based vocabulary editor and publishing system. This tool also provides an offline file-based model, where ontology files are stored on the local user's computer. Neologism can also provide an automatic diagram creation that shows the vocabulary's classes and their relationships. The vocabulary is created by publishing a description of its terms using HTML or formal using RDFS/OWL language, where the classes and properties are identified by URIs.

OntoMaven

OntoMaven[24], developed at Freie Universität Berlin, is a tool for distributed ontology engineering that extends Apache Maven and adapts it. It extends an OntoMaven POM file to describe the ontology project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. OntoMaven downloads ontologies and plug-ins for ontology engineering from OntoMaven repositories. It includes versioning, dependency management for ontologies, documentation, testing and

¹<https://www.mediawiki.org/wiki/MediaWiki>

IDEs/APIs tasks for download, transformation integration/import compilation, installation and deployment.

OnToology

OnToology[1], developed at Universidad Politécnica de Madrid, is a web-based tool designed to work with GitHub and to automate part of the ontology development process in distributed environments. OnToology uses Git repositories to create the vocabulary, and produces the documentation, evaluation and publication of the ontology in the user's repository.

OntoWiki

OntoWiki[30], developed at University of Leipzig, is an application from Semantic Wiki, an extensible tool for managing structured information in a collaborative, web-based environment. This wiki-based tool allows navigating and visualizing RDF-based Knowledge Bases. OntoWiki produces and consumes Linked Data. In order to help human users to access the information, OntoWiki allows to create different views on data, such as tabular representations or maps.

Owl2vcs

Owl2vcs[33], developed at Tomsk Polytechnic University, facilitates the distributed ontology development using version control. Owl2vcs allows to find which entity is deleted, modified or added. It takes into account ontology format changes and import changes. This tool also performs a three-way merge of ontologies to detect changes in them and helps the user to resolve conflicts between versions.

SVoNt

SVoNt[22], developed at Freie Universität Berlin, is a SVN-based approach for versioning of W3C OWL ontologies. This system consists of an extended Subversion server and a special SVoNt client. It reuses existing functionalities like logging, authentication and versioning features of Apache Subversion, and also permits integrating the SVoNt server into existing Subversion environments, such as Eclipse. SVoNt provides new features as consistency check and a change-detection module that includes the basic ontology comparison

algorithm and the CEX logical diff. The main characteristic is that it focuses on the visualization of revisions of changes on the concept level.

VocBench

VocBench[28], which was created by ART Group and the University of Rome, is an open source web application for editing thesauri complying with the SKOS and SKOS-XL standards. VocBench allows for collaborative management of the overall editorial workflow, by introducing different roles with specific competencies, and has features for content validation and publication. Furthermore, it provides a full history of changes and a SPARQL query service.

VoCol

VoCol[25][14], developed at University of Bonn, was designed as a tool to help collaborative vocabulary development inspired by agile software and content development methodologies. It uses Git repositories to create the vocabulary. This tool includes project management, validation, documentation and visualization generation components.

Web-Protégé

Web-Protégé[31], developed by the Stanford Center for Biomedical Informatics Research, is a web-based lightweight ontology editor for the Web that uses Protégé as its backend. Web-Protégé was created as a a collaboration platform that is easily customizable for different users and projects' settings and provides change tracking and revision history. It provides a chat service and the functionality for annotating the vocabulary terms.

2.2. Comparison criteria

In this section we compare the aforementioned tools according to how they manage the main activities for an agile collaborative development[15]. This development needs to support: evaluation, version control system, automatic generation of documentation, multi-mode access to information, continuous

integration, publishing tasks and communication system.

Evaluation

Because of the amount of people collaborating in a vocabulary, it is important to check its consistency, not only semantically but also syntactically, and to check if the ontology follows design patterns and well established practices for its implementation. The ontology can be also evaluated by checking whether its requirements are answered properly.

Version control and conflict resolution

While working in a distributed scenario it is necessary that the tool being used provides version control so the user can track all the work that is being made. It should show which classes and properties are added, removed or modified, enabling the contributors and users to see how the vocabulary has evolved over time. It could also be useful an ability to roll-back to a particular state of the ontology. Distributed development of vocabularies should respond to the evolution of the knowledge domain so it is necessary to support the detection and documentation of differences among versions. It is also important that the tool can detect conflict errors and solve them if several people are working over the same document.

Labeling versions

Release and new versions of vocabularies should be labeled appropriately, so the users and the machines can always know the status of the vocabulary, and carry out tasks according to it. This is important in agile development, where new releases and new versions are frequent and guide the developers to new activities.

Documentation

In these kind of tools the automatic generation of documentation is useful. With this task, the user can obtain a clear description of the vocabulary that

can be published on the web. This documentation can be shown using different types of representation as tables or boxes. The only requirement is that it has to be organized and easy to understand by the user.

Multi-mode access

Multi-mode access and roles are important elements when users are working collaboratively. The information can be shown using different views or permissions depending on the user that access to the project. This feature is useful especially because of the fact that there are a lot of different user profiles working in an ontology development process.

Publishing

In ontology development the user may want to create the vocabulary and publish it on the web. Due to this, these type of tools must create the vocabulary by publishing a description of its terms using HTML or RDFS/OWL language. The classes and properties are identified by URIs. With this publication, the vocabulary can be accessible by everyone.

Continuous integration

Continuous integration is a software engineering practice that consists of merging and testing all developer working copies to a shared mainline several times a day. With this practice, it can be checked every day that there are no important errors in the project. Due to the fact that there are a lot of people collaborating in a vocabulary, this feature is useful to be able to deliver at any moment a product version without errors.

Communication support

The distributed development of vocabularies is based on finding consensus between team members. During the entire development life cycle it is essential to share ideas, agreements and discussions among the heterogeneous teams. Because of this idea, it is necessary for the tools to have a communication mechanism that allows the users to maintain and organize their discussions.

2.3. Discussion

In this section we specify the results obtained after analyzing each of the tools described in the previous section through the comparison criteria. It has been created three tables that summarize these results.

Results for evaluation

Table 2.1 shows that evaluation is supported in OnToology, SVoNt, VocBench, VoCol and OntoMaven. OnToology makes an evaluation using OOPS!^[26], which detects common pitfalls during the ontology development. SVoNt makes a simple semantic and syntactic consistency check, while VocBench only has a GUI that prevents syntax errors. VoCol provides a syntactic validation system using Rapper² and JenaRiot³ that can work while the user is editing a document. OntoMaven has a testing phase which supports the W3C OWL test cases syntax checker, consistency checker, and entailment test. The produced test results are compliant to the W3C recommendation and the created test reports show if the ontology model is consistent, inconsistent or if the result is unknown. No one of these tools evaluate whether the requirements are answered or not. Only VoCol provides a SPARQL query service where the developers can make queries over the ontologies and check their results.

Results for version control

All the tools mentioned in the previous section have a version control system and conflict resolution, due to this is a key task when users work in a distributed environment. The more interesting version control systems are those included in SVoNt, OntoMaven and in VoCol. SVoNt can track conceptual changes among versions, which allows to the users to know which entity has been modified, added or deleted. These conceptual differences are very useful in ontology evolution. The approach in OntoMaven is based on the ontology versioning tool SVont. On the other hand, the main feature of VoCol version control system, which also integrates SVont, is the timeline that allow to the user to see the structural changes and when they were made.

²<http://librdf.org/raptor/>

³<https://jena.apache.org/documentation/io/>

Results for labeling versions

As it is mentioned before, labeling versions can be useful for developers to know the actual status of the vocabulary. Analyzing these tools and the Table 2.1, we conclude that only OnToology, VoCol and Moki support this service. VoCol and OnToology uses GitHub and its tags to label the versions while Moki has its own labeling system, which treats the version tag as a property of the vocabulary.

Results for documentation

As it is shown in Table 2.1 Neologism, OnToology, OntoMaven and VoCol can generate documentation automatically. Neologism generates a diagram that shows the vocabulary's classes and their relationships. OntoMaven uses the SpecGen extension for automated concept grouping, in order to create the technical and user documentation in an OntoMaven plugin. VoCol uses schemaorg publication engine⁴ and Widoco⁵ to generate the vocabulary description, so the users can choose the tool they prefer. Finally, OnToology provides the most complete documentation system which integrates AR2DTool to create taxonomy and entity relationship diagrams of the ontology and Widoco, which creates the ontology description.

Results for multi-mode access

OnToology, VoCol, Moki and VocBench support heterogeneous groups of stakeholders with various roles. VoCol and OnToology roles are supported by repository hosting platform (Git). Moki users can access the ontological and procedural knowledge contained in MoKi using three different access modes: one mode, the unstructured access mode to access the unstructured part of a MoKi page, and two different modes, the fully-structured access mode and the lightly-structured access mode, to access the structured part. VocBench allows the users upon registration to indicate the vocabulary they are interested in and the roles they want to cover.

Results for publishing

⁴<https://github.com/schemaorg/schemaorg/>

⁵<https://github.com/dgarijo/Widoco>

As it is shown in Table 2.1 Neologism, OnToology, VoCol and OntoMaven give publishing services. VoCol delivers the vocabulary through a web server configured to perform content negotiation according to the best practices for publishing vocabularies. Neologism provides an instant web-based publishing, handling URI management and content negotiation, and OntoMaven uses Maven plugins to publishing automated reports. OnToology, even it does not support the publishing process itself, it produces a bundle with the documentation that is ready to be deployed on a server.

Results for continuous integration

Continuous integration is an essential task in agile methodologies but none of these tools include this feature. Only VoCol provides an analogous process to continuous integration through its continuous vocabulary component integration and verification functions while the user is working.

Results for communication support

The majority of tools provide some mechanism for communication between users. Vocol and OnToology can support communication between users due to GitHub, which provides the service. Otherwise, Web-Protégé provides a chat service, VocBench gives a service that allows to submit comments from developers and in Moki users can interact with each other using using the discussion Semantic Media Wiki's built-in functionality. OntoWiki, which is also based on wikis, provides a commenting system where all statements presented to the user may be annotated, commented, and their usefulness can be rated.

TABLE 2.1: Comparison table (+ indicates that it is a supported feature, - indicates that it is not a supported feature)

Features	Tools											
	Content	CVS	Moki	Neologism	OntMaven	OnToology	OntoWiki	Owl2vcs	SVont	VocBench	VoCol	Web-Protégé
Evaluation	-	+	-	+	+	+	-	-	+	+	+	+
Documentation	-	+	-	+	+	+	+	+	-	+	+	-
Version control	-	-	+	-	+	+	-	-	-	-	+	-
Labelling versions	+	+	+	+	+	+	+	+	+	+	+	+
Roles	-	-	-	-	-	-	-	+	+	+	+	-
Continuous integration	-	-	-	-	-	-	-	-	-	-	-	-
Publishing	-	-	-	+	+	+	-	-	-	-	+	-
Communication support	-	+	-	-	-	+	-	-	+	+	+	+

2.4. Limitations and improvements

All of the analyzed tools support some of the activities needed in agile ontology development, such as version control or communication systems. However, the agile practices are not integrated enough in ontology development. Therefore ontology development tools cannot take advantage of the benefits that this methodology would provide. The main limitations identified in the analyzed tools are related to:

- **Testing and continuous integration.** The idea of continuously integrating is to find bugs quickly, thus giving each developer feedback on their work. This is an essential practice in agile methodologies because it helps the system stay robust enough that customers can use it whenever they like.
- **Publication.** In an ontology engineering scenario, publication is the end of the development process. This publication, which is analogous to the deployment process in agile methodologies, makes the ontology accessible for everyone.
- **Versioning.** Versioning is an important concept in agile methodologies, due to the fact that it is important to know the status of the project in every moment so that developers can carry out different tasks according to it. Because of this, the project needs labels according to each version or release and deltas among version to be informed about the changes made to the ontology.

These elements should be taken into account due to the fact that they can help to improve the ontology development process and optimize the results.

Chapter 3

Design

In this chapter we describe how we have designed the framework. First, we describe the development tools on which we build the framework and then we describe the agile ideas that inspire our framework. In the final section of the chapter we model the scenario of distributed ontology engineering.

3.1. Tools to support the framework

As mentioned in Chapter 1, we want to design a distributed ontology engineering framework based on agile methodologies. This framework uses OnToology[1] as the tool to support distributed ontology development and Travis CI to support continuous integration. OnToology is designed to work with Github, which is one of the most common environments for software development, hence we exploit it to support agile strategies. OnToology, Travis CI and GitHub are hence going to be the main tools that are hence going to support the framework proposed:

OnToology OnToology¹ is a web-based tool designed to automate part of the ontology development process in collaborative environments. It is integrated with four external systems, as can be seen in Figure 3.2. These systems are OOPS![26], AR2DTool² and Widoco³, which provide evaluation and documentation services, and GitHub, which provides the development platform. After registering a repository to OnToology, developers just push their changes to Github and the tool will produce the documentation (with several proposals for diagram representation), evaluation and publication of the ontology in the user’s repository. OnToology allows developers to customize which of the integrated tools are enabled or disabled through a configuration file. Regarding its architecture, OnToology is composed of two main parts:

¹<http://ontology.linkeddata.es/>

²<https://github.com/idafensp/ar2dtool>

³<https://github.com/dgarijo/Widoco>

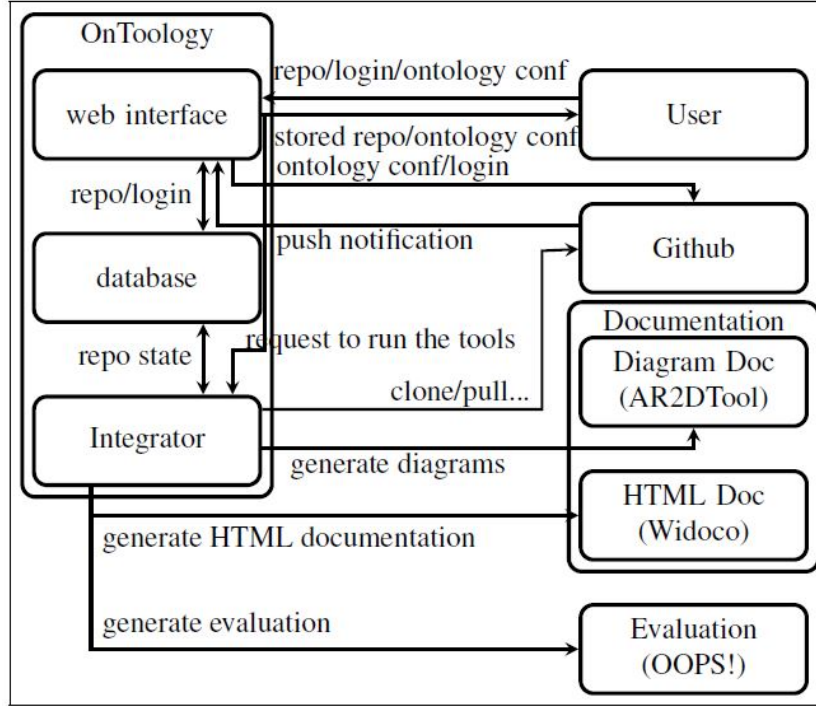


FIGURE 3.1: OnToology Architecture, extracted from [1]

the web interface and the integrator. The purpose of the web interface is to handle Github notifications of the changes of a registered ontology repository and the purpose of the integrator is to talk to the external systems.

GitHub GitHub⁴ is a web-based Git repository hosting service for version control and collaboration. Git⁵ is a free and open source distributed version control system. It lets the users work together on projects from anywhere. Its version control system allows the users to have a complete project history with all the changes made, including the date and the author, and the different versions of the project. It also provides project management components as issues and milestones, that can be used in the users communication and organization. web

Travis CI Travis CI⁶ is a hosted continuous integration platform, free for all open source projects hosted on Github. With just a file called “.travis.yml” containing information about the project, we can trigger automated builds with every change to our code base in the master branch, other branches or even a pull request.

⁴<https://github.com/>

⁵<https://git-scm.com/>

⁶<https://travis-ci.org/>

These three tools are going to be exploited to support agile practices and to integrate agile tools, as it is described in the following sections.

3.2. Agile practices in ontology engineering

We have analyzed software engineering best practices and ideas in order to obtain the methods and tools that can be useful in ontology engineering. As we mentioned in Chapter 1, agile methodologies have the most useful ideas and practices to integrate with ontology development, due to its flexibility and similarity with ontology engineering. After analyzing the methodologies [16], we have selected six basic strategies that can help ontology engineering:

- **Evaluation using acceptance and unit tests.** Continuous evaluation is an important aspect in software development. In software engineering there are several types of errors according to the behavior, the integration or the acceptance of the product. Adapting these ideas to our scenario, if we execute an acceptance and unit test we can verify the correct behavior of the ontology and check if it meets all the requirements, what is going to give us powerful information about the functionality and quality of the ontology created.
- **Version control and conflict resolution.** While working in a collaborative scenario it is necessary that the tool being used provides version control so that the user can track all the changes that are being made. Every change tracked includes who made the change, why they made it, and it can include references to problems fixed, or enhancements introduced by the change. The version control system can also reverse changes when necessary. In our scenario, GitHub provides this version control system so the developers can use it to track and revoke changes made to the ontology.
- **Roles.** When we are working in a collaborative scenario it is necessary to have different roles, which have specific tasks associated according to their responsibilities. In an ontology development scenario, different roles participate according to their knowledge about ontologies and the domain of the problem.
- **Continuous integration.** Continuous integration is a software development process where code is tested against its dependencies (and code that depends on it) regularly. The goal of continuous integration is

to reduce risks and overhead, identifying bugs as soon as possible. In ontology development, integrating continuous integration and unit and acceptance test can allow the developers to know at every moment the errors identified in an ontology.

- **Continuous communication.** Communication is one of the central aspects of agile development, both within the development teams as well as with customers. All the participants in the development process should know the changes made to the ontology, so it can be improved. It is also important to encourage discussions between users about issues related to the ontology to reach collective decisions.
- **Quick reaction to changes.** Agile developers embrace change, accepting the idea that requirements will evolve throughout the project. In both software engineering and ontology engineering it is not usual that all the requirements are fixed at the beginning of the development process, so the developers have to be prepared to face changes over them without waste of time or resources. To achieve this flexibility the agile developers promote continuous development to be able to improve the product continuously.
- **Frequent releases.** Agile projects are commonly broken up into mini-projects, each with a small set of features that take only a few weeks to implement. Every release has a specification, development and testing phase. This means that every couple of weeks the software is fully usable, although it may have very few features at the start. This idea introduces the concept of sprint, which is an agile development term that represent a set period of time during which specific work has to be completed. In the ontology development scenario, these frequent releases can be applied as frequent versions of the ontology including evaluation and documentation.

These agile strategies are going to be included in our framework in order to make more efficient the development process.

3.3. Modeling the agile scenario in ontology engineering

We are going to define the distributed scenario to develop ontologies. First, we identify the roles that have to take part in an ontology development scenario and their responsibilities, and then we propose an ontology development process which will support all the phases in ontology engineering.

3.3.1. Roles

In order to structure the scenario and analyze how the ontology has to be created we have identified several roles that have to participate in ontology development. Each role has specific responsibilities and consequently it will have specific tasks associated:

- **Domain expert:** The domain expert represents the person with the appropriate prior knowledge about the domain. This person knows everything about the domain in which the ontology is created and doesn't need to know how to create ontologies. His tasks are to define the requirements and user stories needed and to complete the documentation of the ontology so that the vocabulary can be understandable by other similar users.
- **Ontology engineer:** The ontology engineer is the person that has to create the ontology in OWL or any other ontology language, following the requirements and user stories identified by domain experts.
- **Curator:** The curator has to validate if the metadata of the vocabulary is correct. The curator doesn't need to know about the domain of the ontology or about the ontology development process since he only has to check if the documentation includes all the metadata that is needed.

3.3.2. Ontology development process

Adapting the software engineering process to ontology engineering, we designed an ontology development process built on specific technologies: OnToolology, GitHub and Travis CI⁷. OnToolology is the main tool that supports the framework, GitHub is the environment for development in OnToolology

⁷<https://travis-ci.com/>

and Travis CI is the tool integrated in OnToology that supports continuous integration. In this development process we support all the ontology development phases (i.e versioning, evaluation, documentation and implementation) and include agile techniques that can help the development process as continuous integration and sprints. In our scenario, each sprint will generate a new version of the ontology that meets a subset of the requirements identified by the domain expert.

The ontology development process proposed, which is represented in Figure 3.2, consists of 3 iterative phases, namely: *Requirements iteration*, *Development iteration* and *Release iteration*. Each of these phases is described below:

- **Requirements iteration.** This initial phase consists of extracting and analyzing ontology requirements and formalizing them into SPARQL. Furthermore, during this phase a new sprint will be initialized selecting the requirements that it has to meet. The phase includes two activities:
 - **Identification of requirements as User Stories or Competency Questions.** In this activity, domain experts have to identify and analyze the ontology requirements and formalize them into user stories or competency questions. These requirements have to be described clearly and concisely, so that other participants in ontology development can understand them. In addition to the competency questions, domain experts have to identify the possible results for each question, so that the system can verify if the results obtained from the ontology are correct. Domain experts can add new requirements during ontology development, due to the fact that it is difficult to know all the requirements at the beginning of the process. Each requirement can be prioritized according to the interest of domain experts.
 - **Formalization of User Stories or Competency Questions into SPARQL queries.** Once we have received the competency questions or the user stories from the domain expert, the ontology engineer has to convert them into SPARQL queries. The ontology engineer also has to formalize the possible results for the SPARQL queries to adequate them to SPARQL results, including the data type, the number of expected results and some samples. These queries and results are going to be used in the acceptance test described later in Section 3.4.2.2 to assure that the ontology meets

all the requirements that the domain expert ask for. These formalized queries has to be stored in GitHub in order to let the system use them. In this activity, the developers can choose to only formalize certain user stories according to their priority. These user stories represent the “backlog” requirements for the new sprint so they are the only ones that have to be taken into account during the following phase.

- **Development iteration.** During this phase the ontology has to be developed in OWL and stored in GitHub. This development iteration includes the concept of *continuous integration*, which will allow us to have a continuous evaluation of the ontology. The development cannot be finished until there are no bugs in the ontology and it meets all the requirements formalized in the previous phase. This iteration includes two activities:
 - **Ontology development.** This activity integrates ontology implementation, users communication using GitHub issues and version control system. During this activity the developers implement the ontology using editors such as NeOn-toolkit⁸ or Protégé⁹. The implementation created using such editors has to be stored in GitHub in order to let the system and other developers use it. The participants in the ontology creation can also access the change history project of the ontology stored, to the different versions and communicate with the other participants of the project using GitHub, as it is a collaborative environment for development which integrates these functionalities. In Section 3.4.1 we propose a communication system exploiting GitHub issues in order to improve the discussions between developers in the ontology development process.
 - **Evaluation with Continuous Integration.** This is an automatic activity executed by the continuous integration tool Travis CI which, following the concept of continuous integration and the agile idea of continuous evaluation, executes an evaluation system whenever there is a push in Github. We propose an evaluation system for this framework that includes a *unit* and *acceptance test*, which will indicate continuously to the user if the ontology implementation is correct and if the ontology meets the requirements

⁸<http://neon-toolkit.org/>

⁹<http://protege.stanford.edu/>

formalized in the requirements iteration. With this continuous evaluation we can reduce risks and overhead and detect bugs early. This evaluation system is explained in detail in Section 3.4.2.

- **Release iteration.** This is the final phase of the iterative process. This iteration follows the agile idea of frequent releases promoting versions of the ontologies by make use of the sprints. It contains four activities:
 - **New version.** Once the development iteration is done a new version of the ontology is created. This new version has to be notified to the team and to the system through GitHub tags (e.g “v0.1”). Each version can not have bugs and needs to meet all the requirements selected in the requirement phase, which are the ones chosen for the sprint. After each version is completed, it can be started a new sprint if the developers return to the initial phase of the development process and implement new requirements of the ontology.
 - **Documentation.** Once we have a new version created and tagged with GitHub labels, the documentation is automatically generated by OnToology using Widoco and AR2DTool, which generate HTML documentation, diagrams and examples that shows how to use the vocabulary. The documentation generated from Widoco, which is the description of the vocabulary and its terms, has to be completed by the domain experts, as they are the ones that knows everything about the domain.
 - **Metadata Validation.** After the system generates the documentation, the curator has to check the metadata. The curator receives a checklist with the metadata that has to be contained in the ontology. If this metadata is not correct or incomplete, the documentation has to be modified until it is. The standard metadata that has to be contained in a vocabulary is:
 - Title and release
 - Current version of the ontology
 - Latest version of the ontology
 - Previous version of the ontology
 - Revision number

- Author
- Contributor
- License
- Abstract
- Imported ontologies
- Extended ontologies

In this activity the curator verifies that the domain expert has completed the documentation generated by Widoco. It is recommended for the curator to use the online tool RawGit¹⁰ to visualize the HTML document generated online, in order to ease the verification process.

- **Generate release and publication.** After we have the documentation completed and with the correct metadata, the ontology can be published if the developers want to. OnToology produces a bundle with the documentation that is ready to be deployed on a server. To notify the fact that the ontology is ready to be published to the team, a “release version” of the ontology has to be created using GitHub tags, so that all the participants can know the status of the ontology. After this “release version”, a new sprint can also be created if the developers return to the requirements phase of the development process.

¹⁰<https://rawgit.com/>

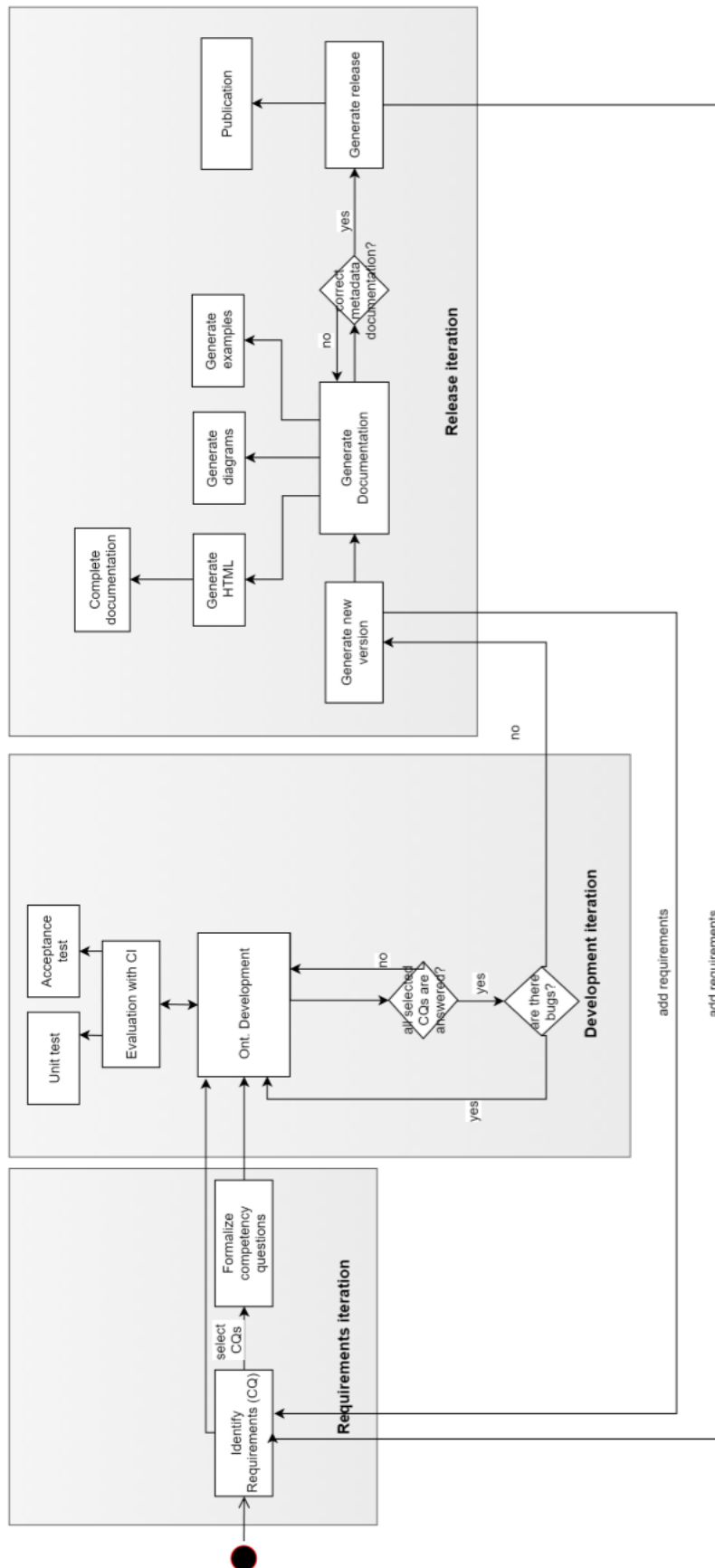


FIGURE 3.2: Ontology development process

Each of the activities defined below has a role associated. Figure 3.3 represents the integration of the development process proposed and the roles defined in Section 3.3.1, showing which role is involved in each activity.

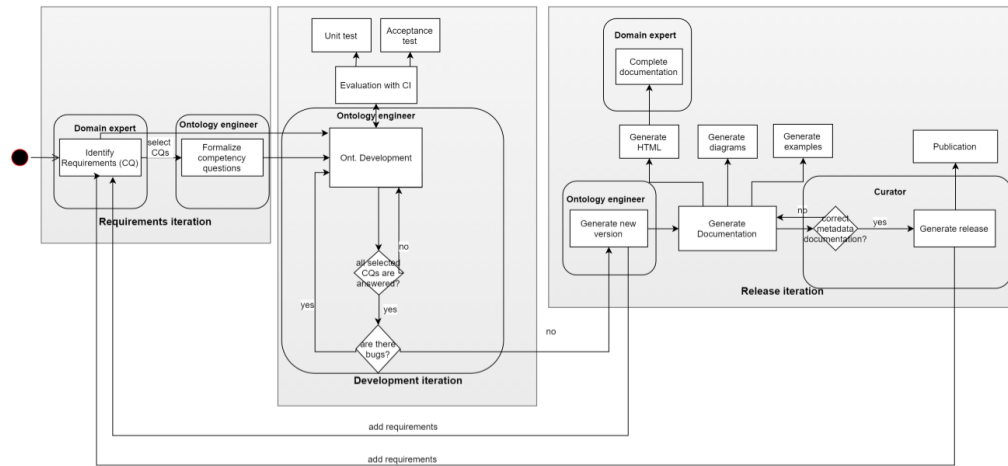


FIGURE 3.3: Development process with roles

On the other hand, each activity in the development process has also outputs associated that can be used by the developers, the costumers or the system. These outputs are represented in Figure 3.4.

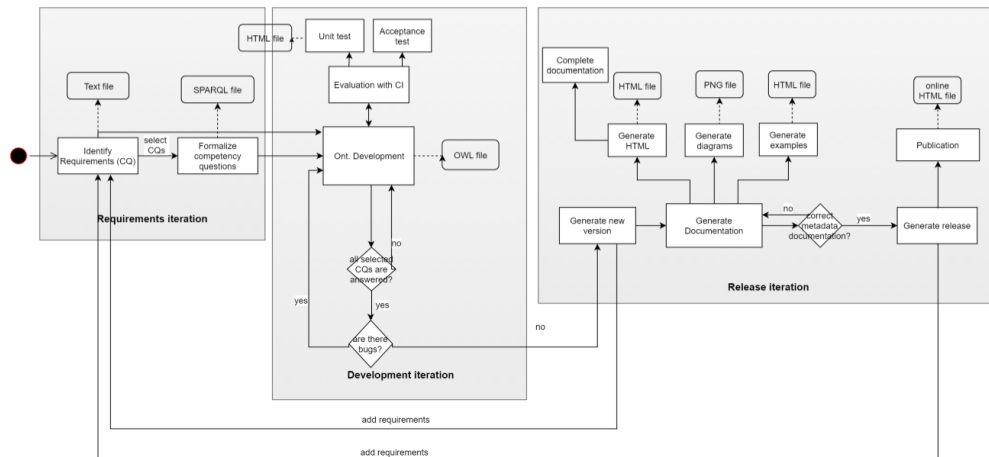


FIGURE 3.4: Development process with outputs

Following this development process all the ontology development phases are supported integrating tools for evaluation, documentation and diagrams. Due to it uses OnToology for carrying out these tasks the developers can customize which of the integrated tools are enabled or disabled through a configuration file given by OnToology.

3.4. Agile techniques adapted to ontology engineering

After analyzing the agile methodology and its techniques, we conclude that the communication between users and the continuous evaluation of the ontology created in an agile scenario are crucial tasks. Because of that, we propose two systems for our framework that apply agile methods and tools to improve these two aspects.

3.4.1. Communication system exploiting Github issues

Due to the importance of communication between users in an agile development project we propose a communication system exploiting Github issues, which is the environment for development that supports the communication between users in our framework. The main idea is to propose a procedure to use GitHub labeled issues to notify and suggest modifications over the ontology. This labeled issues will have a tag that resumes the topic of the issue, which can be used to structure the discussions and organize the tasks according to the roles. This procedure proposed is based on the agile technique of *continuous communication* between users and is represented in Figure 3.5.

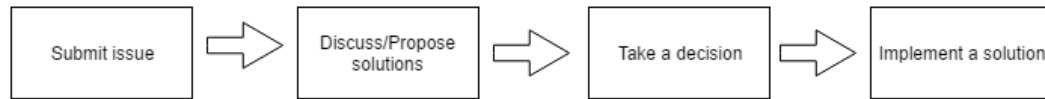


FIGURE 3.5: Modification procedure

Following Figure 3.5, first an issue is detected and communicated to the other participants using GitHub. This issue has to be labeled indicating the topic of the issue, which can be about implementation, documentation, questions or other tasks related to ontology development. Second, the possible solutions to the issue have to be discussed among users, who need to take a collective decision of how to solve it. Finally, the issue can be solved by the user assigned to it. Even though this is the general procedure, in some scenarios where the modification over the ontology is minor and doesn't need different opinions this procedure is not completely necessary. For example, if there is a syntax bug in the ontology, the developer can simply fix the error and submit a labeled issue notifying the modification.

In this work we propose a labeling system to tag each issue systematically according to the task associated to it. To achieve this objective, we first

identified the tasks that can be carried out by the team during ontology development. Each task is going to have a topic associated that is going to help us to label the different issues that can be created during the development process. In Table 3.1 there are summarize all these tasks identified that can be carried out including those related to ontology implementation, validation, documentation and versioning. Each of these tasks has also associated a role which represents who is responsible for the task. Following agile continuous communication it is useful for the team to notify, using issues, each action that is being made over the ontology. These notifications will let everyone know the modifications done.

Action		Role
Implementation	Propose term	Ontology Engineer
	Identify bug	Ontology Engineer
	Insert/delete/modify term	Ontology Engineer
	Bug resolution	Ontology Engineer
Documentation	Document modification	Domain expert
Validation	Metadata validation	Curator
Versioning	Questions	Domain expert, Curator, Ontology engineer
	Release version	Curator
	New version	Ontology Engineer
	New requirement	Ontology Engineer

TABLE 3.1: Actions and roles in ontology development

Once we have the tasks identified, we propose labels associated to each task so that the team can use them to tag issues. Table 3.2 shows the labels proposed associated to each action identified in Table 3.1:

	Action	Label
Implementation	Propose term	enhancement
	Identify bug	bug
	Insert/delete/modify term	edit
	Bug resolution	bug resolution
Documentation	Document modification	document
Validation	Metadata validation	metadata
Versioning	Questions	question
	Release version	release
	New version	new version
	New requirement	requirement

TABLE 3.2: Actions and labels in ontology development

With this labeling system we want to structure the discussions between the team associating issues with particular topics of development, allowing the team to track the changes made by contributors. Moreover, the issues will also indicate with their tags which role has to participate in the issue: if there is an issue submitted related to “metadata validation”, the curators know that they have to take part in the issue, and the domain experts don’t. This issue will promote a discussion between developers which have to propose solutions for it and take a decision. The issue will be solved by the curator that has the issue assigned.

To improve the basic issue system of GitHub we propose its integration with *Zenhub*¹¹, which is a tool that provides Kanban boards to organize issues. Kanban boards are a popular tool in agile development to organize tasks, issues and their status. These boards can help to optimize the productivity of our work. The goal of a Kanban system is to limit the amount of work in progress so that the work flowing through the system matches its capacity.

Using the Kanban strategy provided by *Zenhub* in our approach, we are going to have four types of issues distributed in four boards. These boards are going to represent the status of each issue:

- **Backlog issues.** List of to-do items at the beginning of the project. This pipeline is a prioritized backlog of items ready for development. The Backlog is used during sprint planning meetings: the higher an issue is on this list, the higher the priority.

¹¹<https://www.zenhub.com/>

- **Icebox issues.** List of issues “on hold”. The Icebox represents items that are a low priority in the product backlog. These issues are kept in the icebox with just enough information attached and they can be picked it up some time in the future.
- **In progress issues.** List of tasks that are being executed by a developer. Each issue in this pipeline should have an assigned owner who is responsible for its completion. If a team member decides to take on a task, she or he simply self-assigns the issue and moves it to the In Progress column, instantly communicating to the rest of the team that the task is underway.
- **Done issues.** Issues in this pipeline need no further work and are ready to be closed. Having a good definition of “Done” agreed upon before work starts on an issue is very helpful. If there were any objectives associated with the issue, they can be appended prior to closing.

An example of Kanban boards and issues is represented in Figure 3.6.

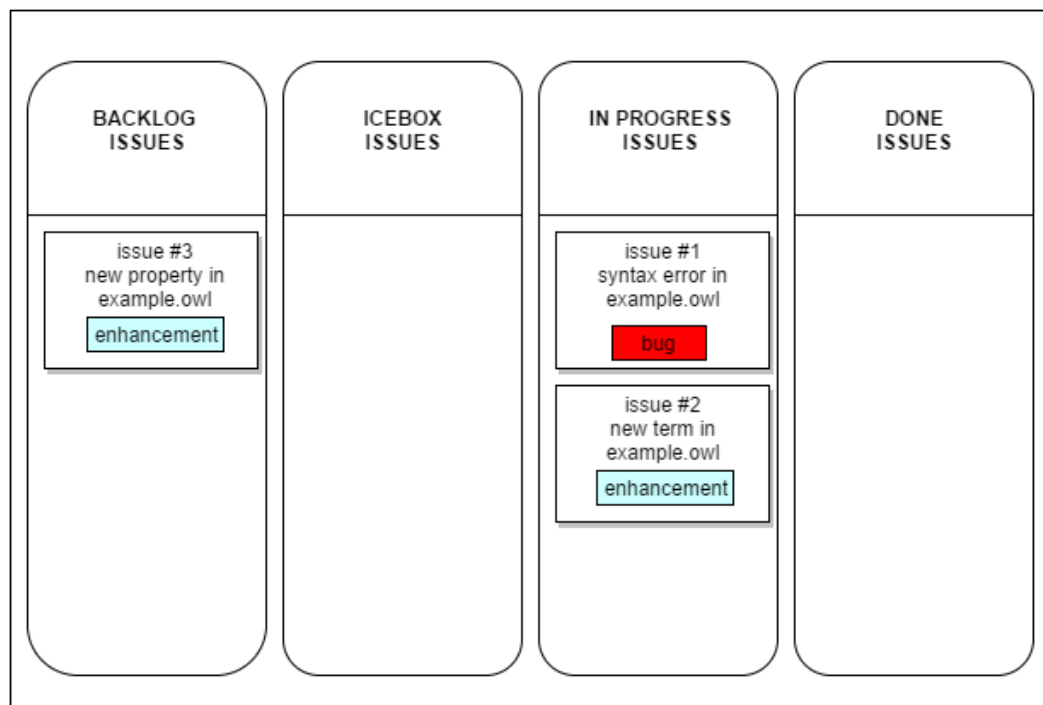


FIGURE 3.6: Kanban example

The procedure to use this technique points out that when a developer creates an issue it has to be placed in the “Backlog” board. When this issue is assigned to someone, it has to be modified and placed in the “In progress” board. Finally, when the issue is solved it has to be placed in the “Done”

board. The issues that are not going to be started for the moment have to be placed in the “Icebox issue”.

With this Kanban procedure, the boards allow the users to work easily with the issues that are identified, providing a quick visualization of the status of each issue. Besides this organization, the tags associated to each issue will also help the developer to know the topic of the issues identified. In summary, with this Kanban labeled system we want to facilitate the communication between users with an intuitive management of GitHub issues.

3.4.2. Evaluation system

Testing becomes an essential component of each and every phase of the development process. Agile software engineering proposes four main types of tests to verify the correct behavior of a system: *Acceptance test*, which determine whether or not the software system has met the requirement specifications, *Integration test*, where program units are combined and tested as groups in multiple ways, *Unit test*, easy manageable tests for small parts of a program, and *Regression test*, which tests changes to computer programs to make sure that the older programming still works with the new changes.

Ontologies, as every software product, need to be tested in order to guarantee quality and functionality. Unfortunately, ontologies are not exactly a classic software product so the ideas and tests of software engineering evaluation cannot be applied directly without an adaptation to ontology engineering. There are some approaches that try to make the adaptation from software engineering to ontology engineering like the ones presented in [32] or [18]. The work presented in [32] tries to adapt the unit test concept to ontology engineering while the work presented in [18] follows the software engineering idea of test-driven development[4] and specifies 36 generic tests, as TBox queries and TBox axioms tested through individuals.

In our framework we want to design an evaluation system that adapts the tests proposed by agile software engineering, as they can evaluate several aspects of their products. We want to check the functionality and quality of ontologies, hence in our evaluation system we include two aspects: *Verification of the ontology* and *Validation of the ontology*. Verification relates a system implementation and its explicit specification whereas validation relates a system and its end users implicit requirements. If the requirements have been specified, then the system is verified against this specification, and

the specification is validated against its end users needs [21]. Because of these two aspects, in this document we are going focus on the mentioned *Acceptance test*, which represents the validation aspect, and *Unit test*, which represents the verification aspect. *Acceptance test* and *Unit test* are the software evaluations that we consider important and useful for ontology development, due to they are going to check if the ontology meets the requirements and if it is well implemented.

These tests, which are explained in Sections 3.4.2.1 and 3.4.2.2, are adapted to our scenario of ontology engineering and are going to be executed using the continuous integration tool Travis CI, as it is explained in the development process in Section 3.3.2. Because of this continuous integration, they can also be considered as *Regression tests*, since they are going to be executed whenever there is a change in the ontology. With this continuous tests we can check that even there is a change, the implementation remains correct and the requirements are achieved. The errors encountered by the tests are going to be communicated to the users through GitHub labeled issues as it is explain in the following sections. Following the communication system proposed in Section 3.4, the developers can organize these issues created by the system and promote discussions about them in order to solve the problems detected.

3.4.2.1. Unit test

Unit testing is a level of software testing where individual units, which are the smallest testable parts of a software, are tested. A unit may be an individual program, function, procedure or method. The purpose of this test is to validate that each unit of the software performs as designed. Unit testing increases confidence in changing/ maintaining code. If good unit tests are written and if they are run every time any code is changed, we will be able to promptly catch any defects introduced due to the change. Also, if codes are already made less interdependent to make unit testing possible, the unintended impact of changes to any code is less.

The idea of adapting unit tests from software engineering to ontology engineering was proposed by [32]. In their work they conclude that in ontology engineering we can not apply the unit test used in software engineering because ontologies behave differently than program units, as there is no information hiding in ontology engineering, and thus no black box components. Because of this, they propose the unit test in ontology engineering as indicators of

potential errors or omissions. Some of the ideas presented by them, inspired by the notion of unit testing, are:

- **Affirming derived knowledge.** This consists in creating two ontologies, the positive test ontology and the negative one. The developers should check that for each axiom in the positive test ontology, such axiom is inferred by test ontology. The developers should check that for each axiom in the negative test ontology, such axiom is not being inferred by the tested ontology.
- **Expressive consistency checks.** This test consists in introducing a test ontology T for a ontology that includes the high axiomatization of the terms included in the ontology O and check the satisfiability of the merged ontology $T \cup O$.

Regarding ontology evaluation, there are approaches that wants to assure quality by identifying the existence of anomalies in the ontology. Some of these approaches are OOPS! [26], Moki [10] and XD Analyzer [5]. OOPS! makes an evaluation of an ontology and provides a list of 33 typical pitfalls for developers. These pitfalls have associated, among others, the importance level of each pitfalls, which indicates how crucial is the appearance of the pitfall regarding to the ontology quality and its functionality, and the ontology evaluation aspects in which the pitfall is classified. Other evaluation approach is Moki, which adopts ontology evaluation to automatically check an ontology for compliance with modeling guidelines to detect potential modeling errors. The evaluation of the ontology happens iteratively, while the ontology being developed. MoKi has a checklist of modeling guidelines where it checks if the ontology elements comply with them. Finally, XD Analyzer is one component of eXtreme Design Tools, which is a plugin for NeOn toolkit composed of components that support pattern-based design. This design supports reusability and use of best practices in ontology engineering. XD Analyzer provides feedback to the user with respect to the best practices of ontology design. The feedback can contain errors about missing type, warnings about the bad practices and suggestions for improvements.

In our framework, we chose to use OOPS! because it is used in a lot of projects and got a lot of positive feedback. It is also an online tool and provide a REST web service, so no installation is needed. Furthermore, it detects the greater number of pitfalls comparing with Moki or XD Analyzer. In Figure 3.7, extracted from [26], it is shown the OOPS! pitfalls classification according to

the pitfalls aspects and the importance of them . The complete catalogue can be found in [26].

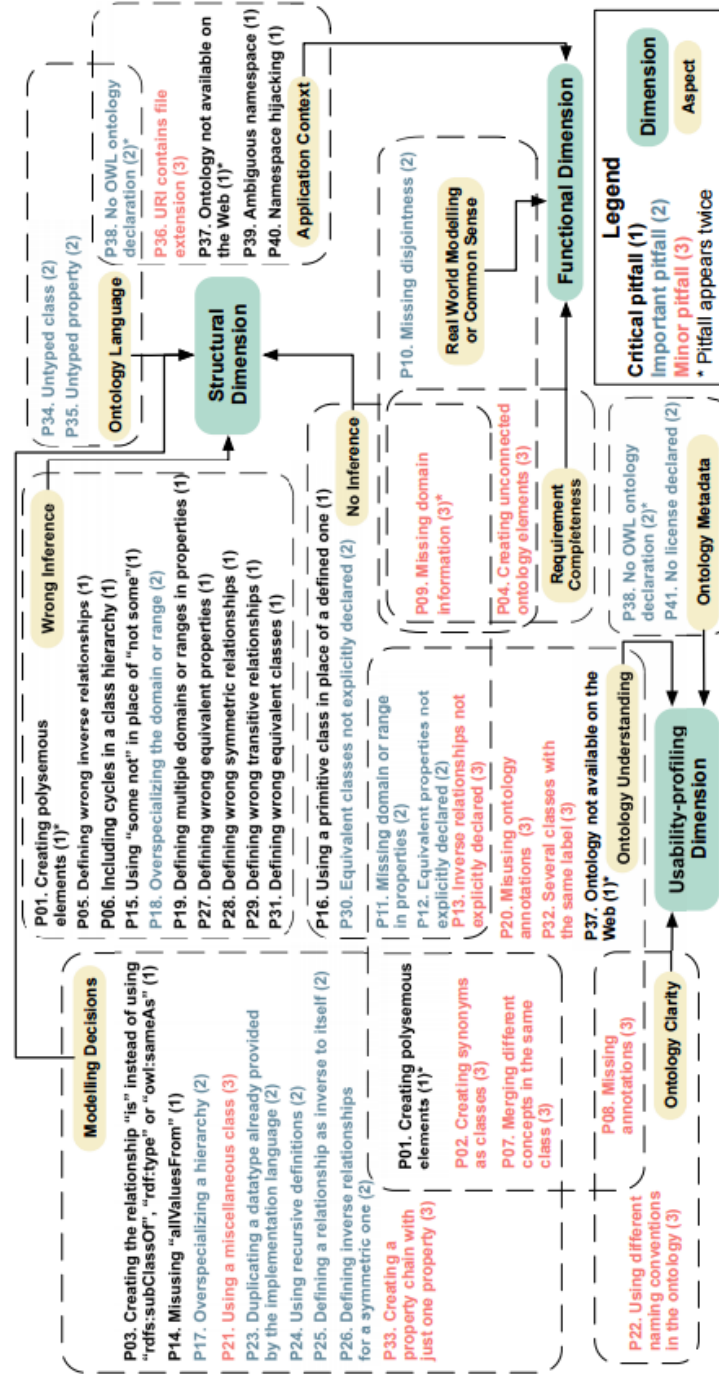


FIGURE 3.7: OOPS! pitfalls classification, extracted from [26]

Using the unit test ideas presented below, which are focus on ontology functionality, and OOPS! as the evaluation tool, we are going to design a unit test for ontologies based on OOPS! pitfalls. To design this unit test we are going to focus, as it motivates the work of [32], on ontology functionality and

behavior. Because of this, we are only interested in those OOPS! pitfalls that can affect the consistency of the ontology, the inference or the behavior. We are going to select the pitfalls that we are interested in and distribute them in groups according to their topic. OOPS! will provide us information including a brief description of each, number of occurrences and affected elements.

To select the pitfalls we focus on the importance level and on the classification of the pitfall. Regarding the importance level, there are three groups of pitfalls:

- *Critical* pitfalls. This type of pitfalls can affect the ontology consistency, reasoning and applicability and because of that it is crucial to correct them.
- *Important* pitfalls. This type of pitfalls, even they are not critical for ontology function, can carried out errors in consistency or inference.
- *Minor* pitfalls. These pitfalls don't present a problem, due to they don't affect to ontology behavior, but they are useful to ontology understanding.

Taking into account the importance of the pitfall in the ontology, we decide that *Crucial* and *Important* pitfalls should be included in our test, because they include the pitfalls that affect the functionality of the ontology. *Minor* pitfalls, however, are related to ontology improvement so we are not interested in them to be included in the unit test. On the other hand, regarding to the classification of the pitfalls, we distribute them in six groups of issues according to the aspects identified in OOPS! related to them:

- *Inference*, which includes pitfalls that has some type of error in inferred knowledge. This type of pitfalls includes the aspect of “No inference”, which refers to checking whether desirable or expected knowledge could actually be inferred from the given ontology but it is not, and “Wrong inference”, which refers to the evaluation of the inference of erroneous or invalid knowledge.
- *Modeling*, which includes pitfalls that has some type of error related to modeling decisions. This group includes pitfalls related to the evaluation whether developers use the primitives provided by ontology implementation in a correct way, and also deals with the knowledge that the domain expert expect to appear, but it is not represented.

- *Metadata*, which includes pitfalls that has errors related to the metadata of the ontology or the application context, which refers to the adequacy for the ontology for a given application.
- *Ontology language*, which includes pitfalls that has errors related to ontology language specification or syntax.
- *Ontology understanding and clarity*, which includes pitfalls related to the information that can help the user to understand the ontology content, including elements and properties.
- *Requirements completeness*, which includes pitfalls related to the coverage of the requirements specified in the Ontology Requirements Specification Document.

Due to we are only interested in those pitfalls that can affect the behavior of the ontology, we select only those pitfalls related to *Inference*, *Modeling*, *Metadata* and *Ontology Language*. The pitfalls related to *Ontology understanding and clarity* and the pitfalls related to *Requirement completeness* are not considered in this test since they don't affect any important aspect of the ontology implementation. The complete list of pitfalls selected are summarized in Table 3.3, which are the pitfalls that are going to be notified to the developers in our test.

In order to communicate these selected pitfalls encountered by OOPS! to the user we are going to use GitHub issues for each owl file evaluated, as it is an easy way to get information. Each issue is going to contain the pitfalls of one ontology file and of only one of the groups presented in Table 3.3 (i.e. Inference, Modeling, Metadata or Ontology language). As in the communication system presented in Section 3.4.1, the issues created by the tests are going to be tagged to give to the developers a quick visualization of the errors detected in the ontology. Each issue is going to be tagged according to:

- It is a unit test issue, so it can be distinguished from errors from other tests.
- The group of the pitfalls contained in the issue, so the developers can know the type of pitfalls encountered in the ontology.
- The importance of the pitfalls contained in the issue, so the developers can prioritize the issues that has to be solved.

With these groups of issues and tags it will be easy for the developers to identify which type of errors has the ontology and the importance of them without having to open the issue and read the content. These tags also indicate the priority of the issues, due to it is more crucial to solve a critical issue than an important or minor issue. The title of each issue will inform the user about the file in which the errors are encountered.

Pitfalls		
Group	Pitfall	Importance
Inference	P01. Creating polysemous elements	Critical
	P05. Defining wrong inverse relationships	Critical
	P06. Including cycles in class hierarchy	Critical
	P11. Missing domain or range in properties	Important
	P12. Equivalent properties not explicitly declared	Important
	P15. Using “some not” in place of “no some”	Critical
	P16. Using a primitive class in place of a defined one	Critical
	P18. Overspecializing the domain or range	Important
	P19. Defining multiple domains or ranges in properties	Critical
	P27. Defining wrong, equivalent properties	Critical
	P28. Defining wrong symmetric relationships	Critical
	P29. Defining wrong transitive relationships	Critical
	P30. Equivalent classes not explicitly declared	Important
	P31. Defining wrong equivalent classes	Critical
	P03. Creating relationships “is” instead of using “rdfs:subClassOf”	Critical

Modeling	P10. Missing disjointness	Important
	P14. Misusing “owl:allValuesFrom”	Critical
	P17. Overspecializing a hierarchy	Important
	P23. Duplicating a datatype already provided by the implementation language	Important
	P24. Using recursive definitions	Important
	P25. Defining a relationship as inverse to itself	Important
	P26. Defining inverse relationships for a symmetric one	Important
Metadata	P37. Ontology not available on the Web	Critical
	P38. No OWL ontology declaration	Important
	P39. Ambiguous namespace	Critical
	P40. Namespace hijacking	Critical
	P41. No license declared	Important
Ontology Language	P34. Untyped class	Important
	P35. Untyped property	Important

TABLE 3.3: Selected pitfalls

As it is mentioned before, OOPS! also gives to the system some minor pitfalls and suggestions that, even they don't affect the behavior of the ontology, it is useful to solve them in order to improve ontology understanding. These pitfalls are related to the groups mentioned before *Ontology understanding and clarity* and *Requirement completeness*. To take advantage of this type of pitfalls identified by OOPS! they are going to be included in an special issue that is going to be communicated to the developers. This special issues is going to be tagged according to:

- It is an enhancement issue
- If it includes minor pitfalls it is going to has the tag “minor”.
- If it includes suggestions it us going to has the tag “suggestions”.

These minor pitfalls are shown in Table 3.4.

Pitfalls	
Pitfall	Importance
P04. Created unconnected, ontology elements	Minor
P07. Merging different concepts in the same class	Minor
P09. Missing domain information	Minor
P13. Inverse relationships not explicitly declared	Minor
P21. Using a miscellaneous class	Minor
P33. Creating a property chain with just one property	Minor
P36. URI contains file extension	Minor

TABLE 3.4: Enhancement pitfalls

This special issue is not included in the unit test and the pitfalls contained in it don't represent a problem for ontology functionality, but the users should take them into account to improve the ontology.

3.4.2.2. Acceptance test

In software engineering an acceptance tests (also called customer tests or customer acceptance tests) describe black-box requirements, identified by the project stakeholders, which the system must conform to. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

If we adapt this idea into ontology engineering, the acceptance test describe what kind of knowledge the resulting ontology is supposed to answer. This resulting knowledge is described by the competency questions provided by the domain experts. These competency questions, which can be formalized in a query language as SPARQL, are proposed as the best technique for establishing the ontology requirements. Taking as inspiration software engineering, in which requirements are divided into functional and non functional requirements [27], in ontology engineering the requirements can also be divide in two types. They are defined in [29]:

- Non-functional ontology requirements refer to the characteristics, qualities, or general aspects not related to the ontology content that the ontology should satisfy.

- Functional ontology requirements, which can be also seen as content specific requirements, refer to the particular knowledge to be represented by the ontology.

Formalizing these competency questions, and formalizing the expected answers as well, allows the system to automatically check if the ontology meets the requirements stated with the competency questions. These competency questions can not guarantee that the ontology is complete, but gives an idea to the developers that the ontology can answer to all the questions that the domain expert considers necessary.

In the acceptance test proposed in this document the competency questions, which are going to represent the ontology requirements both functional and non-functional, are going to be described by domain experts in natural language. They also have to provide some expected results of the query, so they system can verify that the ontology answer correctly to the competency question. The competency questions can be prioritized according to the importance of the requirement associated. Once the domain expert has the competency question described, the ontology engineers have to formalize them and their results into SPARQL language. They have to convert the question written in natural language into a SPARQL query that can be executed over the ontology. Regarding the results, the ontology engineers also have to formalized them into technical results. They have to include the following properties:

- The number of results expected. It can be expressed with the symbols $>$ or $<$ (e.g >5).
- The data type of the results expected, which can be URIRef, Literal or Boolean.
- A list of samples that has to be included in the results given by the system.
- The priority of the requirement (optional property).

Figure 3.8 shows an example of query and its results that has to be generated by the ontology engineer from the user stories given by the domain experts.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?label
WHERE {
    ?subject rdfs:label ?label
}

#Results

#Number of results
>15

#Type of the results
Literal

#List of samples
xsd:anyLiteral

#Priority
1
```

FIGURE 3.8: Query-Result example

We need to consider that not every competency question can be formalized into a SPARQL query. In that case, the competency questions has to be checked by the ontology engineer.

Once the query is executed over the ontology, the results are going to be checked to know if they are consistent with the information provided by the domain expert and the ontology engineer. In the case that the results of a requirement are not consistent with the information given, the system will create an issue in GitHub indicating the ID of the requirement and the error encountered, which includes data type error, sample error or number of results error. In the case that the system did not receive any results from the query, the system will create an issues that notifies that the ontology can not answer to that query. The issue generated from this test is going to have the tag “Acceptance test bug” so it can be easily identified by the developers.

Chapter 4

Implementation and experimentation

We have created a first version of the evaluation system in order to validate the feasibility of the design proposed in Section 3.4.2. In this chapter we are going to explain how this evaluation system was implemented and how the experimentation was made with ontologies that are already created.

4.1. Implementation

The evaluation system designed in Chapter 4 has being implemented using Python. It uses three external services, according to the continuous integration process, the evaluation tool and the query service over the ontology created:

- The continuous integration tool chosen is Travis CI, due to the fact that it has an easy synchronization with GitHub which, as it is mentioned in previous sections, is the environment for development and storage of the ontology files.
- The unit test is implemented using the OOPS! RESTful web service¹ as the evaluation tool, which identifies the pitfalls of the ontology.
- The acceptance test is implemented using RDFLib², which is a pure package work working with RDF that includes a SPARQL 1.1 implementation which support queries and update statements. In this implementation the SPARQL service receives files with specific name structure: *ontologyname_ID.rq*. Each file stores a SPARQL query and the results associated to it.

¹<http://oops-ws.oeg-upm.net/>

²<https://rdflib.readthedocs.io/en/stable/>

The results are communicated to GitHub from Travis CI using the library PyGitHub³, which is going to create the issues that are going to inform the developers about the errors encountered. The complete structure of the implementation is represented in Figure 4.1.

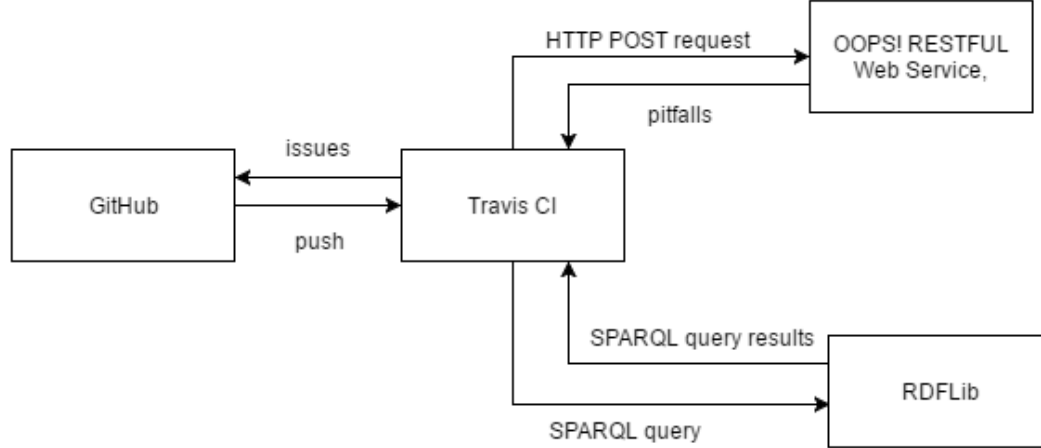


FIGURE 4.1: Implementation structure

In Figure 4.1, there are shown the three external services and how they interact with each others. The developers only have to interact with GitHub, which is the tool that will notified to them the issues.

4.1.1. Limitations

In the implementation of the evaluation system are several limitations that are summarized below:

1. Ontology Evaluation does not work with very large ontologies (due to OOPS! web-service timeout)
2. The developer has to initiate session in Travis CI manually using the GitHub account.

4.2. Experimentation

In this section we explain how we have validated the feasibility of the proposed evaluation system design in Chapter 4 after its implementation. Thus, we analyze two real ontologies in sections 4.2.1 and 4.2.2. These ontologies are extracted from the GitHub account of the Spanish thematic network on

³<https://github.com/PyGithub/PyGithub>

Open Data for Smart Cities⁴. The goal of this section is to check whether applying the tests proposed we can identify the errors about implementation and requirements.

4.2.1. Example 1: “Turismo” ontology

In this section we are going to validate the unit and acceptance test designed in Chapter 4 over “Turismo” ontology. This ontology⁵ belongs to the AENOR open data group vocabularies⁶ and it represents tourism and culture topics. Before creating this ontology, the domain experts who participated in the ontology creation process proposed eighteen user stories that the ontology has to answer. These user stories, as aforementioned, represent the ontology requirements.

Unit Test The unit test is automatically executed when there is a change in GitHub. After it is executed over the “Turismo” ontology two issues are reported to GitHub, which represent the pitfalls encountered in the ontology. The list of unit test issues created for this ontology is shown in Figure 4.2. As it is expected, these issues’ tags show the type of the pitfalls and the importance of them, giving the developer an idea of the errors encountered in the ontology without opening the issue.

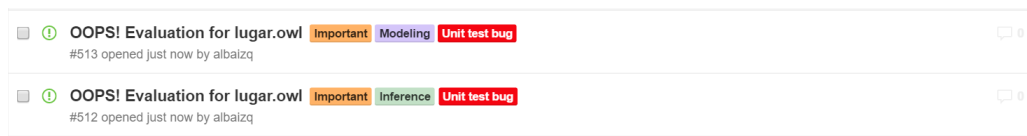


FIGURE 4.2: List of issues

The content of each issue is shown in Figures 4.3 and 4.4. This content includes a description of the error encountered.

⁴<https://github.com/opencitydata>

⁵<https://github.com/opencitydata/vocabularios-datos-abiertos/tree/master/turismo>

⁶<http://vocab.linkeddata.es/datosabiertos/>

OOPS! Evaluation for lugar.owl #513

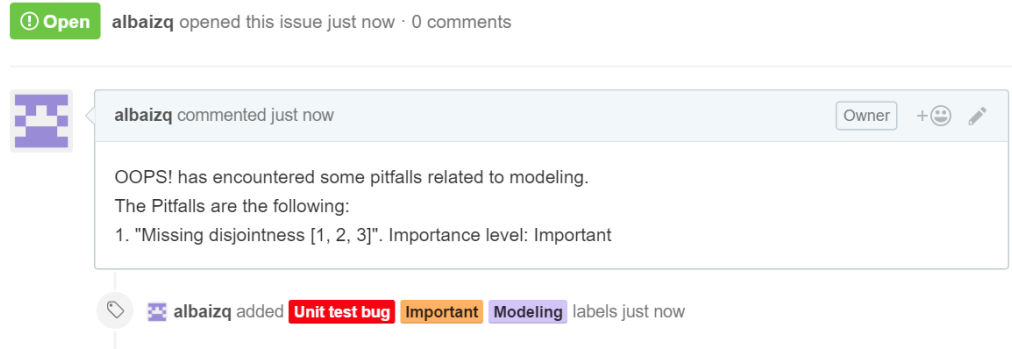


FIGURE 4.3: Modeling issue

OOPS! Evaluation for lugar.owl #512

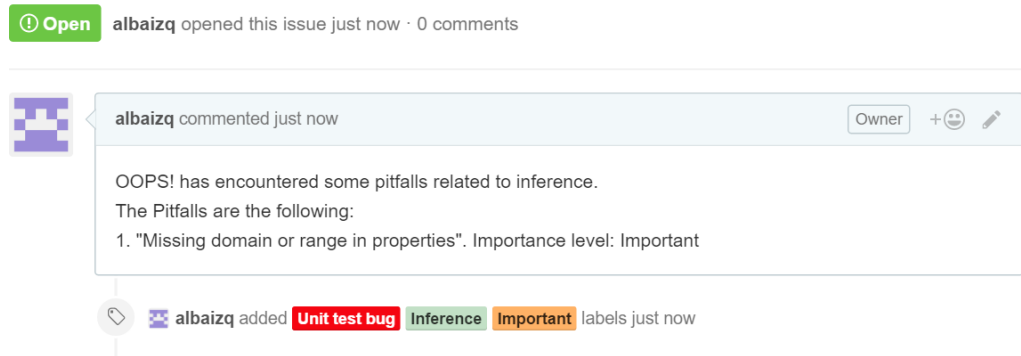


FIGURE 4.4: Inference issue

With these issues in GitHub the developers can know at any moment the most important errors about the ontology implementation.

Acceptance Test This test is also executed by Travis CI when there is a change in GitHub. This test has to identify the requirements that the ontology does not meet. To validate this, we extract five samples of the user stories given to the developers to create the ontology. These user stories are formalized into SPARQL queries, using both SELECT and ASK queries in order to test different formats, and then executed over the ontology. Since the ontology extracted does not have data, we can only test the ontology structure. The user stories and its results obtained from the test are summarized in the following tables:

User story	Quiero ofrecer un servicio de venta de helados itinerante en zonas turísticas y quiero conocer la afluencia de público a las mismas
Query	PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#> SELECT ?label WHERE {<http://vocab.linkeddata.es/datosabiertos/-def/turismo/lugar# ZonaTuristica> rdfs:subClassOf ?superClass ?label rdfs:domain ?super }
Expected results	cardinality: >1 type: URIRef sample: http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar#afluenciaPublico
Results	cardinality: 1 type: URIRef sample list: http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar#afluenciaPublico

TABLE 4.1: User story 03. Influx of people in an area

User story	Quiero crear una app con información de una ciudad e incluir todos los LIT(Lugares de Interés Turístico) que tenga clasificados por categoría, ofreciendo una información básica de cada uno de ellos y un enlace a más información. Así mismo el usuario podrá ver esta información en un mapa
Query	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> SELECT ?,subclass,?label WHERE {?subclass rdfs:subClassOf http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar #LugarInteresTuristico ?subClass rdfs:label ?label,}
Expected results	cardinality: >1 type: URIRef, Literal sample: http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar-#EdificioHistorico, siglo http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar-#Monumento, afluencia de público
Results	cardinality: >1 type: URIRef, Literal sample list: http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar-#EdificioHistorico, siglo http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar-#Monumento, afluencia de público

TABLE 4.2: User story 01. Information about a touristic area

User story	Quiero crear una web con un planificador de viajes en los que los usuarios puedan ir seleccionando LIT según el interés que tengan y agregándolos a su planificador y luego ofrecer una ruta por todos ellos
Query	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> SELECT ?label ?class WHERE { <http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar#Itinerario> rdfs:label ?label <http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar#Itinerario> rdfs:subClassOf ?class}
Expected results	cardinality: >0 type: Literal, URIRef sample: Itinerario, http://schema.org/TouristAttraction
Results	cardinality: 1 type: Literal, URIRef sample list: Itinerario, http://schema.org/TouristAttraction

TABLE 4.3: User story 02. Tours in tourist areas

User story	Quiero saber si un lugar (de interés turístico, comercio, etc.) es accesible, y qué tipo de accesibilidad ofrece.
Query	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> SELECT DISTINCT ?z WHERE { {?subclass rdfs:subClassOf ?class ?x rdfs:domain ?class ?x rdfs:label ?label FILTER regex(str(?label), "accesible", "i") }UNION{ ?subclass rdfs:subClassOf ?class ?x rdfs:domain ?class ?x rdfs:label ?label FILTER regex(str(?label), "tipo de accesibilidad", "i") }} }}
Expected results	cardinality: >1 type: URIRef sample: http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar#ZonaTuristica
Results	cardinality: 3 type:URIRef sample list: http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar#ZonaTuristica http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar#LugarInteresTuristico http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar#Itinerario

TABLE 4.4: User story 18. Accessibility of a place

User story	Quiero crear una app que suministre información de las actividades culturales clasificadas de mi ciudad, que permita visualizarlas en modo calendario, las ubique en un mapa e incluya información sobre el lugar de celebración y la forma de llegar en transporte público.
Query	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> SELECT ?x WHERE { ?x rdfs:subClassOf <http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar#TouristAttraction> FILTER regex(str(?x), "cultural", "i") }
Expected results	cardinality: >0 type: URIRef samples: http://vocab.linkeddata.es/datosabiertos/def/turismo/lugar#EventoCultural
Results	cardinality: 0 type: list of samples:

TABLE 4.5: User story 04. Information about cultural events

The queries are given to the system in files with extension *.rq* as it is shown in the template in Chapter 4. The results obtained by these queries inform the developers that the ontology does not meet all the requirements given by the domain expert. In this example, the query associated to User story 04 does not return the correct results to the users. Because of this, the system generates a GitHub issue to inform the developers about the error. Figure 4.5 shows the complete list of issues created for this ontology. The issue generated from this test is the one tagged as “Acceptance test notification”. The content of this issue is shown in 4.6.

<input type="checkbox"/>		OOPS! Evaluation for lugar.owl	Important	Modeling	Unit test bug		
		#597 opened 8 minutes ago by albaizq	New Issues				
<input type="checkbox"/>		OOPS! Evaluation for lugar.owl	Important	Inference	Unit test bug		
		#596 opened 8 minutes ago by albaizq	New Issues				
<input type="checkbox"/>		Acceptance test notification	Acceptance test bug				
		#595 opened 8 minutes ago by albaizq	New Issues				

FIGURE 4.5: List of issues

Acceptance test notification #595

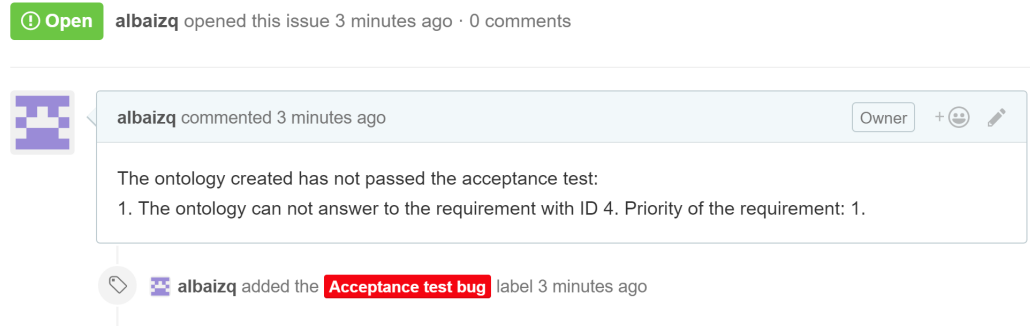


FIGURE 4.6: Content of the issue

The description of the issue, which notifies that the ontology can not answer this user story, means that the ontology did not receive any results from the query associated with the requirement with ID 04.

4.2.2. Example 2: “Callejero” ontology

The “Callejero” ontology⁷ also belongs to the AENOR open data group vocabularies⁸ and it represents street topics. As in the ontology of the Example 4.2.1, before creating this ontology the domain experts proposed seventeen user stories that the ontology has to answer.

Unit Test This test has to identify the pitfalls and its importance in the street ontology. After the test is executed over it, two issues are reported to GitHub, which represent the pitfalls encountered in the ontology. The list of unit issues created are shown in 4.7. As in Example 1, the tags associated to each issue gives information about the issue without opening it.

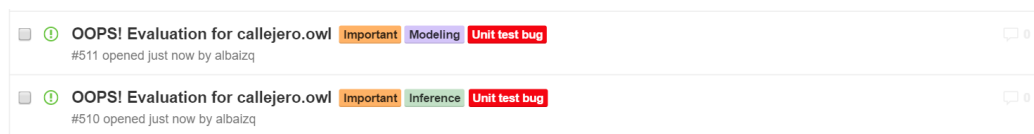


FIGURE 4.7: List of issues

The content of each issue is shown in Figures 4.8 and 4.9.

⁷<https://github.com/opencitydata/vocabularios-datos-abiertos/tree/master/urbanismo-infraestructuras>

⁸<http://vocab.linkeddata.es/datosabiertos/>

OOPS! Evaluation for callejero.owl #517

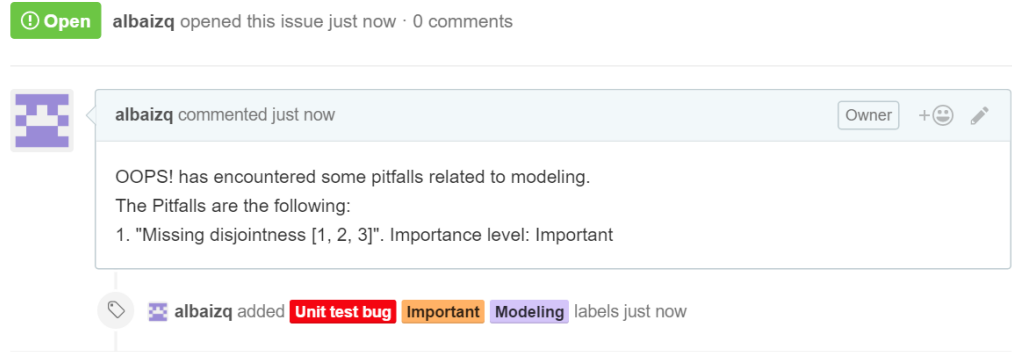


FIGURE 4.8: Modeling issue

OOPS! Evaluation for callejero.owl #516

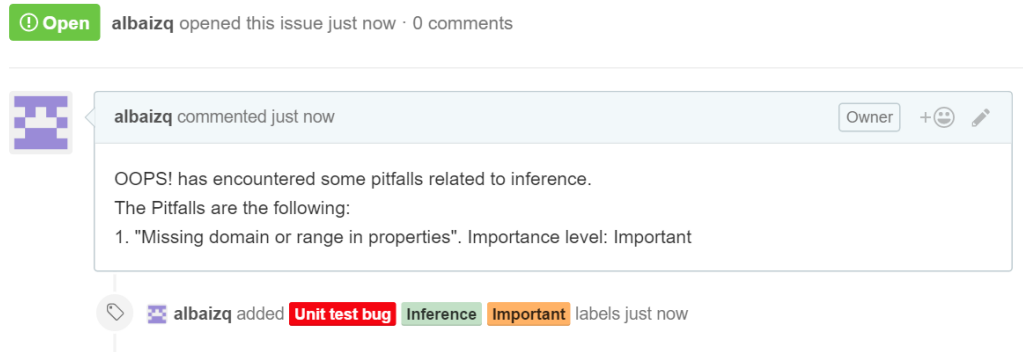


FIGURE 4.9: Inference issue

With these issues the developers can be warned after executing the test that there are two errors in the ontology related to inference and modeling. As this test is executed when there is a change in GitHub, the issues related to the tests are updated continuously.

Acceptance test This test is also executed by Travis CI when there is a change in GitHub. This test, as in Example 1, has to identify the requirements that the ontology does not meet. To validate this, we extract five samples of the user stories given to the developers to create the ontology. This user stories are formalized into SPARQL queries, and then executed over the ontology. This queries are also formalized using both SELECT and ASK queries to test different formats. Since this ontology extracted neither has data, we can only test the ontology structure. The user stories and its results obtained after executing the test are summarized in the following tables:

User story	Represento a una franquicia de panaderías y me interesa conocer cuáles son las calles que se han creado más recientemente y en qué barrios están, para poder determinar si es viable abrir una panadería en alguna de ellas.
Query	PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#> SELECT ?x1 ?x2 WHERE{{ ?x1 ?y1 <http://vocab.linkeddata.es/datosabiertos/def-urbanismo-infraestructuras/callejero#Via FILTER regex(str(?y1), "año", "i")} UNION{ ?x2 ?y2 <http://vocab.linkeddata.es/datosabiertos- /def/urbanismo-infraestructuras/callejero#Via FILTER regex(str(?y2), "barrio", "i")}}
Expected results	cardinality: >0 type: Literal, Literal samples:
Results	cardinality: 0 type: samples:

TABLE 4.6: User story 01. Information about streets

User story	Soy un desarrollador de aplicaciones que necesita obtener las coordenadas de un portal para poder representarlo en un mapa.
Query	PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> ASK{ <http://vocab.linkeddata.es/datosabiertos/def-urbanismo-infraestructuras/callejero#Portal> rdfs:subClassOf <http://www.geonames.org/ontology#Feature> }
Expected results	cardinality: 1 type: Boolean samples: true
Results	cardinality: 1 type: Boolean list of samples: true

TABLE 4.7: User story 12. Coordinates of a building

User story	Estoy haciendo un estudio de localización de franquicias, y además de las calles estoy interesado en los tramos de calle, porque quiero tener un modelo de afluencia de público que considere tramos de calle.
Query	<pre> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> ASK{{ <http://vocab.linkeddata.es/datosabiertos/def- /urbanismo-infraestructuras/callejero#TramoVia> rdfs:subClassOf ?superC ?label ?prop ?superC FILTER regex(str(?label), "afluencia", "i") } UNION{ <http://vocab.linkeddata.es/datosabiertos/def- /urbanismo-infraestructuras/callejero#TramoVia> ?label ?prop FILTER regex(str(?label), "afluencia", "i"),}} </pre>
Expected results	cardinality: 1 type: Boolean samples: true
Results	cardinality: 1 type: Boolean list of samples: false

TABLE 4.8: User story 07. Influx of people in a section of a street

User story	Soy una distribuidora de agua y tengo dos productos diferenciados para particulares y empresas y necesito saber cual es el uso de los portales (comercial? Residencial?) para dirigir una oferta comercial u otra.
Query	<pre> PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#> ASK { ?x ?y <http://vocab.linkeddata.es/datosabiertos- /def/urbanismo-infraestructuras/callejero#Portal> FILTER regex(str(?x), "tipo", "i")} </pre>
Expected results	cardinality: 1 type: Boolean samples: true
Results	cardinality: 1 type: Boolean list of samples: false

TABLE 4.9: User story 14. Information about the type of the buildings

User story	La concejalía de parques y jardines quiere,tener siempre información actualizada sobre,todas las calles y tramos de calles,para poder tener bien posicionados todos los árboles que tienen que podar.
Query	<pre> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> SELECT ?label WHERE {{ <http://vocab.linkeddata.es/datosabiertos/def-urbanismo-infraestructuras/callejero#TramoVia> rdfs:subClassOf ?superC ?urip ?prop1 ?superC . ?urip rdfs:label ?label }UNION{ ?urip ?prop ?urip rdfs:label ?label} UNION { <http://vocab.linkeddata.es/datosabiertos/def-urbanismo-infraestructuras/callejero#Via>rdfs:subClassOf ?superC ?urip ?prop1 ?superC ?urip rdfs:label ?label }UNION { ?urip,?prop ?urip rdfs:label ?label}}</pre>
Expected results	cardinality: >0 type: URIRef samples: Portal Tramo de vía
Results	cardinality: >0 type:URIRef list of samples: nombre internacional Portal Tramo de vía Vía

TABLE 4.10: User story 02. Information about streets and sections of streets

The queries are given to the system in files with extension *.rq* as it is shown in the template in Chapter 4. The results obtained by these queries inform the developers that the ontology does not meet all the requirements given by the domain expert. In this example, the queries associated to User story 12 , User story 14 and User story 07 do not give the correct results to the users. Because of this, the system generate a GitHub issue to inform the developers about the errors. As in the Example 1, Figure 4.10 shows the complete list of issues created for this ontology. The issue generated from this test is the one tagged as “Acceptance test notification”. The content of this issue is shown in 4.11.

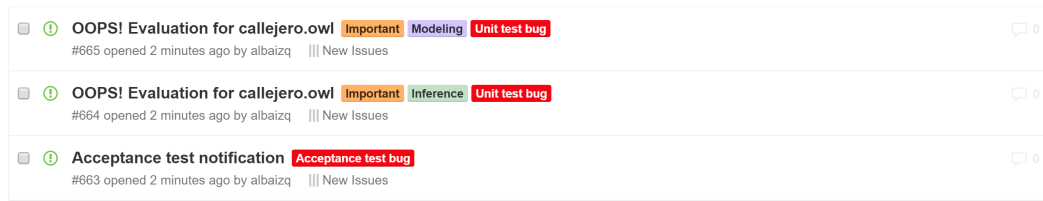


FIGURE 4.10: List of issues

Acceptance test notification #696

Open albaizq opened this issue 24 seconds ago · 0 comments

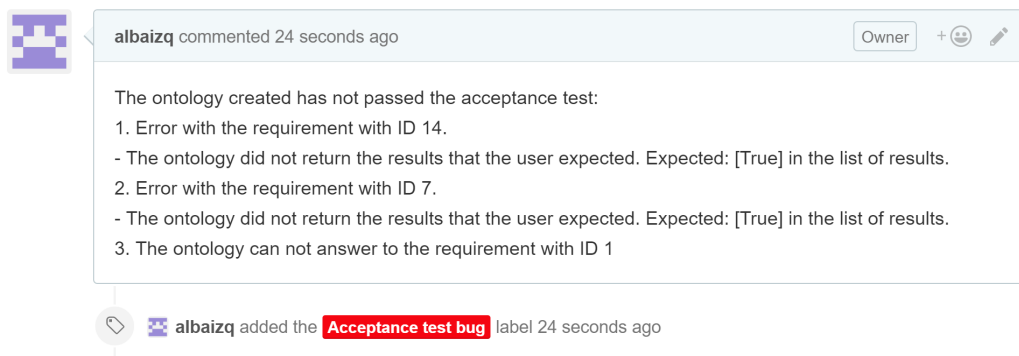


FIGURE 4.11: Content of the issue

The description of the issue notifies that the ontology did not meet the requirements with ID 14, 07 and 01. The issue informs to the user that for the requirements with ID 14 and 07 the system did not return the results expected, as it was expected to return a True value and the results was a False. Furthermore, the ontology can not answer the requirement with ID 01, what means that the ontology did not return any results from the query associated with that requirement.

These issues are continuously updated whenever there is a change in a file, so the developer can be continuously informed about the actual errors encountered in the ontology, both implementation errors and the acceptance ones.

Chapter 5

Conclusions and future work

In this work we have analyzed the most relevant ontology development tools and we compare them in order to check their the main benefits and limitations regarding agile development. As a result, we have determined their main deficiencies.

Starting from agile development processes and the deficiencies of the ontology development tools, we have designed a framework for distributed ontology engineering to structure the entire development process using agile practices to help it. This framework allows the development team to organize their tasks according to their responsibilities. We have also proposed a communication system based on the agile idea of continuous communication and an evaluation system based on the main types of tests that are used in agile software engineering.

The work described in this document is beneficial for distributed ontology engineering since it supports and organizes all the tasks that have to be done during the development process and supports complementary activities like version control or the generation of documentation. As well as software engineering is based on exchanging files, our framework uses a distributed ontology engineering that is mostly done by exchanging OWL files. With this idea in mind, ontology engineering and software engineering are getting closer, making the interaction with each other easier.

As future work, we would like to validate the feasibility of the evaluation system over ontologies with data. Due to the fact that we do not have any data for the ontologies that we use for the validation system we can only test the ontology over its structure. It would be useful to test both ontologies with data and without it in order to analyze the differences in the results. Finally, another future line of work that would be interesting due to the importance of version control in development processes, is to design an ontology version system which can detect pervasive changes in the semantics of two ontologies.

Until now, GitHub, which is the platform that provides us the version control system, can only provide developers information about structural changes over the ontology. This can be enough for software projects but not for ontologies, as it is important to know which terms are added, removed or modified along the different versions. To do that, we can integrate the tool OWL2VCS [33] in our framework, which can give us the deltas among versions with semantic changes using the CEX Logical Diff Algorithm [20].

Bibliography

- [1] Ahmad Alobaid et al. “OnToology, a tool for collaborative development of ontologies”. In: *ICBO2015* (2015).
- [2] Sören Auer and Heinrich Herre. “RapidOWL—An agile knowledge engineering methodology”. In: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer. 2006, pp. 424–430.
- [3] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.
- [4] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [5] Eva Blomqvist et al. “Experimenting with eXtreme design”. In: *International Conference on Knowledge Engineering and Knowledge Management*. Springer. 2010, pp. 120–134.
- [6] Simone Braun et al. “Ontology Maturing: a Collaborative Web 2.0 Approach to Ontology Engineering.” In: *Ckc* 273 (2007).
- [7] Alistair Cockburn. *Crystal clear: a human-powered methodology for small teams*. Pearson Education, 2004.
- [8] Richard Cyganiak et al. “Neologism: Easy Vocabulary Publishing”. In: (2008).
- [9] Mariano Fernández-López, Asunción Gómez-Pérez, and Natalia Juristo. “Methontology: from ontological art towards ontological engineering”. In: (1997).
- [10] Chiara Ghidini, Marco Rospocher, and Luciano Serafini. “Moki: a wiki-based conceptual modeling tool”. In: *Proceedings of the 2010 International Conference on Posters & Demonstrations Track-Volume 658*. CEUR-WS. org. 2010, pp. 77–80.
- [11] Chiara Ghidini et al. “Moki: The enterprise modelling wiki”. In: *The Semantic Web: Research and Applications*. Springer, 2009, pp. 831–835.
- [12] Asunción Gómez-Pérez and Richard Benjamins. “Overview of knowledge sharing and reuse components: Ontologies and problem-solving methods”. In: *IJCAI and the Scandinavian AI Societies*. CEUR Workshop Proceedings. 1999.

- [13] Asuncion Gomez-Perez, Mariano Fernández-López, and Oscar Corcho. *Ontological Engineering: with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web*. Springer Science & Business Media, 2006.
- [14] Lavdim Halilaj et al. “VoCol: An Integrated Environment to Support Collaborative Vocabulary Development with Version Control Systems”. In: (2015).
- [15] Orit Hazzan and Yael Dubinsky. *Agile software engineering*. Springer Science & Business Media, 2009.
- [16] Jim Highsmith and Alistair Cockburn. “Agile software development: The business of innovation”. In: *Computer* 34.9 (2001), pp. 120–127.
- [17] Ernesto Jiménez-Ruiz et al. “ContentCVS: A CVS-based Collaborative ONTology ENgineering Tool.” In: *SWAT4LS*. Citeseer. 2009.
- [18] C Maria Keet and Agnieszka Ławrynowicz. “Test-Driven Development of Ontologies”. In: *International Semantic Web Conference*. Springer. 2016, pp. 642–657.
- [19] Holger Knublauch. “An agile development methodology for knowledge-based systems including a Java framework for knowledge modeling and appropriate tool support”. PhD thesis. Universität Ulm, 2002.
- [20] Boris Konev, Dirk Walther, and Frank Wolter. “The logical difference problem for description logic terminologies”. In: *Automated Reasoning*. Springer, 2008, pp. 259–274.
- [21] Sandra Lovrencic and Mirko Cubrilo. “Ontology evaluation-comprising verification and validation”. In: *CECIIS-2008*. 2008.
- [22] Markus Luczak-Rösch et al. “SVoNt-Version Control of OWL Ontologies on the Concept Level.” In: *GI Jahrestagung (2)* 176 (2010), pp. 79–84.
- [23] Nabil Mohammed Ali Munassar and A Govardhan. “A comparison between five models of software engineering”. In: *IJCSI* 5 (2010), pp. 95–101.
- [24] Adrian Paschke. “OntoMaven API4KB-A Maven-based API for Knowledge Bases.” In: *SWAT4LS*. 2013.
- [25] Niklas Petersen et al. “VoCol: An Agile Methodology and Environment for Collaborative Vocabulary Development”. In: (2015).
- [26] María Poveda-Villalón, Mari Carmen Suarez-Figueroa, and Asunción Gómez-Pérez. “Validating ontologies with OOPS!” In: Springer, 2012, pp. 267–281.

- [27] Ian Sommerville. “Software Engineering. International computer science series”. In: *ed: Addison Wesley* (2004).
- [28] Armando Stellato et al. “VocBench: a web application for collaborative development of multilingual thesauri”. In: *The Semantic Web. Latest Advances and New Domains*. Springer, 2015, pp. 38–53.
- [29] Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, and Boris Villazón-Terrazas. “How to write and use the ontology requirements specification document”. In: *On the move to meaningful internet systems: OTM 2009*. Springer, 2009, pp. 966–982.
- [30] Sebastian Tramp and Norman Heino. “OntoWiki a Semantic Data Wiki Enabling the Collaborative Creation and (Linked Data) Publication of RDF Knowledge Bases”. In: (2010).
- [31] Tania Tudorache, Jennifer Vendetti, and Natalya Fridman Noy. “Web-Protege: A Lightweight OWL Ontology Editor for the Web.” In: *OWLED*. Vol. 432. 2008.
- [32] Denny Vrandečić and Aldo Gangemi. “Unit tests for ontologies”. In: *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*. Springer. 2006, pp. 1012–1020.
- [33] Ivan Zaikin and Anatoly Tuzovsky. “Owl2vcs: Tools for Distributed Ontology Development.” In: *OWLED*. Citeseer. 2013.