applied sciences

MDPI

Article

Ontology Fixing by Using Software Engineering Technology

Gabriela R. Roldan-Molina [1], Jose R. Mendez [1,2,3,*], Iryna Yevseyeva [4] and Vitor Basto-Fernandes [5]

1 Department of Computer Science, University of Vigo, ESEI-Escuela Superior de Ingeniería Informática, Edificio Politécnico, Campus Universitario As Lagoas s/n, 32004 Ourense, Spain; groldan@uvigo.es
2 CINBIO-Biomedical Research Centre, University of Vigo, Campus Universitario Lagoas-Marcosende, 36310 Vigo, Spain
3 SING Research Group, Galicia Sur Health Research Institute (IIS Galicia Sur), SERGAS-UVIGO, 36312 Vigo, Spain
4 Cyber Technology Institute, School of Computer Science and Informatics, Faculty of Computing, Engineering & Media, De Montfort University, Gateway House, The Gateway, Leicester LE1 9BH, UK; iryna@dmu.ac.uk
5 Instituto Universitário de Lisboa (ISCTE-IUL), University Institute of Lisbon, ISTAR-IUL, Av. das Forças Armadas, 1649-026 Lisboa, PT, Portugal; vitor.basto.fernandes@iscte-iul.pt
* Correspondence: moncho.mendez@uvigo.es; Tel.: +34-988-387-015

Received: 31 July 2020; Accepted: 9 September 2020; Published: 11 September 2020

Abstract: This paper presents OntologyFixer, a web-based tool that supports a methodology to build, assess, and improve the quality of ontology web language (OWL) ontologies. Using our software, knowledge engineers are able to fix low-quality OWL ontologies (such as those created from natural language documents using ontology learning processes). The fixing process is guided by a set of metrics and fixing mechanisms provided by the tool, and executed primarily through automated changes (inspired by quick fix actions used in the software engineering domain). To evaluate the quality, the tool supports numerical and graphical quality assessments, focusing on ontology content and structure attributes. This tool follows principles, and provides features, typical of scientific software, including user parameter requests, logging, multithreading execution, and experiment repeatability, among others. OntologyFixer architecture takes advantage of model view controller (MVC), strategy, template, and factory design patterns; and decouples graphical user interfaces (GUI) from ontology quality metrics, ontology fixing, and REST (REpresentational State Transfer) API (Application Programming Interface) components (used for pitfall identification, and ontology evaluation). We also separate part of the OntologyFixer functionality into a new package called OntoMetrics, which focuses on the identification of symptoms and the evaluation of the quality of ontologies. Finally, OntologyFixer provides mechanisms to easily develop and integrate new quick fix methods.

Keywords: ontologies; fixing ontologies; quick fix; quality metrics

1. Introduction and Motivation

Ontologies are knowledge representations, in which concepts and categories of a certain domain are stored together with their properties and the relations between them. Currently, ontologies are used to represent knowledge from a large number of domains in order to solve different problems and improve the experience of users in different contexts. For example, in the Semantic Web, they are used to describe terms, retrieve information, and interconnect web services. Due to the increasing use of ontologies, a large number of models and languages have been introduced to manage them, including resource description framework (RDF) [1], resource description framework schema (RDFS) [2],

and ontology web language (OWL) [3], among others. However, it is necessary to be able to evaluate the quality of the creation of these ontologies in order to guarantee good performance, and take advantage of the benefits they offer. Currently, ontologies facilitate aspects such as communication, interoperability, and automatic reasoning [4–7]. Ontologies allow us to represent and share knowledge using a common vocabulary, and to exchange data between different systems and contexts [8].

Furthermore, ontologies are very useful to facilitate automatic reasoning. On the basis of inferencing, a reasoning engine can use the ontology data (categories, concepts, relations, and properties) to reach conclusions. On the other hand, the use of ontologies allows knowledge engineers to organize and structure the information so that software agents can interpret their meaning, and, consequently, search and integrate data much better. Using the knowledge stored in ontologies, applications can automatically extract data from web pages, process them, draw conclusions, make decisions, and negotiate with other agents or people [9,10].

A significant number of ontologies are manually generated or created by taking advantage of applications, and implementing the extraction of information from natural language text [11]. The use of these tools in conjunction with the natural inconsistencies of human languages can lead to the appearance of errors, inconsistencies, or bad designs that require further debugging or repair processes. The process of detecting and fixing errors or bad design symptoms is a difficult task and should be done as an iterative process, where each step should include the evaluation of the state of the ontology, selecting the most appropriate change for the current situation and evaluating whether the changes made are appropriate.

The detection of ontology errors, inconsistencies, and flaws can be made using tools such as OOPS! [12], a web application that detects bad practices when modelling ontologies. This tool provides mechanisms for automatically detecting potential errors, called pitfalls, in order to help developers during the validation process. However, some pitfalls are detected semi-automatically, such as "Creating synonyms as classes" or "Creating unconnected ontology elements", among others [13]. Each pitfall provides the following information: title, description, elements affected, and importance level. A recent study [14] showed a method for finding errors in apparently coherent and consistent ontologies, but these may contain contradictions in the axiom statements and provide incorrect information. The approach uses knowledge from other knowledge bases that debug ontology modelling errors. Moreover, some frameworks have been introduced for measuring the quality of ontologies, most notably OquaRE [15], which implements several quality metrics based on the SQuaRE (system and software quality requirements and evaluation) [16] software quality standard. This framework provides a guide to evaluate the quality of ontologies in diverse dimensions, such as reliability, operability, maintainability, compatibility, transferability, and functional adequacy. Although some tools for error detection and/or quality evaluation of ontologies are available, the process of aiding the debugging/fixing of ontologies has not been addressed in a global form. Particularly, we found that these tools could be combined to create a tool to fix errors, and improve the global quality of ontologies. Additionally, taking advantage of quick fix schemes used in integrated development environments (IDEs), the fixing tool could suggest appropriate solutions to address each detected trouble, to simplify the fixing process. In this study, we developed a software tool implementing the proposed solution for ontology fixing, detailing its architecture, functionality, and usage. The software integrates a wide amount of software technology that has been successfully adapted to the context of knowledge engineering (OQuaRE, pitfalls, quick fix schemes, etc.). The result is the creation of the OntologyFixer tool, which can be successfully downloaded from GitHUB (Available at https://github.com/gabyluna/OntologyFixer) and is available on http://ontologyfixer.online.

The remainder of the paper is structured as follows: Section 2 presents the state of the art in the context of repairing ontologies. Section 3 presents the architecture of the developed software in detail. Section 4 shows the main features of the generated software. Finally, Section 5 shows the main conclusions, and future developments to complement this work.

2. State of the Art

Given the increased use among software and knowledge engineering communities of ontologies to represent knowledge, their quality and correctness have become two key aspects to consider. Quality evaluation assists in finding design defects, inconsistencies, errors, or limitations in stored knowledge. Moreover, the correction of an ontology implies the detection of a problem, the exploration of possible ways to fix it, and the application of the selected correction. This section compiles previous studies that have introduced algorithms and techniques to implement these functionalities, which are key to the implementation of OntologyFixer.

The evaluation of ontology quality has been addressed in previous resources including (ordered from most to least recent): (i) OquaRE [15], (ii) Foval [17], (iii) OntoQA [18], and (iv) OntoClean [19].

OQuaRE is a method of evaluating the quality of ontologies that emerged as an adaptation of the SQuaRE (system and software quality requirements and evaluation) standard (ISO/IEC 25000) to the context of knowledge engineering, comprising evaluation support, evaluation processes, and quality metrics. OQuaRE uses different metrics to assess the quality of the ontologies with regard to different dimensions, including reliability, operability, maintainability, compatibility, transferability, and functional adequacy. Most quality sub-characteristics suggested by SQuaRE (system and software quality requirements and evaluation) [20] were also adopted in OQuaRE. Additionally, OQuaRE includes the structural characteristic, which is important in evaluating ontologies.

FOval provides an evaluation model to select ontologies that best fit the user needs (requirements), while OntoQA is a suite of metrics that evaluates the quality of ontologies in different dimensions, including schemas, knowledge base, and class metrics. Finally, OntoClean is a methodology for the validation of the ontological adequacy and logical consistency of taxonomic relationships. All of these works provide interesting measures to assess different aspects of quality including (i) lexical, (ii) hierarchy, (iii) other semantic relationships, (iv) context, (v) syntactic, and (vi) structure, as recommended in a previous study [21].

As in the case of OQuaRE, software engineering domain technology has inspired some proposals that aid in the detection of the potential troubles of ontologies. Particularly, pitfall, code smells, or simply smells, were popular forms of symptoms of software design troubles [22]. A recent study [12] introduced how the same concept (pitfall) can be applied to the context of knowledge engineering, to address the detection of design troubles. Given the success of the application of these software engineering technologies, we suggest the possibility of using quick-fix schemes to improve some knowledge engineering design processes. Introducing these concepts, in the context of ontologies, and combining them with current technology will lead to new, and better, ontology fixing tools (OntologyFixer).

The next section describes how quick-fix support has been included in our proposal, to be used as an aid in fixing and improving the quality of ontologies.

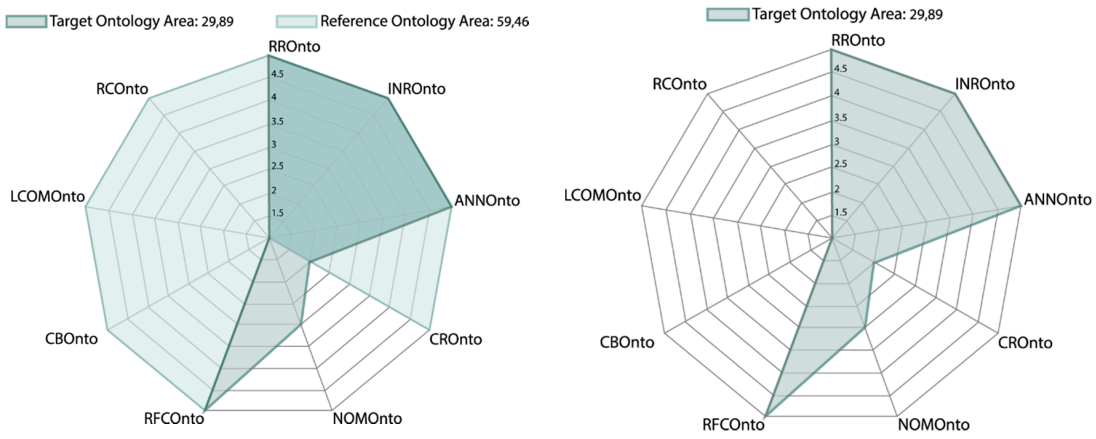
3. System Overview

OntologyFixer diagnoses the quality of the ontologies, and also allows for the detection and correction of errors. For the diagnosis, OntologyFixer applies different metrics that allow measuring different aspects of the quality of ontologies, such as structure, logic, and semantics, among others. Table 1 shows the measures we have selected for the evaluation of ontologies.

Table 1. Quality Metrics.

Metric	Description
ANOnto	Measures annotation richness
CBOnto	Determines coupling between Objects
CROnto	Assess Class Richness
INROnto	Number of relationships per class
LCOMOnto	Measures the lack of Cohesion in Methods
NOMOnto	Finds the number of properties per class
RCOnto	Instances distributed across classes
RFCOnto	Determines the response measure for a class
RROnto	Assess relationship richness

Most of the selected metrics are available in the OQuaRE framework except for RCOnto, which is provided by the OntoQA framework. Assuming the selected measures should be maximized, we represent them in a radar chart, and compute the area of the polygon formed using the measures evaluated as vertices. OntologyFixer supports the loading of an additional ontology (as a reference), to compare its quality with that of the ontology being corrected. Figure 1 shows an example of a diagnosis generated by the application.



(**a**) (**b**)

Figure 1. Ontology quality assessment result: (**a**) Quality comparison with a reference ontology, (**b**) Quality evaluation of an ontology.

As shown in Figure 1, the application can also compare a model ontology with the ontology to be evaluated (Figure 1a), allowing the user to better visualize the weaknesses of the ontology that is being fixed. However, it is also possible to visualize the diagnosis of the quality of the ontology without the need to load a reference ontology (Figure 1b).

Additionally, OntologyFixer integrated OOPS! [12], to detect errors or pitfalls. OOPS! is a framework that detects pitfalls, and prioritizes them according to their importance. OntologyFixer implements a quick fix strategy to automate the correction of ontologies. Using OOPS!, OntologyFixer is able to provide complete information about the errors of an ontology.

Currently, only two quick fixes were implemented in the first version of the application: (i) RM_INVOLVED_ELEMENTS and (ii) RM_SIMILAR_ELEMENTS. The former removes some (or all) ontology elements (classes, object properties, or data properties) that are causing troubles. The latter searches and removes similar elements caused by typos (and the use of ontology automatic generation tools) that are causing a pitfall in the ontology. To carry out this process, we applied the Levenshtein algorithm [23] to find the lexical distance between two words. This quick fix removes elements having a distance lower than, or equal to, 1. For example, when comparing the elements

"action" and "actions", which could cause circularity in the ontology, the distance calculated between them is 1; that is, there is a similarity between the terms so that when the RM_SIMILAR_ELEMENTS quick fix is applied, one of the elements is removed from the ontology in conjunction with the axioms related to it.

When the available quick fixes are not adequate, OntologyFixer allows downloading of the current status of the ontology for manual editing. This allows users to modify the ontology using their favourite ontology editor (e.g., Protégé), and to then upload the resulting ontology again.

Another functionality of OntologyFixer is the possibility of performing a rollback of the ontology; that is, the application has a history of versions with the possibility of returning to an earlier version, which allows undoing changes, with respect to a quick fix previously applied. To this end, OntologyFixer stores ontology snapshots before applying any operation (quick fix or manual edition). Nevertheless, before applying a quick fix, OntologyFixer shows a detailed description of the actions that are going to be performed to ensure the user agrees with the action.

One notable advantage of OntologyFixer is the evaluation and correction of errors without needing to use multiple applications. Another advantage, is the possibility of integrating new quality metrics and quick fix schemes, due to the scalable and decoupled nature of the tool's design.

The following subsections provide a brief description of the architecture of the application (Section 3.1), the integration with frameworks and APIs (Section 3.2), and the metrics adopted for the research project (Section 3.3).

3.1. General Architecture

OntologyFixer was designed following a web-based client/server scheme. This section shows the design of the architecture in detail as well as the main features of the application. Figure 2 shows the interaction of web browsers (clients) with OntologyFixer, which was deployed in a J2EE Application Server.

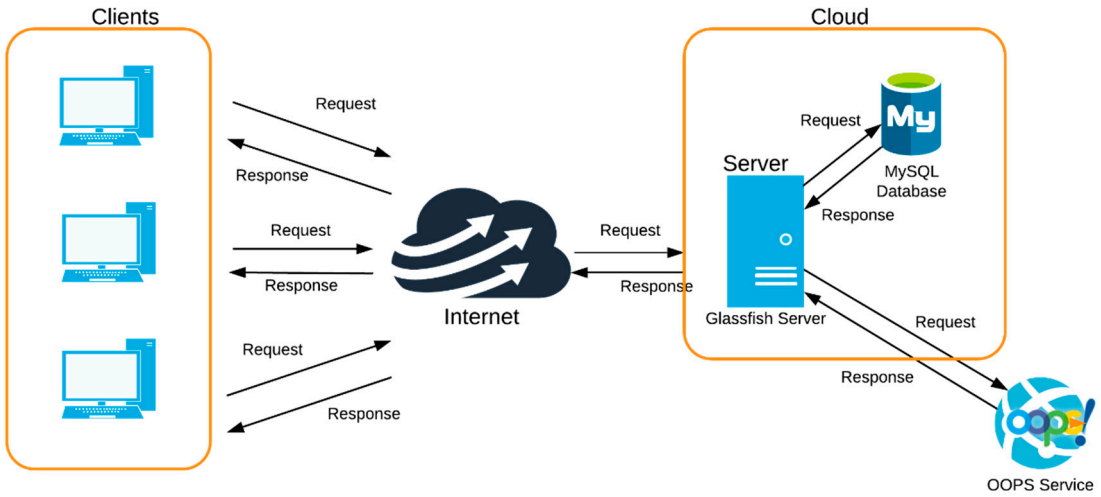


Figure 2. Application architecture.

As shown in Figure 2, user interaction takes place through a web browser, sending HTTPS requests to a Glassfish application server running in the cloud. Persistence is supported by a Spring ORM (object relational mapping) implementation that transparently manages the information stored in a MySQL database engine. Additionally, the external RESTful Web Service (OOPS!) [24] API is used for the evaluation and detection of errors in the ontologies. Figure 3 shows a set of technologies separated into different layers, in order to delegate specific functions for each of them, promoting the software development single responsibility principle [25]. Each of the technologies was specially selected for

the development of OntologyFixer, for its ease of integration, robustness, and availability as open source software.

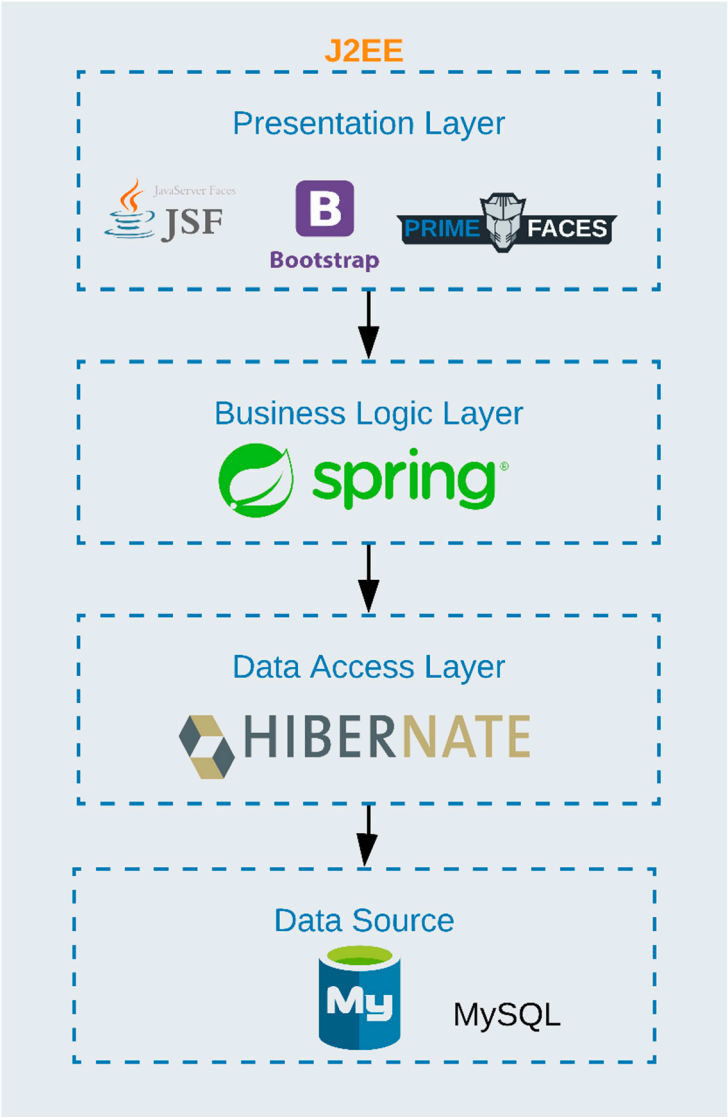


Figure 3. Layered architecture.

As shown in Figure 3, OntologyFixer implements a multiple-layer design in which the functionalities defined in the lower layers provide access to resources or services to the upper layers. The main advantage of this type of architecture is the ability to build scalable applications that favor the integration of other components, software maintenance, and evolution. The presentation layer uses web technologies such as JSF (Java Server Faces) version 2.2 [26]. JSF is a user-interface framework implementing the MVC (model view controller) architecture pattern, which facilitates the development and maintenance of web applications. Additionally, PrimeFaces framework version 5.3, which provides open source visual components for JSF 2.2 and Bootstrap [27], was adopted to provide a responsive interface design that can be easily used in a wide range of computing devices. Finally, for ontology visualization, we took advantage of the WebVOWL (Web-based Visualization of Ontologies) (Available at http://vowl.visualdataweb.org/webvowl.html) external web application.

The business logic layer was implemented using Spring framework because of its relevant features (i.e., "Dependency Injection" or "Inversion of Control"). Currently, Spring provides a wide variety of functionalities in the form of modules, including Spring Security, Spring AOP, Spring JPA, etc.

Spring JPA and Hibernate were used for the development of the persistence and access layer of OntologyFixer. These frameworks allowed the transparent access to the data stored in a MySQL database. The next subsection introduces the main design patterns that were used in the development of OntologyFixer and its main purpose.

3.2. Software Design Details

The MVC [28] design pattern was adopted for the software development, as it presents well-known robust properties and software quality attributes. MVC separates business logic with respect to the data (model) and the user interface (view/GUI). It allows independent changes in each component without affecting the others. In other words, changes in the graphical user interface (GUI) do not affect data handling, and data can be reorganized without changing the user interface. Figure 4 shows the components that are part of the model.

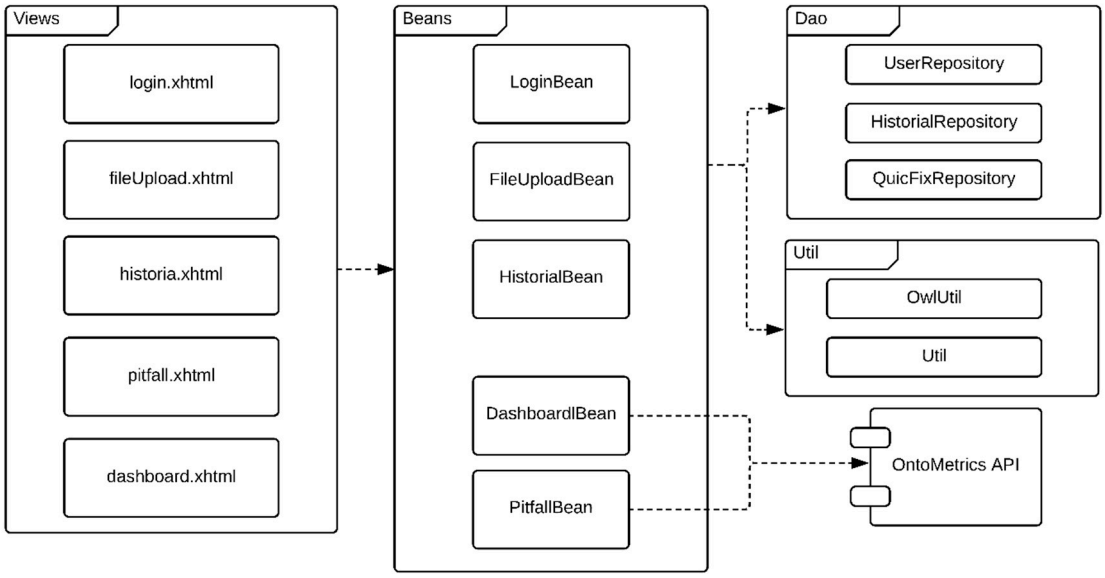


Figure 4. Model View Controller design.

In the Views component shown in Figure 4 we can see the user interface for each functionality. The controller uses Beans (one per view) that are responsible for making the connection between the view and the application logic. Finally, the model component uses Spring JPA to access the data layer, i.e., to make the call to the persistence layer that communicates with the MySQL database. To do so, Spring uses the DAO (data access object) objects [29], which are design patterns, in which a data access object provides an abstract interface to some type of database or other persistence technology. DAOs provide some specific data operations without exposing database data model details (i.e., create, update, or delete).

Additionally, OntologyFixer integrates an external OntoMetrics API, developed in the context of this research for computing the different metrics used to evaluate the quality of the ontologies. This API was integrated as a Maven dependency in the OntologyFixer project. OntoMetrics were structured into three modules: basic metrics, quality metrics, and symptom identification methods.

The development of this API involved the use of some design patterns that facilitate software comprehension, maintenance, and evolution. The behavior pattern strategy [30] was used to define a family of algorithms, each of them in separate classes, making their objects interchangeable. Figure 5 shows the design of the OntoMetrics API package.

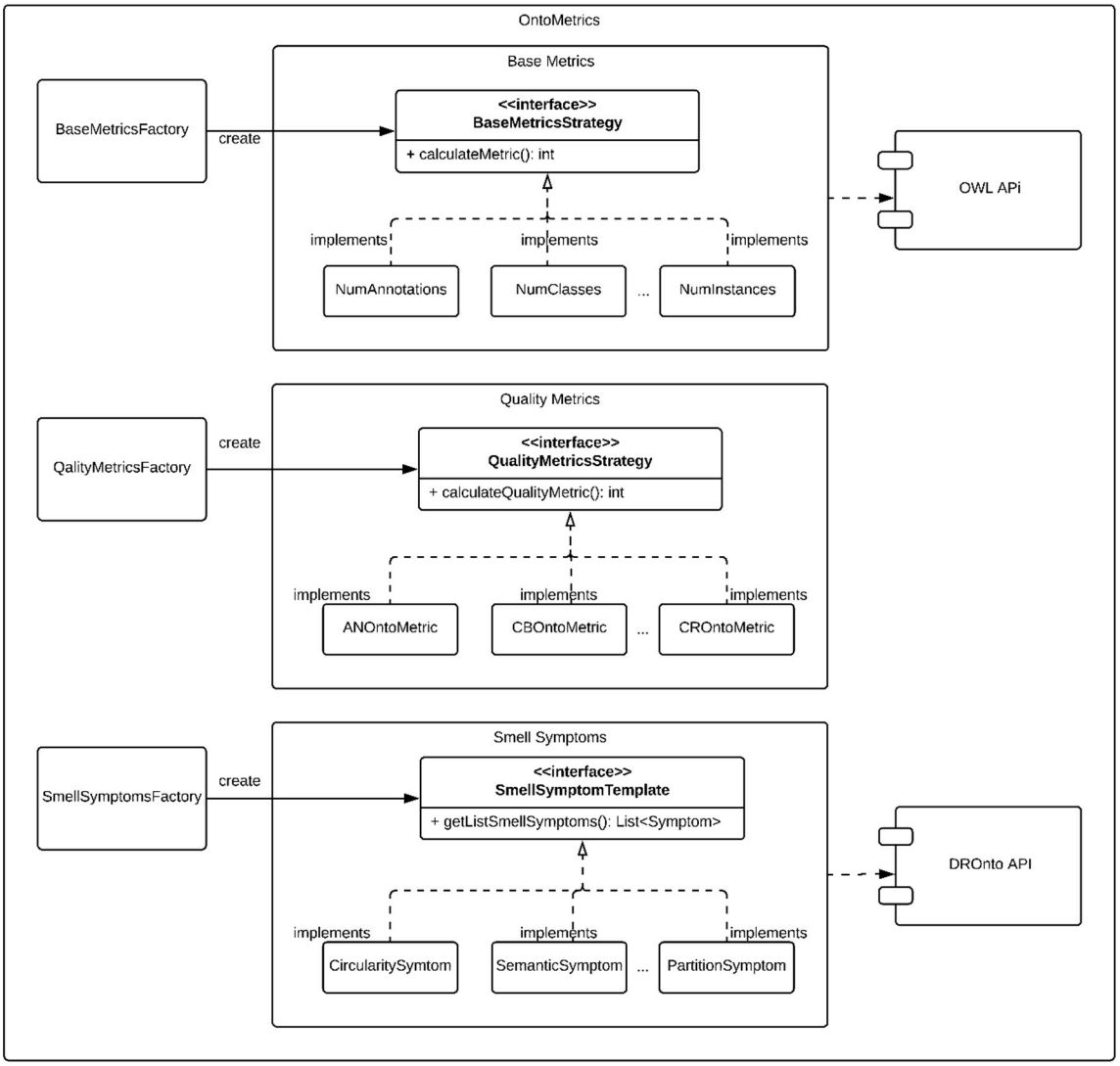


Figure 5. OntoMetrics design.

As shown in Figure 5, the implementation of the package includes a Strategy interface, implemented by strategy classes for basic and quality metrics. Moreover, the Template pattern [30] was used to implement smell symptom identification. Finally, we used the Factory design pattern [30] for creating objects, without having to specify their exact class. This Creational pattern avoids close coupling between the creator and concrete products. In addition, it complies with the principle of sole responsibility because it can move the product creation code to a specific place (the Factory class) in the program, making the code easier to maintain. In addition, by combining the selected patterns, the use of the metrics is clearly easier for developers. Figure 6 shows different examples of using the metrics included in the library.

```
01  //Example BaseMetrics (Number of annotations of ontology)
02  BaseMetricsStrategy baseMetricsStrategy;
03  baseMetricsStrategy = BaseMetricsFactory.getBaseMetric(BaseMetricEnum.ANNOTATIONS);
04  int annotations = baseMetricsStrategy.calculateMetric(ontology);
05
06  //Example QualityMetrics
07  QualityMetricsStrategy qualityMetricsStrategy;
08  QualityMetricFactory qualityMetricFactory = new QualityMetricFactory();
09  qualityMetricsStrategy = qualityMetricFactory.getQualityMetric(indexMetric);
10  resultMetric =  qualityMetricsStrategy.calculateQualityMetric(metricsOntology);
11
12  //Example SmellSymptoms
13  SmellSymptomTemplate circularitySymptomTemplate =
14       SmellSymptomFactory.getSmellError(SmellSymptom.CIRCULARITY);
15  List<Symptoms>  listCircularitySymptoms =
16       circularitySymptomTemplate.getListSmellSymptoms(pathOntology);
```

Figure 6. Example of using different kinds of metrics from the package OntoMetrics.

Another of the main features of OntologyFixer is the possibility of applying quick fixes. The application was designed with an architecture that allows for the easy addition of new quick fixes, to improve ontologies. Figure 7 shows the QuickFixInterface interface, which specifies the methods that should be implemented to develop additional quick fixes.

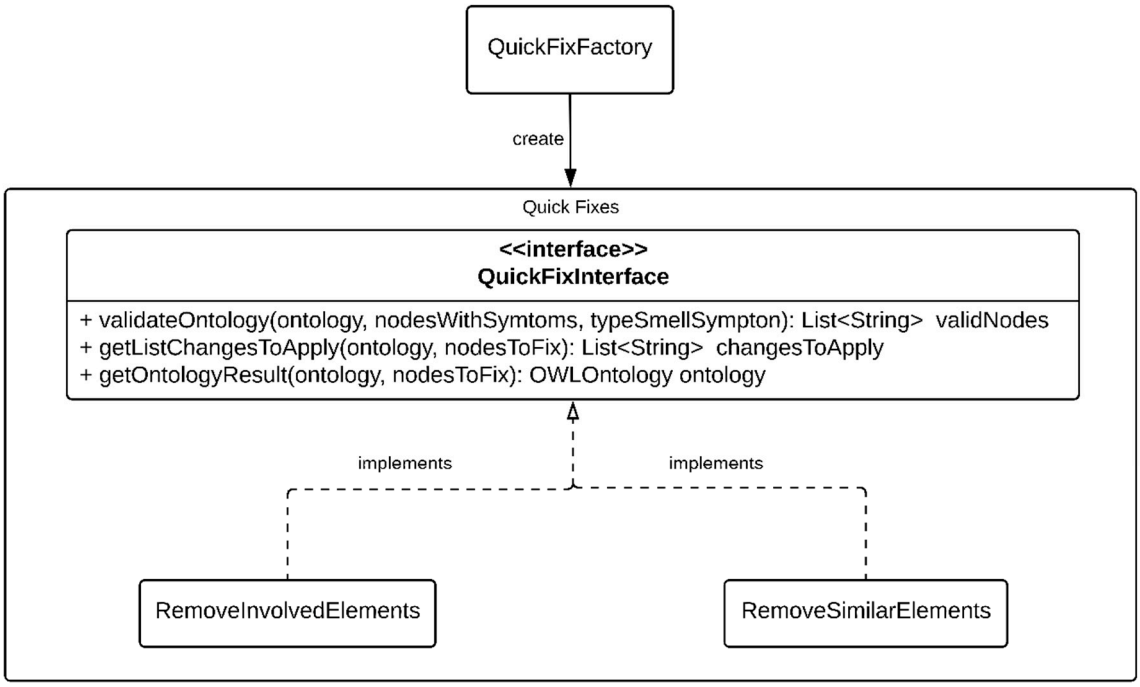


Figure 7. QuickFix interface.

In order to develop a new quick fix scheme, three methods should be implemented. The first one (validateOntology) determines which nodes of an ontology that have a problem can be fixed by applying the quick fix. Moreover, getListChangesToAppy finds the list of changes that are required for fixing certain nodes of the ontology using the quick fix. Finally, the getResult method allows the quick fix to be applied to certain nodes of an ontology. Currently, two quick fix methods (RemoveInvolvedElements and RemoveSimilarElements) are supported by OntologyFixer. However, the inclusion of new quick fix schemes can be easily added to the design by implementing the QuickFixInterface and registering the implementation in the QuickFixFactory class. The simplification and extensible design of the architecture ensures new quick solutions can be easily added to OntologyFixer.

The next subsection compiles the metrics that were included in the OntoMetrics package.

3.3. Implemented Metrics and Symptoms Detection

This subsection provides a complete list of the smell symptoms detection mechanisms and the metrics (basic and quality) implemented by OntoMetrics API, as well as some of the implementation details.

To compute basic metrics, we used the OWL API [31], which manages the ontologies and provides components for the manipulation of ontological structures in different formats, such as OWL and RDF (resource description framework) among others. In addition, we also used reasoning engines. Eight basic metrics were implemented. Table 2 describes each of the basic metrics.

Table 2. Basic metrics implemented in OntologyFixer.

Metric	Description
Number of annotations	Indicates the total entries that exist in the ontology.
Number of classes	Finds how many classes exist in ontology.
Number of classes with individuals	Shows the number of classes that have at least one individual.
Number of instances	Computes the number of instances of the ontology.
Number of properties	Indicates the number of properties contained in the ontology.
Number of relations of Thing	Stands for the relationships that exist towards "Thing".
Number of subclasses	Counts the total subclasses that exist in the ontology.
Number of superclasses	Identifies the number of superclasses that exist in the ontology.

The implementation of quality metrics was based on the OQuaRE (framework described in Section 2), which provides a guide to evaluate the quality of ontologies in diverse dimensions, such as reliability, operability, maintainability, compatibility, transferability, and functional adequacy. Nine quality metrics (see Table 1) were implemented to assess the quality of ontologies by using some basic metrics.

Finally, for identifying smell symptoms, the web service that provides OOPS! [24] was used. The service takes a file with an OWL extension of the ontology as input, evaluates it, and returns a list of smell errors, with additional fields to identify the level of criticality it represents in the ontology. The smell symptoms described in Table 3 were implemented for the project.

Table 3. Smell Symptoms ("ONTOLOGICAL ERRORS—Inconsistency, Incompleteness, and Redundancy" 2008).

Smell Symptom	Description
CircularitySymptoms	Detects cycles between two (or more) classes
IncompletenessSymptoms	The symptom entails not representing all the knowledge that could be included in the ontology.
PartitionSymptoms	Detects symptoms when disjoint decomposition exists. There are three types: Common Instances and Classes in Disjoint, Decomposition and Partitions.
SemanticSymptoms	The symptom entails problems in the logic between elements and relationships of the ontology.

The next subsection presents the outcomes that emerged from the development of the tool.

3.4. Lessons Learned

This subsection identifies the main outcomes achieved by carrying out this study. The most important conclusions are related to the successful testing and validation of software engineering approaches, methods, and technologies, adapted and applied to knowledge engineering problems. After their incorporation into the domain, the identification of bad design symptoms (smells), or the use of quality metrics (OOPS!), we were able to successfully adapt quick fix mechanisms, included in popular integrated development environments (IDEs), to improve the quality of the ontologies.

Additionally, the combination of quality metrics, errors or smell symptoms detection, and quick fix for developing an ontology fixing tool seems to be very reliable, as demonstrated in this study. However, at the moment there is no quick fix tools for ontology-based knowledge representation available, and we have to implement them following strategies similar to those used by software IDEs.

As a part of this study, we provided a collection of ontology quality metrics, the error detection, and the identification of bad design symptoms for the OntoMetrics library. This facilitates the use of these metrics, by providing a uniform way to access these functionalities. For the evaluation of quality, OntoMetrics takes advantage of the metrics defined in the OQUARE framework. In the case of symptom detection, OOPS! was integrated into our OntoMetrics library, to take advantage of DrontoAPI to detect smells in analyzed ontologies. DrontoAPI functionalities are provided through a web service which incorporates methods to find the elements of the ontology that are affected by any type of symptom (circularity, incompleteness, semantics).

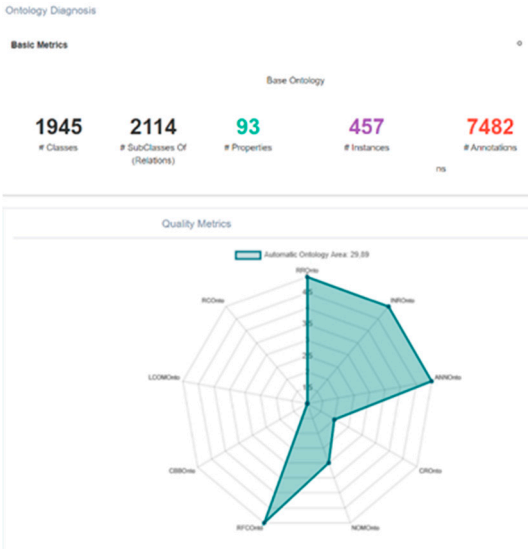
Finally, from a more technical perspective, we also employed different tools including Spring, an open source application development framework for the Java platform. Spring is based on different design patterns including DAO (data access object). This pattern is used to encapsulate data access logic, thus avoiding mixing it with business logic. Spring also provides a consistent approach to data access, either JDBC (Java DataBase Connectivity), or through some data access frameworks, such as Hibernate, iBatis, JDO, or TopLink, among others, and allows changing the framework used for persistence without affecting the code already written. For the construction of OntologyFixer we used PrimeFaces, a library of visual components for Java Server Faces (JSF) providing a large number of elements, to make the development of the presentation layer easier. A notable advantage of using this library is the Ajax support for updating the components and achieving a better user experience. Furthermore, to enhance the interface, we used Bootstrap, which embeds technologies such as JavaScript and CSS (Cascading Style Sheets) in order to help developers quickly and efficiently design a responsive website, and to make the design correct and usable both for conventional and tactile devices (responsive web design). The combination of these tools makes the application more robust and scalable over time, with the possibility of adding or changing new technologies that fit the needs of the project.

The functionality of the resulting application is shown in the next section. Particularly, it highlights some interface details and operations by using some screenshots and providing a detailed description of the inner operation of OntologyFixer.

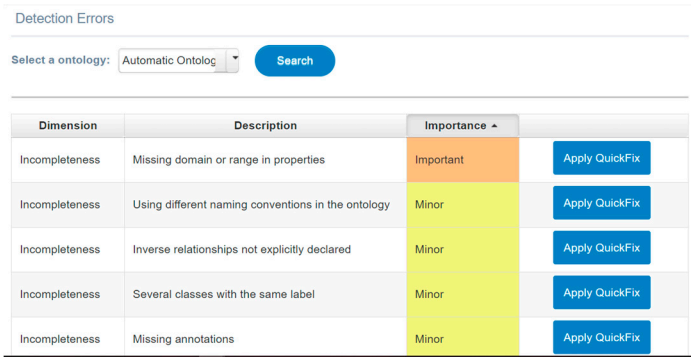
4. System Use

OntologyFixer was designed with a user friendly and easy to use graphical interface. It allows a user to (i) upload an ontology, (ii) evaluate ontology quality, (iii) apply fixes to the ontology, and (iv) use the changes history to restore a previous state when needed. An ontology can be evaluated by applying two different strategies: performing a metrics-based quality evaluation, and finding possible troubles (errors or symptoms of bad design) that may lead to ontology inconsistencies. Figure 8 shows the interface details for both functionalities.

As shown in Figure 8a, the evaluation of the quality of Ontologies is made in the form of a radial chart combining different quality measures. The quality is shown in a dashboard which includes "Base Metrics" (located at the top of the figure), which include the number of classes, subclasses, properties, instances, and annotations of the ontology to evaluate and compare with the reference ontology (optional). The bottom of the dashboard contains a quality evaluation chart that perceives the quality of the ontology at a simple glance. The chart represents RROnto, INROnto, ANNOnto, CROnto, NOMOnto, RFCOnto, CBOOnto, LCOMOnto, and RCOnto scores (see Table 2), achieved by the ontology. Additionally, the area of the figure described by the representation of these measurements (that is also shown) could be used to assess the global quality of the ontology. Finally, the knowledge engineer can graphically visualize the ontology itself if desired.



(**a**)

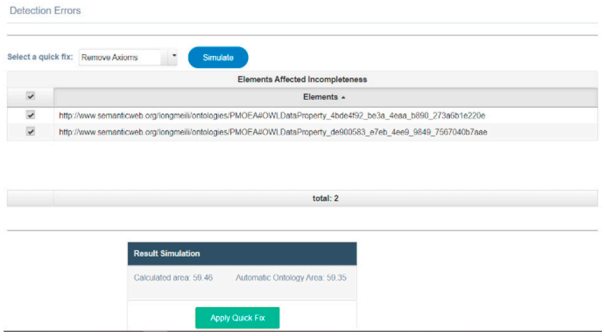


(**b**)

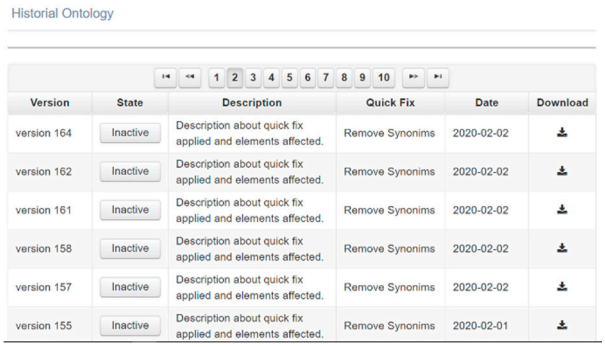
Figure 8. Different forms of ontology diagnosis implemented in OntologyFixer. (**a**) Quality evaluation. (**b**) Errors and smells detection.

As shown in Figure 8b, the features allowing the detection of errors and smells enable the knowledge engineer to identify and address specific troubles found in the ontology. The results table included in Figure 8b details the trouble found in the ontology (circularity, incompleteness, and semantics), a brief description for it, and its level of importance. The level of importance (which is assessed by using the OOPS! framework) is highlighted using different background colors. Specifically, critical errors are highlighted in red, important troubles are marked in orange, and finally, minor issues are represented with a yellow background.

Once the main weaknesses and errors of the ontology have been identified, OntologyFixer supports the application of a quick-fix and the upload of a new version of the ontology. Additionally, OntologyFixer stores snapshots of the ontology each time a modification is made; this ensures that an older version of the ontology can be restored. These snapshots are stored in a local database but also in a Git repository (if configured) to ensure the versions of the ontology are shared according to the preferences of the user. Figure 9 shows capabilities of the application for fixing ontologies.



(**a**)



(**b**)

Figure 9. Fixing errors in ontologies. (**a**) Applying a quick fix. (**b**) History of modifications.

Figure 9a shows the "Apply Quick Fix" button, which allows the execution of a quick fix to those elements of the ontology (instances, classes, relationships) affected by some type of issue. This button is used to compute the list of all affected items for an error. As shown in Figure 9a, OntologyFixer users can select one or more elements found, and apply one of the quick fixes described in Section 3. One of the functionalities of OntologyFixer is the possibility of simulating the results of applying a quick fix, which is useful for deciding whether to apply a certain solution. The simulation result shows a value for the current area and the new area calculated after applying the quick fix in the ontology (Figure 9a). Finally, the "Apply Quick Fix" button shown in Figure 9a will generate a new version of the ontology and store it as a new version in history.

As described in Section 3, one of the advantages of the application is the possibility of performing a rollback in the ontology editions (Figure 9b). The button included in the "Status" column can be used to return to an earlier version of the ontology. This action allows us to activate an older version of the ontology, re-analyze, and fix it by applying a new quick fix. In addition, the rollback function included in OntologyFixer offers the possibility of downloading each of the versions to which changes were applied (button located at "Download" column). The next section presents the main conclusions extracted from this work and outlines future research directions.

5. Conclusions and Future Work

This work introduces OntologyFixer, a tool for assisting users in improving the quality of ontologies and fixing their troubles, by using software engineering inspired techniques. The functionality of OntologyFixer was achieved by combining some recently introduced techniques that are inspired by software engineering (e.g., quality evaluation frameworks, such as OQUARE, or smell symptoms detection, such as OOPS!), and others that are being introduced and used for the first time in this study (quick fix for ontologies). OntologyFixer combines 17 measures to assess the quality of an

ontology and supports the detection of four types of bad design symptoms (smells). This group of functionalities was encapsulated into an external library (OntoMetrics) to standardize their invocation. Additionally, the application was developed by incorporating recent and innovative frameworks (Spring, PrimeFaces, Bootstrap) to ensure robustness, usability, and ease of maintenance.

The use of quick fix technology, extracted from popular IDEs for developing software, provides an easy to use and customizable mechanism to fix ontologies more easily. Particularly, the application of quick fix methods implies developing several methods that (i) determine which issues can be fixed by using the quick fix, (ii) provide a list of changes that will be made through applying the quick fix, and (iii) apply the changes. OntologyFixer functionalities were integrated into a graphical interface that allows the user to identify those elements that present a symptom, also allowing one of the quick solutions to be applied to one or more elements of the ontology that presents problems. One of the more noteworthy advantages of the proposal is the possibility for the user to simulate the application of a quick fix in order to determine whether the result is favorable, before applying it permanently. However, after applying a quick solution there is the possibility of consulting/downloading/restoring the different earlier versions generated by OntologyFixer.

OntologyFixer also defines a graphical method to represent the quality of an ontology (Figure 8a), ensuring the user can get an overview of its quality at a glance. Additionally, the application supports the detection of errors and smell symptoms that will guide the application of quick fixes. Finally, the application of quick fixes can be rolled back if the ontology does not achieve the desired level of quality (Figure 9b).

The number of implemented OntologyFixer quick fixes is expected to be developed as future work in a short period of time.

Additionally, we believe that the tool could benefit from the use of multi-objective optimization algorithms to implement a semi-automatic ontology fixing scheme. OntologyFixer multi-objective optimization features will provide support to address the improvement of multiple ontology quality metrics simultaneously, by searching and identifying the optimal subset of quick fix actions to be applied to the ontology, and the optimal order of their application.

Although quality improvements of the ontology by the means of automatic and optimal fixing decisions reduce the effort required by the knowledge engineer, the automatic decisions might favor some quality attributes that do not correspond to the knowledge engineer's preferences. Therefore, the optimization process must consider the benefits and costs of the automatic decision, i.e., decrease the efforts of the knowledge engineer by an automatic decision that generally improves the ontology quality, or forward a diverse subset of optimal alternatives for the knowledge engineer to select, according to his/her preferences.

The set of actions to be considered by the optimization process are dependent on the detected pitfalls, e.g., annotation pitfalls that refer to the lack of information in the ontology can be fixed by identifying the classes of greatest relevance in the ontology, and by asking the user to add the comments or annotations that are pertinent to improve the understanding of the ontology and its elements. Reasoning pitfalls may include relations incorrectly defined as inverse, which can be solved by checking that they have a domain and range. If this were not the case, the tool could suggest possible options for domains and ranges or suggest removing them. The OntologyFixer optimization process will identify an optimal sequence of actions (quick fixes such as RM_INVOLVED_ELEMENTS and/or RM_SIMILAR_ELEMENTS) to improve the ontology and will present alternative sequences of actions (with the corresponding ontology quality attributes impact) to apply the one that best fits the knowledge engineer's preferences. The number of decisions to be forwarded to the knowledge engineer must also be considered as criteria to be minimized in the optimization process.

Author Contributions: Conceptualization, G.R.R.-M., J.R.M. and V.B.-F.; methodology, G.R.R.-M. and V.B.-F.; software, G.R.R.-M.; validation, I.Y.; formal analysis, I.Y. and V.B.-F.; investigation, G.R.R.-M., J.R.M., I.Y. and V.B.-F.; resources, G.R.R.-M., J.R.M. and V.B.-F.; data curation, G.R.R.-M. and V.B.-F.; writing—original draft preparation, G.R.R.-M., J.R.M. and V.B.-F.; writing—review and editing, J.R.M., I.Y. and V.B.-F.; visualization, G.R.R.-M., J.R.M., I.Y. and V.B.-F.; supervision, J.R.M. and V.B.-F.; project administration, J.R.M. and V.B.-F.; funding acquisition, J.R.M. and V.B.-F. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Spanish Ministry of Economy, Industry and Competitiveness (SMEIC), State Research Agency (SRA) and the European Regional Development Fund (ERDF) under the project Semantic Knowledge Integration for Content-Based Spam Filtering, grant number TIN2017-84658-C2-1-R". This research was funded by FCT—Fundação para a Ciência e a Tecnologia, I.P., grant numbers UIDB/04466/2020 and UIDP/04466/2020.

Acknowledgments: SING group thanks CITI (Centro de Investigación, Transferencia e Innovación) from University of Vigo for hosting its IT infrastructure.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. RDF Working Group RDF—Semantic Web Standards. Available online: https://www.w3.org/RDF/ (accessed on 26 March 2020).
2. RDF Working Group RDFS—Semantic Web Standards. Available online: https://www.w3.org/2001/sw/wiki/RDFS (accessed on 26 March 2020).
3. OWL Working Group OWL—Semantic Web Standards. Available online: https://www.w3.org/2001/sw/wiki/OWL (accessed on 26 March 2020).
4. Köhler, S.; Bauer, S.; Mungall, C.J.; Carletti, G.; Smith, C.L.; Schofield, P.; Gkoutos, G.V.; Robinson, P.N. Improving ontologies by automatic reasoning and evaluation of logical definitions. *BMC Bioinform.* **2011**, *12*, 418. [CrossRef] [PubMed]
5. Ali, N.; Hong, J.-E. Failure Detection and Prevention for Cyber-Physical Systems Using Ontology-Based Knowledge Base. *Computers* **2018**, *7*, 68. [CrossRef]
6. Munir, K.; Sheraz Anjum, M. The use of ontologies for effective knowledge modelling and information retrieval. *Appl. Comput. Inform.* **2018**, *14*, 116–126. [CrossRef]
7. Arch-int, N.; Arch-int, S. Semantic Ontology Mapping for Interoperability of Learning Resource Systems using a rule-based reasoning approach. *Expert Syst. Appl.* **2013**, *40*, 7428–7443. [CrossRef]
8. Zhang, J.; Zhao, W.; Xie, G.; Chen, H. Ontology- Based Knowledge Management System and Application. *Procedia Eng.* **2011**, *15*, 1021–1029. [CrossRef]
9. Uschold, M.; Gruninger, M. Ontologies: Principles, methods and applications. *Knowl. Eng. Rev.* **1996**, *11*, 93–136. [CrossRef]
10. Gruber, T.R. A translation approach to portable ontology specifications. *Knowl. Acquis.* **1993**, *5*, 199–220. [CrossRef]
11. Storey, V.C.; Chiang, R.; Chen, G.L. Ontology Creation: Extraction of Domain Knowledge from Web Documents. In Proceedings of the 2005 24th Conference on Conceptual Modelling, Klagenfurt, Austria, 24–28 October 2005; pp. 256–269.
12. Poveda-Villalón, M.; Gómez-Pérez, A.; Suárez-Figueroa, M.C. OOPS! (OntOlogy Pitfall Scanner!). *Int. J. Semant. Web Inf. Syst.* **2014**, *10*, 7–34. [CrossRef]
13. Poveda-Villalón, M. OOPS!—OntOlogy Pitfall Scanner!—Pitfall Catalogue. Available online: http://oops.linkeddata.es/catalogue.jsp (accessed on 26 March 2020).
14. Teymourlouie, M.; Zaeri, A.; Nematbakhsh, M.; Thimm, M.; Staab, S. Detecting hidden errors in an ontology using contextual knowledge. *Expert Syst. Appl.* **2018**, *95*, 312–323. [CrossRef]
15. Duque-Ramos, A.; Fernández-Breis, J.T.; Iniesta, M.; Dumontier, M.; Egaña Aranguren, M.; Schulz, S.; Aussenac-Gilles, N.; Stevens, R. Evaluation of the OQuaRE framework for ontology quality. *Expert Syst. Appl.* **2013**, *40*, 2696–2703. [CrossRef]
16. Bøegh, J. A New Standard for Quality Requirements. *IEEE Softw.* **2008**, *25*, 57–63. [CrossRef]
17. Bachir Bouiadjra, A.; Benslimane, S.-M. FOEval: Full ontology evaluation. In Proceedings of the 2011 7th International Conference on Natural Language Processing and Knowledge Engineering, Tokushima, Japan, 27–29 November 2011; IEEE: Piscataway, NJ, USA; pp. 464–468.

18. Tartir, S.; Arpinar, I.B. Ontology Evaluation and Ranking using OntoQA. In Proceedings of the International Conference on Semantic Computing (ICSC) 2007, Irvine, CA, USA, 17–19 September 2007; IEEE: Piscataway, NJ, USA; pp. 185–192.
19. Guarino, N.; Welty, C.A. An Overview of OntoClean. In *Handbook on Ontologies*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 201–220.
20. International Organization for Standardization Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—Guide to SQuaRE 2014. Available online: https://www.iso.org/standard/64764.html (accessed on 26 March 2020).
21. Brank, J.; Grobelnik, M.; Mladenić, D. A Survey of Ontology Evaluation Techniques. In Procroceedings of the 8th International Multi-Conference Information Society, Ljubljana, Slovenia, 17 October 2005; pp. 166–169.
22. Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D. When and Why Your Code Starts to Smell Bad. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; IEEE: Piscataway, NJ, USA; pp. 403–414.
23. Haldar, R.; Mukhopadhyay, D. Levenshtein Distance Technique in Dictionary Lookup Methods: An Improved Approach. *arXiv* **2011**, arXiv:1101.1232.
24. Poveda, M.; Delgado García, M.Á. OOPS!—OntOtology Pitfall Scanner! RESTFul Web Service 2013. Available online: http://oops.linkeddata.es/webservice.html (accessed on 26 March 2020).
25. Martin, R.C. *Agile Software Development, Principles, Patterns, and Practices*; Prentice Hall: Upper Saddle River, NJ, USA, 2003; ISBN 978-0135974445.
26. JSR-314 (JSF 2.0) Expert Group JavaServer Faces.org 2004. Available online: https://jcp.org/en/jsr/detail?id=314 (accessed on 26 March 2020).
27. GrayGrids Inc. Gentelella—Free Bootstrap Admin Template 2019. Available online: https://graygrids.com/templates/gentelella-free-bootstrap-admin-template/ (accessed on 26 March 2020).
28. Grove, R.F.; Ozkan, E. THE MVC-WEB DESIGN PATTERN. In Proceedings of the 7th International Conference on Web Information Systems and Technologies, SciTePress—Science and and Technology Publications, Setúbal (Portugal), Noordwijkerhout, The Netherlands, 6–9 May 2011; pp. 127–130.
29. Baeldung SRL The DAO Pattern in Java 2020. Available online: https://www.baeldung.com/java-dao-pattern (accessed on 26 March 2020).
30. Edwin, N.M. Software Frameworks, Architectural and Design Patterns. *J. Softw. Eng. Appl.* **2014**, *07*, 670–678. [CrossRef]
31. Horridge, M.; Bechhofer, S. The OWL API: A Java API for OWL Ontologies. *Semant. Web* **2011**, *2*, 11–21. [CrossRef]