FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	RDF Editor Plugin for the OpenPonk Tool
Student:	Vojtěch Doležal
Supervisor:	Ing. Jan Blizničenko
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

The goal of this thesis is the implementation of the RDF model editor as a plugin of the OpenPonk tool.
RDF models will be visualized and modified using diagrams.
Support for import and export in JSON-LD format is required.

1. Familiarize yourself with the RDF standard, the JSON-LD format and its use for RDF models
2. Familiarize yourself with the OpenPonk tool and its existing plugins
3. Review existing RDF editing tools
4. Familiarize yourself with relevant libraries/tools applicable for working with the JSON-LD format on the Pharo platform
5. Design an architecture, implement and test the solution
6. Demonstrate your solution on a simple case study
7. Document your solution

Electronically approved by Ing. Michal Valenta, Ph.D. on 6 February 2023 in Prague.

Bachelor's thesis

RDF Editor Plugin for the OpenPonk Tool

Vojtěch Doležal

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Supervisor: Ing. Jan Blizničenko

May 11, 2023

Composed and generated using LibreOffice Writer.

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Doležal Vojtěch. RDF Editor Plugin for the OpenPonk Tool. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Table of Contents

Acknowledgements

I would like to thank everyone for all manner of support during my studies and especially to Ing. Jan Blizničenko for proposing this topic and supervising the process of writing this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 11, 2023

Vojtěch Doležal

Abstract

This thesis discusses design and implementation of a plugin for the OpenPonk platform for work with linked data. The goal of this thesis was to create a software for straightforward visualization and editing of RDF data. Main parts of this thesis include import and export of this data, user interface for visualization and manipulation with the loaded data, and querying of this data. In the final part of the thesis, after the presentation of other functions, is the plugin demonstrated on a case study.

Keywords: data visualization, data management, linked data, OpenPonk, Pharo, Resource Description Framework, JSON-LD, SPARQL

Abstrakt

Tato práce se zabývá návrhem a implementací pluginu pro platformu OpenPonk pro práci s linkovanými daty. Cílem práce bylo vytvořit software pro snadnou vizualizaci a úpravu RDF dat. Hlavní části této práce zahrnují import a export dat, uživatelské rozhraní pro vizualizaci a manipulaci s načtenými daty, a dotazování nad těmito daty. V závěru práce, po prezentaci ostatních funkcí, je použití pluginu předvedeno na modelové studii.

Klíčová slova: vizualizace dat, správa dat, propojená data, OpenPonk, Pharo, Resource Description Framework, JSON-LD, SPARQL

Introduction

The concept of linked data, invented in the late 1990s, solved many issues connected to the process of publishing data on the internet. Using URLs as identifiers means not only that resources can be unambiguously referenced by other resources in datasets published by a different entity, it also means these identifiers may be dereferenced at any time to provide more information. Specifically RDF (Resource Description Framework) provides this data in terms of Graphs containing Subject-Predicate-Object triples. Due to predicates being resources as well, these datasets can also share predicates defined by a third party, provided a rigorous descriptions for their intended usage is also given, making it easier to process multiple datasets in a uniform way.

Soon after the introduction of RDF also came SPARQL, a language for querying these linked data graphs. SPARQL is now commonly supported and makes using RDF even easier for someone willing to learn it. However, the various formats of RDF make it still fairly difficult for a laic to meaningfully interact with an RDF data file, and a graphical visualization and editing tool would clearly be preferable to reading or editing the files directly in their text form.

Goals

The main goal of this thesis is to develop an OpenPonk plugin which would allow general user to import, visualize, edit and export linked data in format of JSON-LD files with knowledge of basic RDF concepts and little to no knowledge of the exact specifications of the JSON-LD standard. As an optional goal, this thesis should explore possibility of querying the imported data, for example using SPARQL.

The analysis section briefly explains main concepts and technologies associated with RDF and overviews similar products available on the market, comparing their features and flaws. Finally, a small selection of JSON-LD libraries is examined.

The practical section then uses knowledge gained in the previous section to design and implement a solution which would fit the criteria. This implementation is documented, tested, and demonstrated on a case study.

The goal is however not necessarily to create a brilliantly polished product that could be immediately deployed, as the OpenPonk platform is currently awaiting a large update changing the version of the underlying graphical library, and the resulting plugin will have to be appropriately prepared before a deployment regardless.

Chapter 1

Technologies

1.1 Pharo

"Pharo is a pure object-oriented programming language and a powerful environment, focused on simplicity and immediate feedback (think IDE and OS rolled into one)." [1]

Pharo is a modern dialect of Smalltalk, one of the original object oriented programming languages. It uses very simple syntax with only six reserved keywords, but which was tailored for ergonomics when working with objects. Furthermore, there are no access modifiers, with methods that are available publicly and instance variables that are protected (only available to an instance, even if it is an instance of a child class), which naturally leads to simple to understand classes and heavy use of composition. [2]

Lastly, one of its other attractive features is also the fact there are no black boxes: everything, including elements of the environment itself, can be easily inspected and have its source code modified. That leads to very responsive and hands-on development experience.

1.2 OpenPonk

"OpenPonk is a metamodeling platform and a modeling workbench implemented in the dynamic environment Pharo aimed at supporting activities surrounding software and business engineering such as modeling, execution, simulation, source code generation, etc." [3].

OpenPonk is currently mainly developed by persons associated with the Faculty of Information Technology at the Czech Technical University in Prague. It offers plugins for creation of various types of models including but not limited to ontology domains, finite state machines and BPMN process models. Due to already being used by plugins that are similar in nature, it features components and functions that make it suitable to be the base of implementation part of this thesis.

Plugins for OpenPonk are expected to follow architecture known as MVC (Model-View-Controller). That means the class for representing the state of a modelled object (the Model) is separate from class that represents the modelled object in the user interface (the View). Due to that, operations with modelled objects can be done without a view even existing. Model and View are then connected by the Controller, on which they are independent. [4]

In OpenPonk each project consists of one or more models, which may or may not be of the same kind. These models, and elements they may contain, are provided by the OpenPonk plugins. These models and elements both commonly derive from the `OPModelObject` class as it provides UUID generation and methods for work with subelements and owner element, as well as announcer, which are usually useful. Simularly, controllers for models and elements usually derive from classes `OPDiagramController` and `OPElementController`

respectively. Special case are controllers for edge elements, in other words elements connecting other elements, which usually derive from `OPDirectionalRelationshipController`. Finally, the view classes that are responsible for displaying the modelled object on the canvas derive from `OPDIDiagramElement`.

1.3 Resource Description Framework (RDF)

"The Resource Description Framework (RDF) is a framework for representing information in the Web. […] The abstract syntax has two key data structures: RDF graphs are sets of subject-predicate-object triples, where the elements may be IRIs, blank nodes, or datatyped literals. […] RDF datasets are used to organize collections of RDF graphs, and comprise a default graph and zero or more named graphs." [5]

In other words RDF describes given domain using subject-predicate-object triples, or in graph terms with triples node—directed edge—node. Any node is either an IRI (Internationalized Resource Identifier), a literal (such as a number or a string), or a blank node, which can be used to describe an object in one or multiple statements without specifying its global identifier (for example when expressing the sentence "Alice knows a person who was born on the 1st of April 1939", "a person" might be referred to using a blank node). The edge, describing

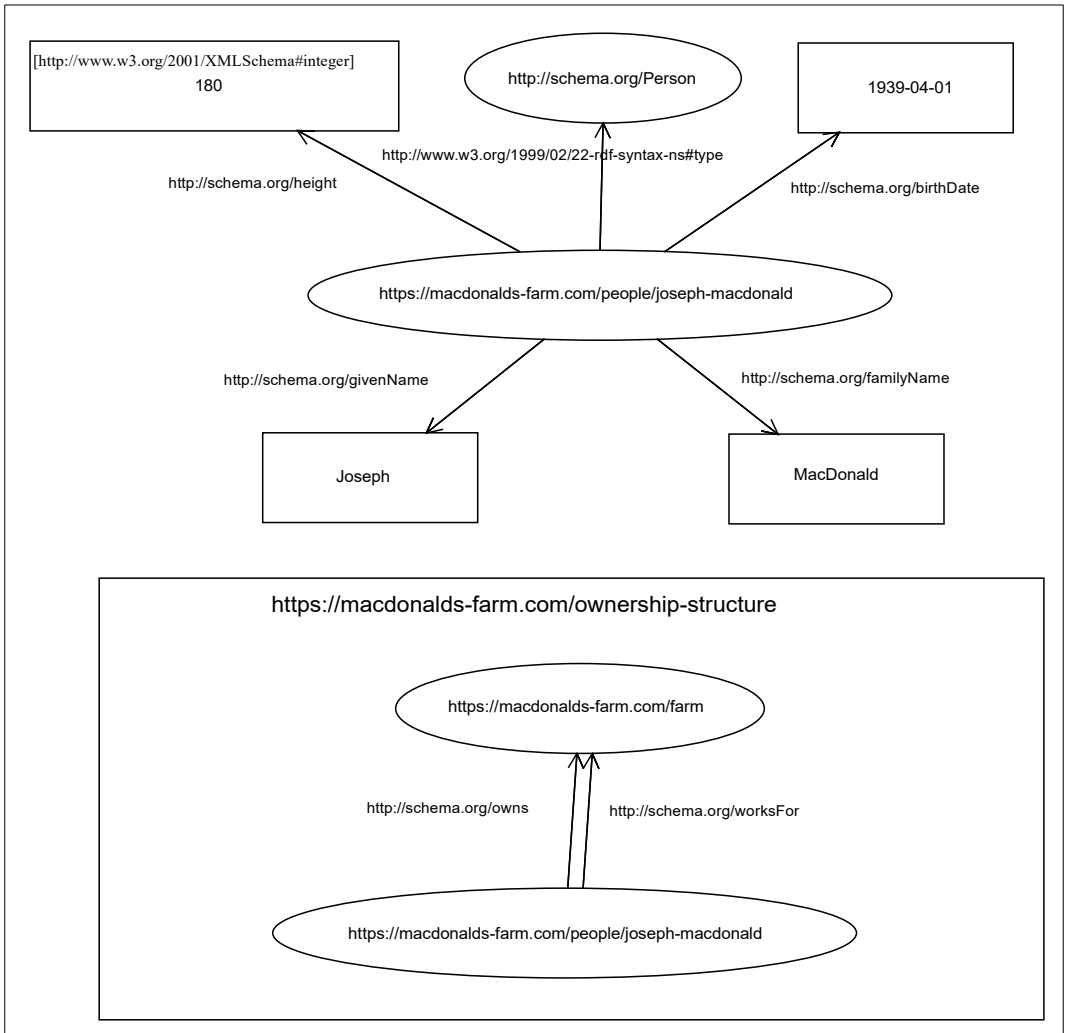

Figure 1: Visualization of the example domain

the predicate, must be an IRI. Since any IRI is either a URL (Uniform Resource Locator), or URN (Uniform Resource Name), not only is it much simpler to keep identifiers of nodes and predicates globally unique, they can also be dereferenced, usually providing more information about the resource.

"This Web of Data […], also referred to as Semantic Web […], presents a revolutionary opportunity for deriving insight and value from data. By enabling seamless connections between data sets, we can transform the way drugs are discovered, create rich pathways through diverse learning resources, spot previously unseen factors in road traffic accidents, and scrutinise more effectively the operation of our democratic systems." [6]

Common storage formats include RDF/XML (the original RDF format), various supersets of N-Triples (such as N-Quads, Turtle, Notation3 and TriG) and JSON-LD. These formats have various differences in capabilities and intended purpose. Main differences include whether a file of the format can contain multiple graphs (most formats do not support this) and whether there are ways to shorten common prefixes of IRIs (most formats do support this).

Let us introduce an example RDF domain. The data in this domain will be related to certain Joseph MacDonald and his farm (both fictitious). This domain will include two graphs, 'https://macdonalds-farm.com/ownership-structure' and the default graph. The default graph contains the date of Joseph's birth, his height rounded to centimetres, his given and family name and the fact he is a person. The named graph, on the other hand, contains the fact he works for MacDonalds farm, which he also owns. Visualization of this domain can be seen on figure 1.

1.4 Relevant RDF Formats

1.4.1 JSON-LD

"JSON-LD is a lightweight syntax to serialize Linked Data in JSON [...]. Its design allows existing JSON to be interpreted as Linked Data with minimal changes. JSON-LD is primarily intended to be a way to use Linked Data in Web-based programming environments, to build interoperable Web services, and to store Linked Data in JSON-based storage engines. Since JSON-LD is 100% compatible with JSON, the large number of JSON parsers and libraries available today can be reused." [7]

JSON-LD is an RDF format based on JSON. One of its biggest features is that communication that uses standard JSON data can simply have JSON-LD context data inserted at a special keys that start with '@' to make it JSON-LD. That means legacy software or devices can then ignore the additional JSON-LD data and process the rest in the same way they used to process the old data, but up to date consumers can also read the additional context from the additional objects.

Likely the most important of these special properties are '@id', '@type', '@context', '@base' and '@vocab'. Properties '@id' and '@type' are used to denote identifier and type of an element, respectively. '@context' denotes block where definitions take place, '@base' specifies the base domain to be used for identifiers and '@vocab' specifies the domain to be used for predicates. [7]

Other important properties include '@graph' which denotes named graph content (see snippet 1), '@list' for declaring content is ordered and '@import' which allows loading of an external context. [7]

```
{
  "@context": {
    "@base": "https://macdonalds-farm.com/",
    "@vocab": "https://schema.org/"
  },
  "@graph": [
    {
      "@id": "ownership-structure",
      "@graph": [
        {
          "@id": "people/joseph-macdonald",
          "owns": {"@id": "farm"},
          "worksFor": {"@id": "farm"}
        }
      ]
    },
    {
      "@id": "people/joseph-macdonald",
      "@type": "http://schema.org/Person",
      "givenName": "Joseph",
      "familyName": "MacDonald",
      "birthDate": "1939-04-01",
      "height": 180
    }
  ]
}
```

Snippet 1: Example domain in JSON-LD

1.4.2 N-Quads

"N-quads statements are a sequence of RDF terms representing the subject, predicate, object and graph label of an RDF Triple and the graph it is part of in a dataset." [8]

The N-Quads standard was invented as a superset to a standard called N-Triples. Whereas N-Triples statements are just Subject-Predicate-Object triples, N-Quads allow optional fourth term, which then describes the graph the statement belongs to. This is important, because that means N-Quads have the same ability of expression as JSON-LD while being much simpler. Example domain in N-Quads may be seen in snippet 2 (each term of each statement is only given its own line for readability).

```
1  <https://macdonalds-farm.com/people/joseph-macdonald>
       <http://schema.org/owns>
       <https://macdonalds-farm.com/farm>
       <https://macdonalds-farm.com/ownership-structure> .
2  <https://macdonalds-farm.com/people/joseph-macdonald>
       <http://schema.org/worksFor>
       <https://macdonalds-farm.com/farm>
       <https://macdonalds-farm.com/ownership-structure> .
3  <https://macdonalds-farm.com/people/joseph-macdonald>
       <http://schema.org/birthDate>
       "1939-04-01" .
4  <https://macdonalds-farm.com/people/joseph-macdonald>
       <http://schema.org/familyName>
       "MacDonald" .
5  <https://macdonalds-farm.com/people/joseph-macdonald>
       <http://schema.org/givenName>
       "Joseph" .
6  <https://macdonalds-farm.com/people/joseph-macdonald>
       <http://schema.org/height>
       "180"^^<http://www.w3.org/2001/XMLSchema#integer> .
7  <https://macdonalds-farm.com/people/joseph-macdonald>
       <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
       <http://schema.org/Person> .
```

Snippet 2: Example domain as N-Quads

N-Quads are not the only superset to N-Triples, however the other superset are not entirely important for this thesis. Briefly and incompletely, Turtle introduces shorthands and cascades, Notation3 expands that with logical operations, whereas TriG adds syntax for encoding multiple graphs.

1.5 SPARQL

"SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports aggregation, subqueries, negation, creating values by expressions, extensible value testing, and constraining queries by source RDF graph. The results of SPARQL queries can be result sets or RDF graphs." [9]

SPARQL is a querying language not too dissimilar to SQL, but clearly inspired by Turtle syntax. Just like in SQL, typical SPARQL query has SELECT and WHERE clause, optionally followed by clauses such as GROUP BY, HAVING, ORDER BY. The main difference lies in the WHERE clause, which is a body typically containing statements inspired by Turtle, but which may also contain statements for filtering data and binding calculated data to output columns.

As an example, suppose the example domain contains information about employee contracts at the farm and that every contract is tied to specific department and each department has a number and a name. Assume we want to find list of departments "tags" consisting of department number and name, and number of employees ordered by this number, but only if the department number is less than 4. Query retrieving this information might look like the query in snippet 3. Result of this query might look like the data in table 1.

```
1  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2  PREFIX sch: <https://schema.org/>
3  PREFIX macfarm: <https://macdonalds-farm.com/>
4
5  SELECT ?dept_tag (COUNT(?person) AS ?count)
6  WHERE {
7      macfarm:farm sch:department ?department.
8      ?department sch:employee ?person;
9                  sch:numberedPosition ?dept_number;
10                 sch:name ?dept_name.
11     FILTER(?dept_number < 4)
12     BIND(CONCAT(?dept_number, " - ", ?dept_name) as ?dept_tag)
13 } GROUP BY ?dept_tag
14 ORDER BY DESC(?count) ?dept_tag
```

Snippet 3: Example SPARQL query

?dept_tag	?count
"1 - Cow house"	3
"3 - Slaughterhouse"	3
"2 - Maintenance"	2

Table 1: Possible result of the example SPARQL query

Chapter 2

Analysis of similar products

This chapter briefly runs through RDF related products that might be considered a competition on the market. It does not includes analysis of RDF servers such as Amazon Neptune[1] or Apache Jena[2] which are, for the most part, products with a different target audience.

2.1 Neologism 2.0

"Neologism 2.0 is an open-source tool for quick vocabulary creation through domain experts. Its guided vocabulary creation and its collaborative graph editor enable the quick creation of proper vocabularies, even for non-experts, and dramatically reduces the time and effort to draft vocabularies collaboratively. An RDF export allows quick bootstrapping of any other Semantic Web tool." [10]

Neologism is a web tool focused on creation of vocabularies. Due to this fact, it unfortunately does not support editing of general RDF files. Additionally it has no commits in the last 5 months and the live demo linked in the readme file seems to have been taken offline, making the project appear as no longer developed.

2.2 The NeOn Toolkit

"The NeOn toolkit is a state-of-the-art, open source multi-platform ontology engineering environment, which provides comprehensive support for the ontology engineering life-cycle. The toolkit is based on the Eclipse platform, a leading development environment, and provides an extensive set of plug-ins (currently 45 plug-ins are available) covering a variety of ontology engineering activities" [11]

The NeOn Toolkit is one of the RDF tools mentioned by Heath and Bizer [6], unfortunately it does not appear to currently be maintained, with the last version available on the website being from December 2011. This version seems to have difficulties running on modern machines, and even features the notorious Log4J library (unknown whether the exact version is vulnerable in the same way the modern version was).

2.3 Protégé (Desktop)

"Protégé Desktop is a feature rich ontology editing environment with full support for the OWL 2 Web Ontology Language, and direct in-memory connections to description logic reasoners like HermiT and Pellet." [12]

Protégé is a project developed mainly for modelling ontologies. It offers a way of modelling subject-predicate-object, as well as SPARQL querying. However when using the intuitive method of description, there does not seem to be a solid

[1] https://aws.amazon.com/neptune/

[2] https://jena.apache.org/

way to properly visualize the data, and on the other hand when saving the data in a way so that it would be displayable by the class viewer, it is hard to query.

Protégé allows saving into some of RDF interchange formats, however it is not able to load and edit these RDF formats in general, as files that are being loaded must follow its preferred structure (it seems to have a problem especially with RDF files that don't use classes).

2.4 TopBraid Composer

"In use by thousands of commercial customers, Composer offers robust and comprehensive support for building and testing configurations of rich knowledge graphs." [13]

TopBraid Composer claims to have many features for working with both models and data. Some of these include SHACL and SPARQL autogeneration, and refactoring of both model and data.

It claims to have support for import and export of RDF files and SPARQL querying. However, similarly to Protégé, has issues interpreting general RDF files without a class structure. It should be mentioned that it is a commercial closed-source tool as well.

2.5 WebProtégé

"WebProtégé is an ontology development environment for the Web that makes it easy to create, upload, modify, and share ontologies for collaborative viewing and editing." [14]

WebProtégé is the continuation of the Protégé project. The interface runs in the browser and although it does require login, the registration and access to the instance provided by Stanford University seems to be free and open to the public as well.

It is fair bit more polished and intuitive than the desktop version, mainly because the desktop version seems to be mostly abandoned at the moment. Just like its predecessor, it too features a way to make queries, however instead of SPARQL processor, WebProtégé provides a custom graphical query builder. While it might be bit more intuitive for a casual user, it is definitely less expressive than the SPARQL processor provided by the desktop version. It also means the queries cannot be copied and saved elsewhere and are not portable, which might be a great annoyance to some.

It also features options for collaboration (four levels of privileges which can be granted to other users: manage, edit, comment, view), and just like its predecessor allows import from and export to RDF interchange formats, while still not being general RDF editor.

2.6 Conclusion of analysis of similar products

In conclusion, none of the examined products seem to do exactly what this thesis aims to do. Any support for RDF exchange formats in graphical tools seems to be more for export and later import, primarily of ontological models, rather than for import of a generic RDF file, visualization and editing, and later export.

Chapter 3

Analysis of JSON-LD libraries

3.1 Analysis of libraries for Pharo

According to a brief research, there seem to be no existing JSON-LD library for Pharo. While plain JSON libraries (such as STONJSON and NeoJSON) are available and would certainly help with the implementation, after taking closer look at the JSON-LD specification and libraries for other languages, I determined it would not be possible for me to implement JSON-LD library from scratch in satisfying manner in such a short time.

3.2 Analysis of libraries for other languages

According to a list at JSON-LD.org [15], there are many RDF libraries with JSON-LD support for various programming languages, such as JSON-goLD[3] for Go, dotNetRDF[4] for C# and Titanium JSON-LD[5] for Java. However, Pharo does not offer a simple way interoperate with these languages, other than including pre-compiled binaries and running them from the command line.

The simplest language to run with RDF libraries seems to be Python, because it can be executed through the command line without compilation. Main libraries for work with JSON-LD in Python seem to be PyLD[6] and RDFLib[7] for Python. From these two, I chose RDFLib, because it has support for wide array of RDF formats and also comes with an engine for processing SPARQL queries.

3.3 Python-Pharo interoperability

As was stated in previous paragraph, interoperability between Python and Pharo through command line works to a reasonable degree. The main issue with it lies in behaviour on Windows, because Windows still uses code pages and therefore sending any Unicode characters through it is unreliable. This is not an issue for sending paths of data or data itself, as it can be encoded to not feature Unicode characters, but it would be an issue if user has Python executable installed in a location that contains Unicode characters (and does not have this location added in the search path environment variable).

[3] https://github.com/piprate/json-gold
[4] https://github.com/dotnetrdf/dotnetrdf
[5] https://github.com/filip26/titanium-json-ld
[6] https://github.com/digitalbazaar/pyld
[7] https://github.com/RDFLib/rdflib

Chapter 4

Design & Implementation

This chapter explains details about the design and implementation of the thesis.

4.1 RDFLib and intermediary data format

As was hinted at in the analysis section, implementing a custom JSON-LD library turned out to be harder than expected due to nuances in the standard specifications, and pursuing this route would likely be met with failure. As a substitute, Python library RDFLib is being called through the command line, converting the RDF data from various formats into a simple intermediary format based in JSON but similar to the N-Quads format. There are many advantages to this approach, including the fact RDFLib is well tested and receives frequent updates.

This intermediary format is simply just an array of arrays of simple JSON objects. When conditions are met, it may be interpreted as an array of N-Quads statements, with each object representing one term of the statement. It may, however, also carry the result of SPARQL query, in which case there may be any number of objects in the inner arrays. These JSON objects contain information about the type and values of the given term. The 'type' being either 'BLANK_NODE_LABEL', 'IRIREF', 'LITERAL' or 'NONE'. Terms of the type 'LITERAL' may also have optional 'language' and 'datatype' properties, having the same function as in N-Quads (example in snippet 4).

This simplicity allows to outsource the bulk of the parsing to a generic JSON library, output of operation of which can be loaded with just a few lines of code.

```
[[
    {
        "type": "BLANK_NODE_LABEL",
        "value": "b0"
    },
    {
        "type": "IRIREF",
        "value": "https://schema.org/name"
    },
    {
        "type": "LITERAL",
        "value": "Winter",
        "language": "en",
        "datatype": ""
    },
    {
        "type": "NONE"
    }
]]
```

Snippet 4: Example of intermediary RDF format

Naturally the inverse of this process, encoding model data into this intermediary format, is just as simple.

4.2 Class structure

The project has similar structure to other OpenPonk plugins, most significantly to the Finite State Machine plugin, which it is based on. UML Class diagram of the most important classes of the project can be seen on figure 2.

4.2.1 OpenPonk-RDF package

The code of the project is segmented into multiple so called packages. As the main class diagram suggests, the package OpenPonk-RDF contains the code

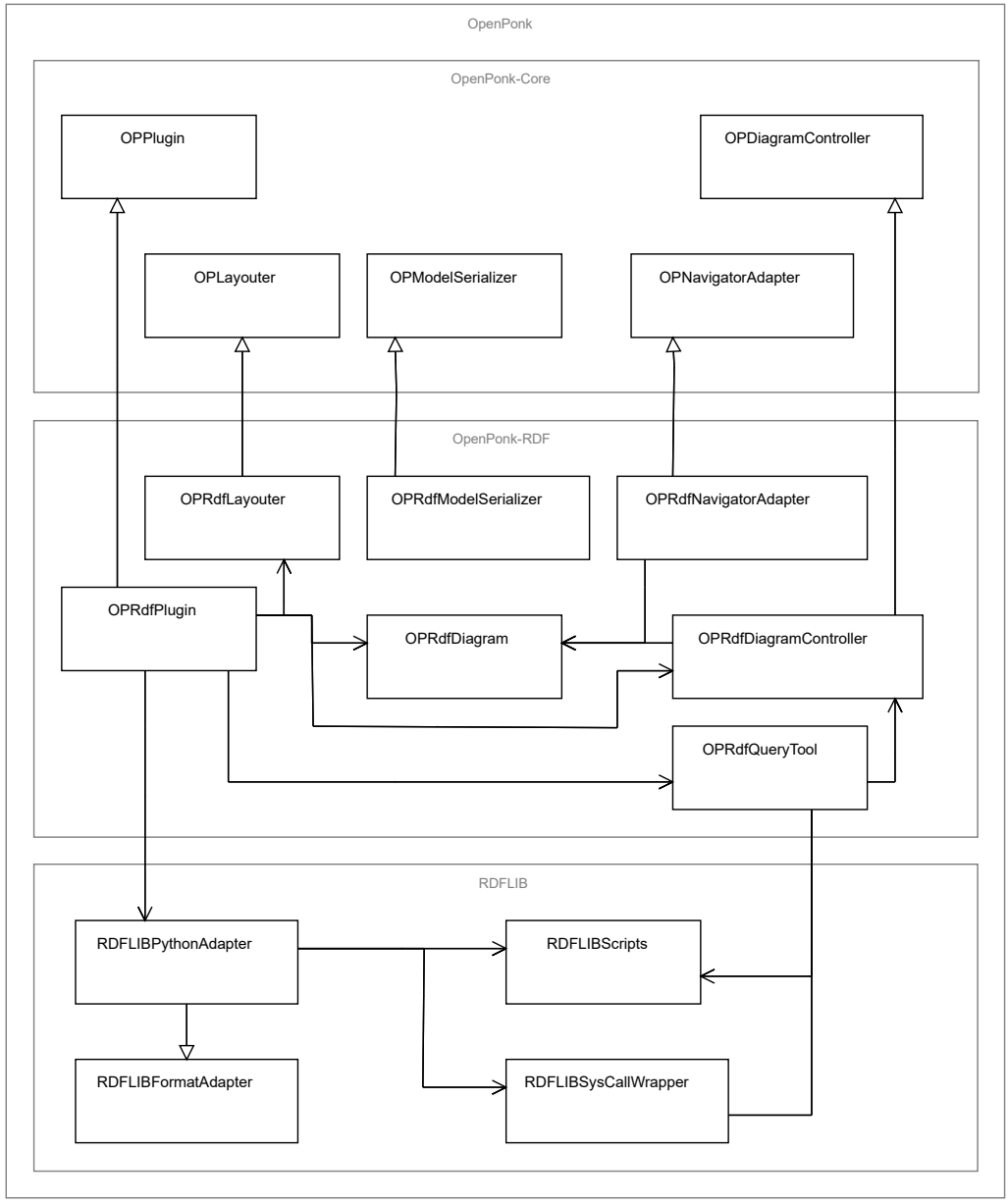

Figure 2: Structure of main classes of the project

connected with the user interface. It is further segmented into multiple "tags", a form of subpackaging.

4.2.1.1 Announcements tag

This tag contains two classes, namely `OPRdfDiagramQueriesModified` and `OPRdfDiagramControllersCountChangedAnnouncement`. These two classes represent custom Announcements that get sent when queries of a model and model controller count gets modified respectively. This is used by the Query Tool, because due to listening for these Announcements it can show up to date list of diagrams and queries even when a diagram was closed since opening, or a query was added or modified in another Query Tool window.

4.2.1.2 Controllers tag

This tag contains controllers for the models. Their main function is, per the MVC architecture, to create the View for the Model, in these cases to create Shapes for the models. Additionally the controllers are also responsible for controls for changing model properties when the object is selected (on the right hand side sidebar).

One exception to this is the `OPRdfDiagramController`, which also contains methods for importing and exporting data in the intermediary format.

4.2.1.3 Examples tag

The examples tag commonly contains classes with examples in Pharo. In this case there is only one example class named `OPRdfExamples`, which contains static methods `exampleEmpty` and `exampleMacDonaldsFarm`. When run, the first of these shows a new empty model, whereas the second shows the example domain. The class also has static `menuCommandOn:` method, which creates menu items from which these examples can be quickly summoned to help the user better understand the capabilities of the plugin.

4.2.1.4 Help tag

This tag contains `OPRdfHelp`, which is a help window the user can open through *Diagram* > *Help*. It contains basic information about the plugin, RDF, SPARQL and the Query Tool.

4.2.1.5 Models tag

This tag contains classes representing the model and the elements, which includes graphs, predicates, literals and nodes.

Since models and graphs may both contain other elements, classes `OPRdfDiagram` and `OPRdfGraph` have a common ancestor, `OPRdfContainer`, which provides methods for getting contained elements by type, as well as method for getting a dictionary form of contained items.

Nodes (`OPRdfNode`) and predicates (`OPRdfPredicate`) are both mainly defined by their IRI, but instances of `OPRdfPredicate` also references to the elements they connect, since they represent a directed edge.

Lastly, instances of `OPRdfLiteral` contain three strings, content, datatype and language, just as in the RDF model.

4.2.1.6 Plugin tag

The Plugin tag contains three classes, the `OPRdfPlugin`, which is a representation of the plugin itself, `OPRdfLayouter` and `OPRdfNavigatorAdapter`.

`OPRdfPlugin` extends `OPPlugin`, and as was stated is the representation of the OpenPonk RDF plugin itself. It includes information about which classes should be used for layouting, as a diagram model, as a model serializer, as well menu items OpenPonk should display.

`OPRdfLayouter` extends the `OPLayouter` class, and is responsible for applying layouts on the elements of diagrams, meaning for organizing them in a specific way. This organization is done automatically when importing new RDF data, but can also be triggered manually by the user through the user interface (figure 3). From there, the user can even choose between several organization styles. Unlike the `OPLayouter`, the `OPRdfLayouter` can organize nested structures. Most of the work is still done by the Roassal display library, exactly as with `OPLayouter`, but this layouter is first run for all graphs in the diagram, and only then on the top level elements, with the elements contained within graphs being moved to their new location afterwards.

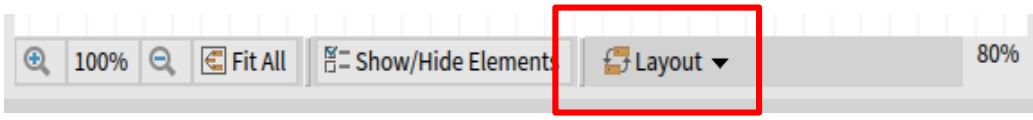

Figure 3: Location of the menu with layouter commands

`OPRdfNavigatorAdapter` extends `OPNavigatorAdapter` and provides additional information about how elements should be shown in the view, such as how should they be displayed in the hierarchic view.

4.2.1.7 QueryTool tag

This tag contains classes connected to the query tool, which can be open through *Diagram* > *Query,* and allows user to execute SPARQL queries. The queries themselves may also be saved in the given diagram for a later recall. When a query is ran, the result is shown in `OPRdfQueryResultTable`, and if the data could be understood as an RDF dataset, options for opening in a diagram are shown.

4.2.1.8 Serialization tag

This tag contains classes `OPRdfModelSerializer` and `OPRdfGraphML`. These classes handle proper serialization and deserialization of the diagram when saving or loading the project. Saved are not only the values, but also positions of the elements.

4.2.1.9 Shapes tag

This tag contains views for graphs, nodes and literals. These classes only extend shapes created earlier with minor changes to colours and shapes (nodes are displayed as ellipsoids), with `OPRdfGraphShape` extending the default `OPBoundaryFigure`, and `OPRdfLiteralShape` and `OPRdfNodeShape` extending `OPUmlClassifierShape` from the UML Class Diagram plugin.

4.2.2 RDFLIB package

This package contains classes wrapping the Python library RDFLib. The `RDFLIBScripts` class provides locations of Python scripts, which are also created by it. Subclasses of `RDFLIBPythonAdapter` then use `RDFLIBSysCallWrapper` to call the conversion script, from or to a specific RDF format.

4.3 Testing

Pharo offers multiple ways to run tests. The System Browser (*Browse > System Browser*) offers button for running tests when a class derives from the `TestCase` class. This button runs every method name of which begins with "test" when pressed, and subsequently it is coloured depending on the result of the tests. Similar buttons are also available for each of the individual test methods.

Another way to run tests in Pharo is through the DrTests test runner (currently under *Browse > Dr Test (Preview)*). The expected test structure is almost exactly the same as with the basic test runner, but it offers advanced options such as line coverage, which basic runner does not offer. It even supports running test suites split into multiple packages.

4.3.1 GitLab CI for Pharo

Continuous Integration can be achieved fairly easily in Pharo, and other dialects of Smalltalk, due to SmalltalkCI[8]. To use it, repository only needs to load the SmalltalkCI image (through YAML file such as in snippet 5) and have test configuration in a .smalltalk.ston file (snippet 6) located in the root directory of the repository.

```
1  image: hpiswa/smalltalkci
2
3  Pharo6410:
4    before_script:
5      # Install python3 and pip
6      - apt-get update -qq && apt-get install -y -qq python3
7      #- python -m ensurepip -upgrade
8      - apt-get update -qq && apt-get install -y -qq python3-pip
9      # Alias python to python3
10     #- apt-get update -qq
                 && apt-get install -y -qq python-is-python3
11     - update-alternatives -install
             /usr/local/bin/python python "$(which python3)" 3
12     # Install rdflib through pip
13     - pip3 install rdflib
14   script: smalltalkci -s "Pharo64-10"
```

Snippet 5: YAML configuration for GitLab CI

[8] https://github.com/hpi-swa/smalltalkCI

```
SmalltalkCISpec {
        #loading : [
                SCIMetacelloLoadSpec {
                        #baseline : 'RdfEditor',
                        #directory : 'repository',
                        #platforms : [ #pharo ]
                }
        ],
        #preLoading : '.github/scripts/preLoad.st',
        #postLoading : '.github/scripts/postLoad.st',
        #testing : {
                #include : {
                        #packages : [ 'OpenPonk*', 'RDFLIB*' ]
                },
                #coverage : {
                        #packages : [ 'OpenPonk-RDF', 'RDFLIB' ]
                }
        }
}
```

Snippet 6: SmalltalkCI .smalltalk.ston configuration file

4.3.2 Content of tests

Both of the main packages, OpenPonk-RDF and RDFLIB, have separate packages for tests, OpenPonk-RDF-Tests and RDFLIB-Tests.

Tests for OpenPonk-RDF mainly focus on operation of controllers and models such as property editing fields, intermediary format import and export, and model serialization and deserialization. This is mainly because these are the most important components, but also because significant changes for other parts are expected in near future.

For RDFLIB, the tests mainly test whether Python code can be executed, RDFLib package is present and scripts work approximately as expected. The difficulty with creating more tests for conversions between RDF formats is that properly testing the outputs are always correct would require the tests to already understand all the tested formats, which is the part that is outsourced to RDFLib. In almost all RDF formats the order of statements should not matter, and many formats furthermore allow for defining shorthands, making possibilities of how a valid output might look potentially endless. For this reason, only conversions to the intermediary format is tested, since while the order of statements can still vary, there is very finite number of possible valid outputs.

Unfortunately, likely due to behaviour changes between RDFLib versions, RDFLIB tests currently only appear to work locally and fail in the GitLab CI.

4.4 Case study

To demonstrate the functionality, let us step through a possible workflow of user of this plugin. Main tasks of this workflow include import of a JSON-LD file, manual correction of an error in the imported data, execution of a SPARQL query and export of the output of the query into an N-Quads file.

4.4.1 Import from an RDF file

If we assume input file is located in a known location, import of a file is done simply through the *Diagram* > *Import* menu, in case of JSON-LD through *Diagram* > *Import* > *JSON-LD*. Assume input data from Snippet 7.

```
{
  "@context": {
    "gn": "http://schema.org/givenName",
    "fn": "http://schema.org/familyName"
  },
  "@graph":[
    {
      "@id":
        "http://macdonalds-farm.com/people/joseph-macdonald",
      "gn": "Joseph",
      "fn": "MacDonald"
    },
    {
      "@id":
        "http://macdonalds-farm.com/people/jamie-macdonald",
      "gn": "Joseph",
      "fn": "MacDonald"
    },
    {
      "@id":
        "http://macdonalds-farm.com/people/jonah-macdonald",
      "gn": "Jonah",
      "fn": "MacDonald"
    }
  ]
}
```

Snippet 7: Case study: Input JSON-LD data

4.4.2 Making manual changes in imported data

To correct an error, we simply navigate to the erroneous part, or in general to the part we wish to change. Removal of elements can be done through context menu when using right mouse button on the given element (figure 4). Addition of elements can be done through the left hand side sidebar (figure 5 and figure 6). Properties of the selected element are shown on the right hand side sidebar (figure 7).

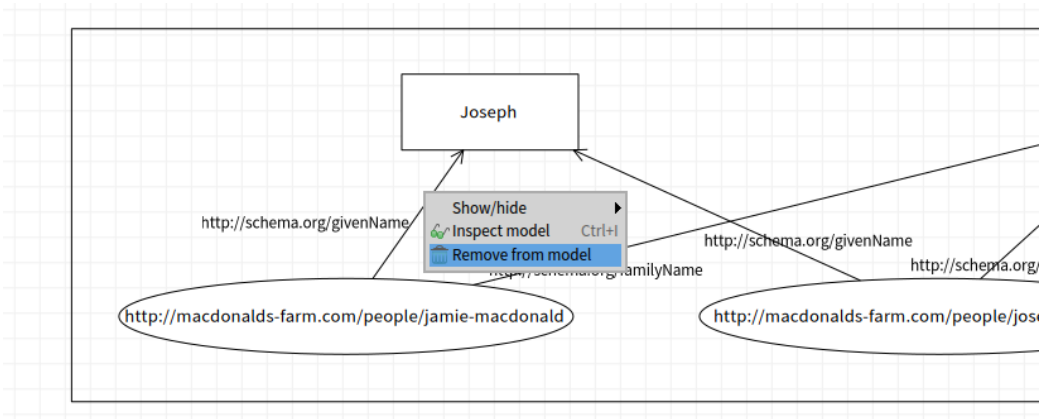

Figure 4: Element context menu showing how to delete an element

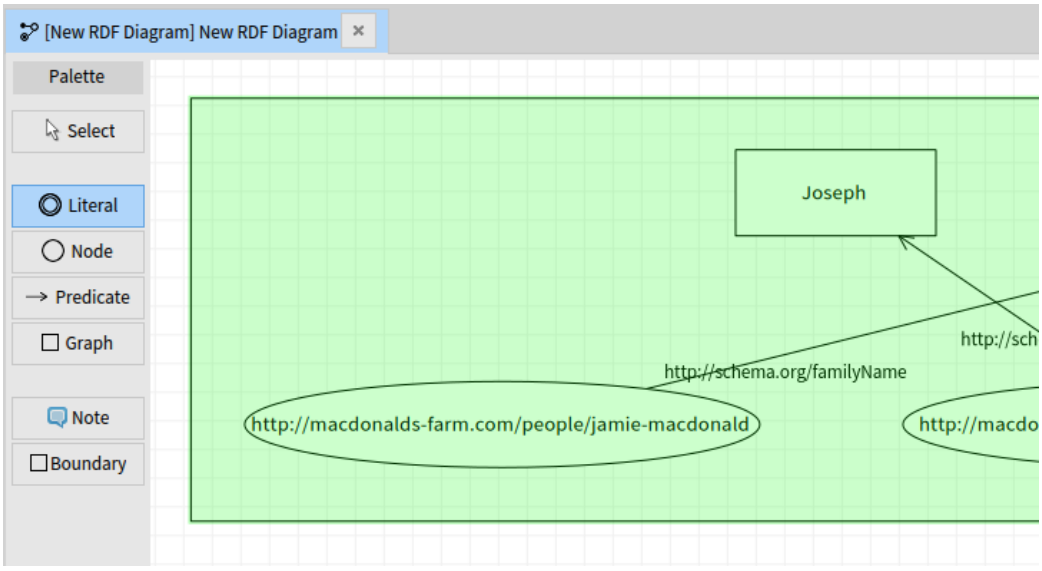

Figure 5: Insertion of new Literal into a graph

This correction may be documented by leaving a Note or creating a comment on the element (presence of which is indicated by an icon). Note that Notes or comments are not treated as RDF data. That means they cannot be queried through SPARQL and are not exported when exporting the data into RDF formats.

When we are happy with the adjustments, we save the project (*Project* > *Save Project*). Unlike exporting to an RDF format, saving a project does preserve Notes and comments.

4.4.3 Running a SPARQL query

To run a SPARQL query, we open the Query Tool using *Diagram* > *Query*. If we wish to query a different diagram from the one that is currently selected, we can select the correct one from the first dropdown list.

On the bottom part of the window is a query area. The default content of the query area is an "identity query", which returns all RDF data in the diagram.

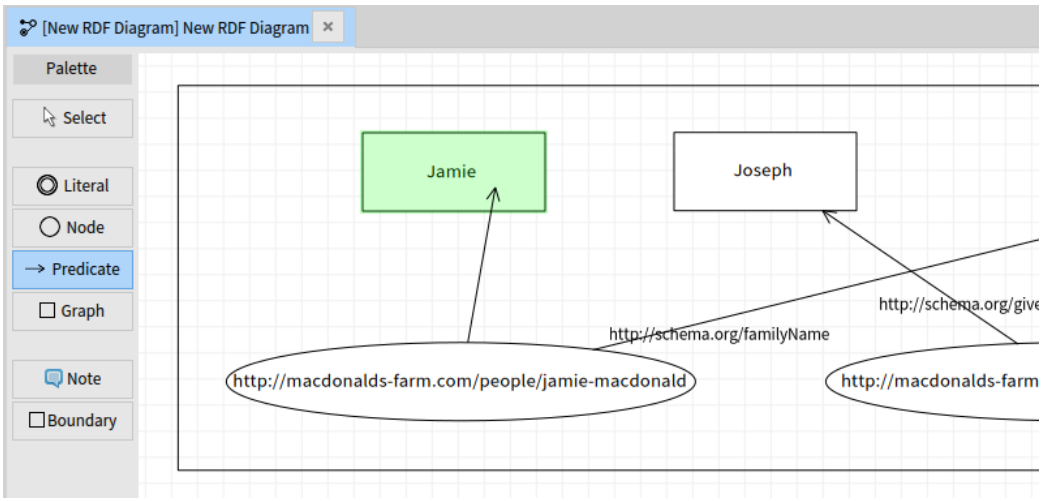

Figure 6: Insertion of new Predicate into a graph

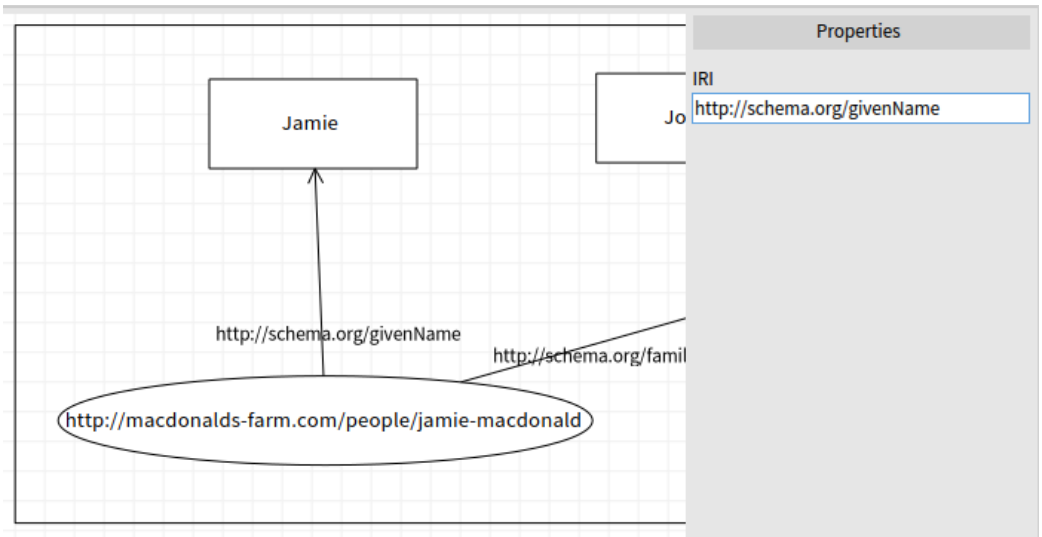

Figure 7: Right hand side sidebar for editing of properties

Content of the query area can be executed even if not saved, however to save content of the area, we enter a name into the second dropdown list and confirm. That can also be used if we want to save modified query under a new name. Saving without query name selected is not possible (unsaved changes indicator will not disappear).

If we assume we want to get aggregate data about counts of given names, the query might look same as on the figure 8.

After query is executed by pressing the *Run* button, window with results will appear. If resulting data could be interpreted as an RDF dataset, it will be indicated by buttons for opening in the original or a new project being enabled, as seen on figure 9.

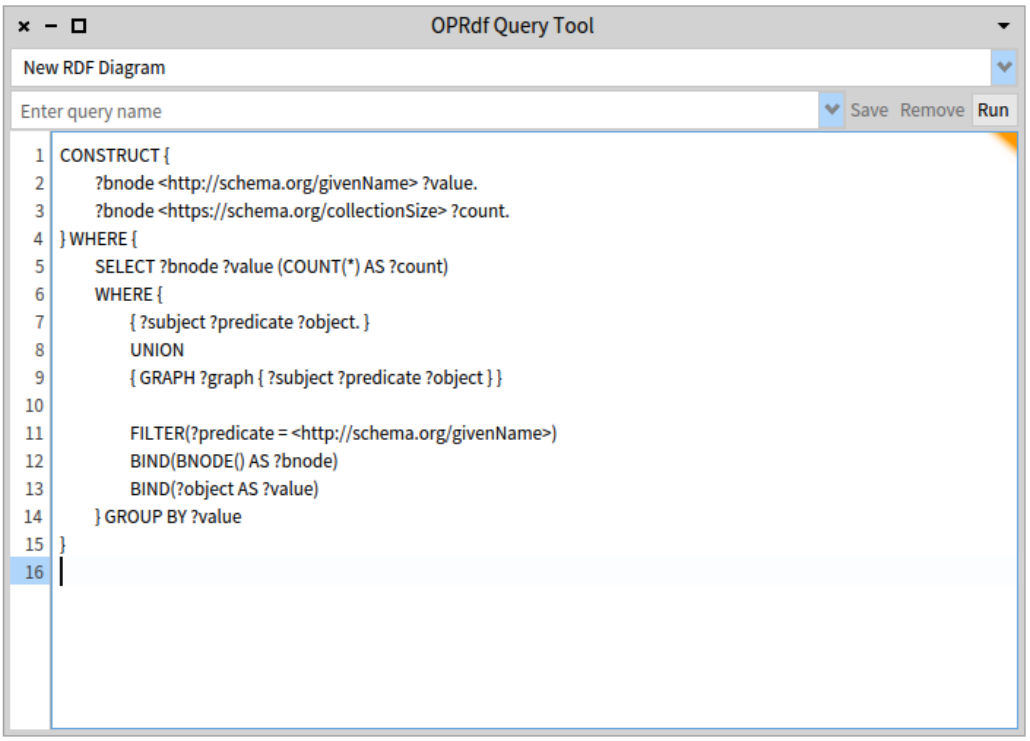

Figure 8: Query Tool window

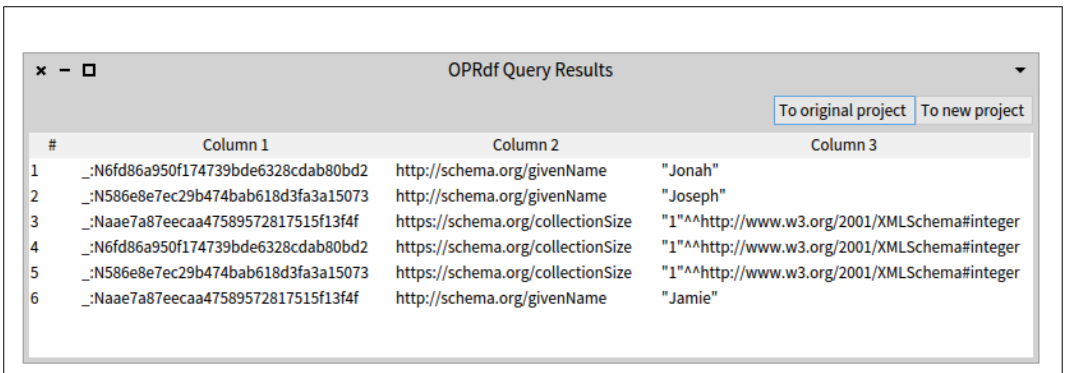

Figure 9: Query Results window

4.4.4 Exporting data into RDF format

With the resulting data (or any data you wish to export in general) being open, click *Diagram* > *Export* and select the preferred output format (see figure 10).

The final output to N-Quads should look similar to Snippet 8 (order of rows and values of blank nodes may be different and the default node specification is not strictly necessary).

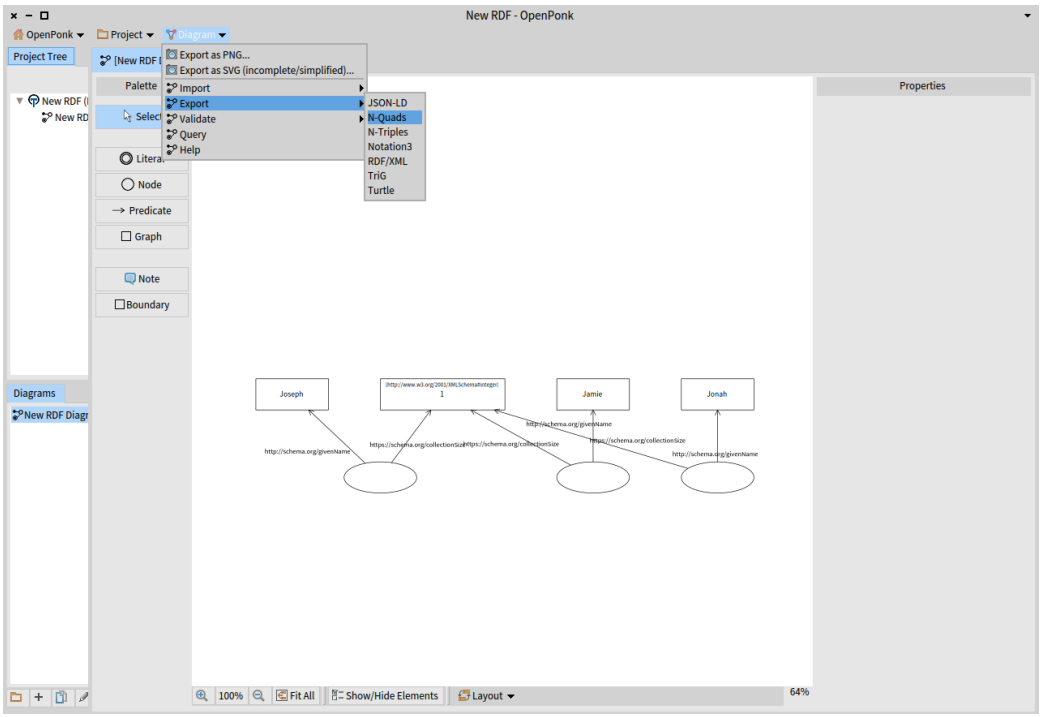

Figure 10: Exporting model to RDF format

```
1  _:b1
       <https://schema.org/collectionSize>
       "1"^^<http://www.w3.org/2001/XMLSchema#integer>
       <urn:x-rdflib:default> .
2  _:b1
       <http://schema.org/givenName>
       "Jonah"
       <urn:x-rdflib:default> .
3  _:b2
       <http://schema.org/givenName>
       "Joseph"
       <urn:x-rdflib:default> .
4  _:b3
       <https://schema.org/collectionSize>
       "1"^^<http://www.w3.org/2001/XMLSchema#integer>
       <urn:x-rdflib:default> .
5  _:b3
       <http://schema.org/givenName>
       "Jamie"
       <urn:x-rdflib:default> .
6  _:b2
       <https://schema.org/collectionSize>
       "1"^^<http://www.w3.org/2001/XMLSchema#integer>
       <urn:x-rdflib:default> .
```

Snippet 8: Case study: Resulting N-Quads data

4.5 Documentation

The project is documented in several ways. Aside from this thesis which describes concepts behind it and general structure, there are also readme files and naturally comments within the code.

There are two separate readme files in the enclosed archive. One is in the root folder and describes the structure of the archive, along with the general content of the files. The second readme file can be found in the project repository and explains how to install the plugin and its dependencies.

4.6 Performance profiling of import functionality

When testing the program with larger datasets, it has arisen that the performance of import code might be an issue. For example, importing Turtle file with approximately 1000 predicates took several minutes, which is significantly longer than could reasonably be expected.

Pharo fortunately features a Time Profiler (*Debug > Time Profiler*), which shows how much time was spent with execution of which call. The profiling was done using code in snippet 9.

```
| workbench controller |
workbench := OPWorkbench new.
controller := OPRdfDiagramController new.
controller
    model: (OPRdfDiagram new announcer: (workbench announcer));
    workbenchAnnouncer: (workbench announcer);
    view: (RTView new);
    layouter: (OPRdfLayouter new diagramController: controller;
                                 yourself);
    importJSONQuads:
        (NeoJSONReader fromString:
            (((FileLocator temp) / 'stw_100.txt')
                readStream upToEnd))
```

Snippet 9: Code used for performance profiling

The results have shown that `OPDiagramExplorer>>showElements:` is the method the most of the time was spent in, meaning most of the time is spent with creating elements rendering in the view.

It is likely there is room for optimization in this class, since the measured times were much larger than could be expected for such a relatively small dataset, but it would require deeper investigation of OpenPonk components and possibly even the underlying graphical libraries. Although it is unfortunate, it will have to be resolved at a later date.

4.7 Future developments

As was mentioned before, one of the bigger issues which could be improved on is the performance when loading larger dataset. This poor performance makes it unusable for anything but the smallest datasets, and should positively be the first one to be addressed.

For smaller improvements in functionality, there is currently almost no validation of the data entered by the user. Validation was experimented with, but there did not seem to be simple way to do this with Magritte, the library responsible for the property editing UI. Additionally, it is also currently not possible to move a node from one graph into another. This is mainly due to the fact the content of a graph currently does not move along with it when dragged, which would be required to implement moving of elements between graphs in a meaningful way. That is however a limitation of OpenPonk which is resolved in the upcoming version, meaning this will be addressed when upgrading to it.

Larger future developments are twofold. For one, it might be useful to develop tools for work with RDF ontologies, such as a panel that would show class hierarchy of given objects and properties associated with the classes. Secondly, it might be useful to extend supported standards to RDF* (RDF-star[9]). This standard is relatively new when compared to RDF, but introduces a simple way to describe additional predicate information. That may for example include a date when a contract takes effect, probability event occurs, or similar information, which can only be expressed in RDF by creating resources for instances of such predicates.

[9] https://www.w3.org/2021/12/rdf-star.html

Conclusion

The goal of this thesis was to implement RDF model editor plugin for the OpenPonk modelling platform. That included brief review of relevant technologies, such as the Resource Description Framework itself, and other RDF editing tools, brief exploration of available libraries and tools, and design, implementation, testing and documenting a solution. Optional goal was to establish a way to work with data loaded or created in the solution through SPARQL querying language.

In the analytical part of the thesis, basic concepts of RDF were explained, along with basics and examples of JSON-LD and N-Quads formats, which were both relevant to the implementation. It was shown that there do not appear to be graphical tools for editing RDF formats except for specific applications such as ontological models. Lastly it was also shown there are no libraries for Pharo that would allow for simple work with JSON-LD or any other RDF format.

In the practical part, it was explained how the plugin uses Python library RDFLib along with custom intermediary format. Basic structure of classes and packages was shown, and expected workflow including execution of SPARQL query was shown on a case study.

While I am certainly happy with the result in terms of capabilities of the implementation with all planned features working to great extent, I must admit the testing and documentation was not given as much attention as would be appropriate. Keeping that in mind, I am still inclined to say the goals of the thesis were met.

Bibliography

[1] *Pharo - The immersive programming experience* [online]. Pharo Project. Unknown date [cit. 2023-04-22]. Available at: https://pharo.org/

[2] *Pharo Cheat Sheet* [online]. Pharo Project. Unknown date [cit. 2023-04-22]. Available at: http://files.pharo.org/media/pharoCheatSheet.pdf

[3] *OpenPonk* [online]. Faculty of Information Technology, Czech Technical University in Prague. ©2022 [cit. 2022-03-11]. Available at: https://openponk.org/

[4] BLIZNIČENKO, Jan. *Podpora simulace a vizualizace v nástroji DynaCASE*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015. Available at: https://dspace.cvut.cz/handle/10467/63136/

[5] *RDF 1.1 Concepts and Abstract Syntax* [online]. W3C. ©2004-2014 [cit. 2023-03-11]. Available at: https://www.w3.org/TR/rdf11-concepts/

[6] HEATH, T. and C. BIZER. *Linked Data: Evolving the Web into a Global Data Space* [online]. New York: Springer Publishing Company, 2022. 122. ISBN: 978-3-031-79432-2. Available at: https://link.springer.com/book/10.1007/978-3-031-79432-2

[7] *JSON-LD 1.1: A JSON-based Serialization for Linked Data* [online]. W3C. ©2010-2020 [cit. 2023-04-04]. Available at: https://www.w3.org/TR/json-ld11/

[8] *RDF 1.1 N-Quads* [online]. W3C. ©2012-2014 [cit. 2023-04-23]. Available at: https://www.w3.org/TR/n-quads/

[9] *SPARQL 1.1 Query Language* [online]. W3C. ©2013 [cit. 2023-04-20]. Available at: https://www.w3.org/TR/sparql11-query/

[10] *Neologism 2.0* [online]. Semantic Society. Unknown date [cit. 2023-04-23]. Available at: https://github.com/Semantic-Society/Neologism

[11] *NeOn Wiki* [online]. Semantic Media Wiki. ©2014 [cit. 2023-04-23]. Available at: http://neon-toolkit.org/wiki/Main_Page.html

[12] *Protégé* [online]. Stanford University. ©2016-2020 [cit. 2023-03-11]. Available at: https://protege.stanford.edu/software.php

[13] *TopBraid Composer* [online]. TopQuadrant. ©2020 [cit. 2023-04-23]. Available at: https://archive.topquadrant.com/products/topbraid-composer/

[14] *WebProtégé* [online]. Stanford University. ©2016-2020 [cit. 2023-04-23]. Available at: https://protege.stanford.edu/software.php

[15] *JSON-LD - JSON for Linking Data* [online]. PaySwarm. Unknown date [cit. 2023-04-23]. Available at: https://json-ld.org/

Table of Figures

Table of Code Snippets

Table of Tables

Acronyms

RDF	Resource Description Framework
IRI	Internationalized Resource Identifier
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
URN	Uniform Resource Name
IDE	Integrated Development Environment
OS	Operating System
BPMN	Business Process Model and Notation
XML	Extensible Markup Language
MVC	Model-View-Controller
UUID	Universally Unique Identifier
JSON	JavaScript Object Notation
SQL	Structured Query Language
SHACL	Shapes Constraints Language
STON	Smalltalk Object Notation
UML	Unified Modelling Language
YAML	YAML Ain't Markup Language
UI	User Interface

Contents of enclosed archive