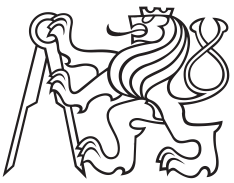
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science and Engineering

# Platform for semantic extraction of the web

**Jakub Podlaha**
Artificial Intelligence
podlajak@fel.cvut.cz

January 2015
Supervisor: Ing. Petr Křemen, Ph.D.

# Acknowledgement

I'd like to thank my parents and family for enormous support, my supervisor for endless patience and guidance and my friends for not letting me go insane.

# Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré informační zdroje v souladu s Metodickm pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 5.1.2015

# Abstrakt / Abstract

Tento dokument je pouze pro potřeby testování.

**Překlad titulu:** Platforma pro sémantickou extrakci webu

This document is for testing purpose only.

# Contents /

# Tables / Figures

# Chapter 1
# Introduction

During past few years the Web has undergone several bigger or smaller revolutions.

- WEB 2.0 and tag cloud
- HTML5 and semantic tags
- Smart-phones, tablets, responsivity and mobile web everywhere
- The run out of IPv4 addresses, nonexistent boom of IPv6
- Cloud technologies and BigData
- Bitcoin, Tor, anonymous internet
- WikiLeaks, NSA, Heartbleed and security concerns
- Google Knowledge Graph, Facebook Open Graph, ...

That is only several examples of some of the biggest recent technology booms and issues on the global network. So little can mean so much in such a global environment. The environment online is constantly changing, usually on a wave of some new, useful or, sometimes, terrifying technology or with popularization of a new phenomena. The Semantic Web technologies have been described, standardized and implemented for several years now [1]) and their tide seems to be near, though yet to come.

Semantic Web itself relates to several principles (along with their implementation) that allow users to add meaning to their data. This meaning brings not only a standardized structure, but also, as a consequence, the possibility to query and reason on data originating from multiple sources. Once given the structure, similar data can be joined in a form of a bigger bulk. Presenting this data publicly creates a virtual cloud. When put together this practices are called the Linked Data.

The intention of this work is to bring the Semantic Web technologies closer to users. Specifically it focuses on the process of creation of semantic data. We will propose a methodology for extracting and annotating data out of unstructured web content, along with a design of a tool, to simplify the process. The design will be supported by implementation of a prototype of the tool. Results will be confronted with real life use cases.

## 1.1 Problem Statement and Motivation

Giving meaning, i.e. semantization of web pages, gets more popular. Probably the most obvious example can be seen in the way the Google search engine serves its results. When possible, Google presents not only the list of pages corresponding to the searched term but also snippets of information scraped directly from the content of the pages such as menu fields parsed from CSS annotation or HTML5 tags, contact information or opening hours. When applicable Google also adds data from their own internal ontology, the Knowledge Graph[2].

[1]) One of the most recent standards – OWL2 – was released in 2008 [1]

What options are there to bring real semantic into a webpage? One direction to go is to annotate data on "the server side", i.e. at the time it is being created and/or published. When we are in position of owner of the data or server, we can help not only Google or DuckDuckGo[1]) to understand our website. To avoid confusion, this part is not focused on SEO – the search engine optimization[3], even though the topic overlaps in many ways. SEO primarily focuses on increasing the ranking of a webpage in the eyes of a search engine whereas pure semantization focuses on best describing the meaning of the pages content no matter how good or bad it appeals to anyone as long as it is valid according to standards of Semantic Web[4] and Linked Data[5].

In order to perform semantization on the server side, the person or engine creating the data have to use the right tool and put some time and effort giving the data the appropriate annotation. There are standards covering this use case.

In the simplest form the HTML5[6] brings in tags for clearer specification of the page structure (such as `nav`, `article`, `section`, `aside`, and others).

Microformats[7] define specialized values for HTML `class` attribute to bring standardized patterns for several basic use cases with fixed structure, such as *vCard* or *Event*. The microformat approach is easy to implement as it does not impose any extra syntax an can simply embed an existing page source. As the community around microformats states "(Microformats are) designed for humans first and machines second."

Last but not least, we can use joined power of HTML and RDFa [8] to annotate data on a webpage with an actual ontology. This technology is part of the Semantic Web stack and we will describe it closer in next chapter 2.

Annotating data on the server side enables users to use tools to highlight data they are specifically interested in, extract them and reason on them. Services can use annotated data, combine them and offer new results based on merged knowledge obtained from multiple sources. Providing data in such a form makes server a part of Linked Data cloud. For completeness let us mention some examples of utilities for extracting and testing or scraping structured data:

- Google Structured Data Testing Tool (i.e. rich snippets) [9]
- RDFa Play – tool for visualisation and extraction of RDFa content [10]
- LDSpider – a semantic data crawler [11]

Unfortunately, it is not always possible or desired by the web owner to embed semantics into their data and support it. Vast majority of the web holds plain text data without any machine readable meaning given to it, leaving it on human readers to understand it.

To bypass the gap between unstructured data present on the web on one side and rich, linked, meaningful ontologies on the other, we can take the opposite direction to the one described so far. We can take the unannotated data already present on the web and retrieve them in a form, defined by some ontology structure.

In some use cases the ontology of the desired data is yet to be created and the user is aware of the data structure and capable to manually spot and select the data on a web page. Currently there are not many tools allowing this kind of operation. The ideal implementation and the vision of result of this thesis will allow user to partially identify the structure of a webpage while leaving the repetitive tedious work on crawler following the same procedure repeatedly on all data of the page.

For such a process we need to create tools that allow users to address previously unstructured content, link it to resources of existing ontology and/or create these re-

[1]) An anonymous search alternative to Google `http://duckduckgo.com`

sources on-the-go. By using existing ontologies we would not only give the meaning to our data, but also create valuable connection to any other dataset annotated using the same ontology.

## 1.2 Use Cases

In a general case our goal is to "obtain data from a webpage in a semantic form." We have a webpage and optionally an ontology as an input and annotated set of data as an output.

For start, we will focus on data having a structure defined in HTML. The data might be structured as a table or set of paragraphs or any other set of HTML tags, and we will handle it on the level of these tags. Some text handling might be performed using regular expressions but usually we will simply select a HTML tag and use its content along with some annotation.

In the "friendliest" cases the data we want to scrape are formed in some repetitive form, most often a table. This is the best case as we can simply define structure on one row of the table and repeat the same pattern over and over. Sometimes the table spreads over several pages, so we need to define a way of advancing to the next page and start over.

Following sections contain description of several use cases that shall be solvable using design proposed in this thesis.

### 1.2.1 Use Case 1 – basic example case

`http://www.inventati.org/kub1x/t/`

The first use case is the simplest task that will be covered by the implemented prototype. As you can see on the picture 1.1 it consists of table holding values about people, and link to a detail page for one of them. On the detail page there is a field with "nickname".

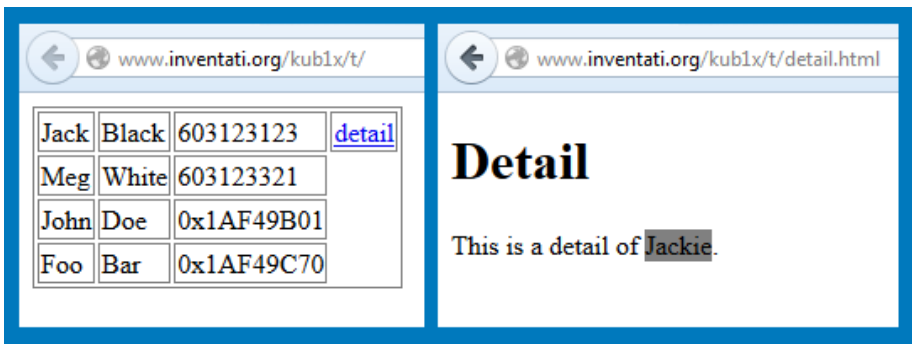



**Figure 1.1.** The example main page and detail page for the basic use case.

In order to fulfill this use case SOWL shall support following operation:

- Load the FOAF ontology that contains resources to describe data about people.
- Create scenario with two templates: init and detail.
- Save this scenario to a file.

CrOWLer shall be able to to perform following tasks.

- Accept and parse scenario created by SOWL.
- Follow this scenario while scraping data from the page.

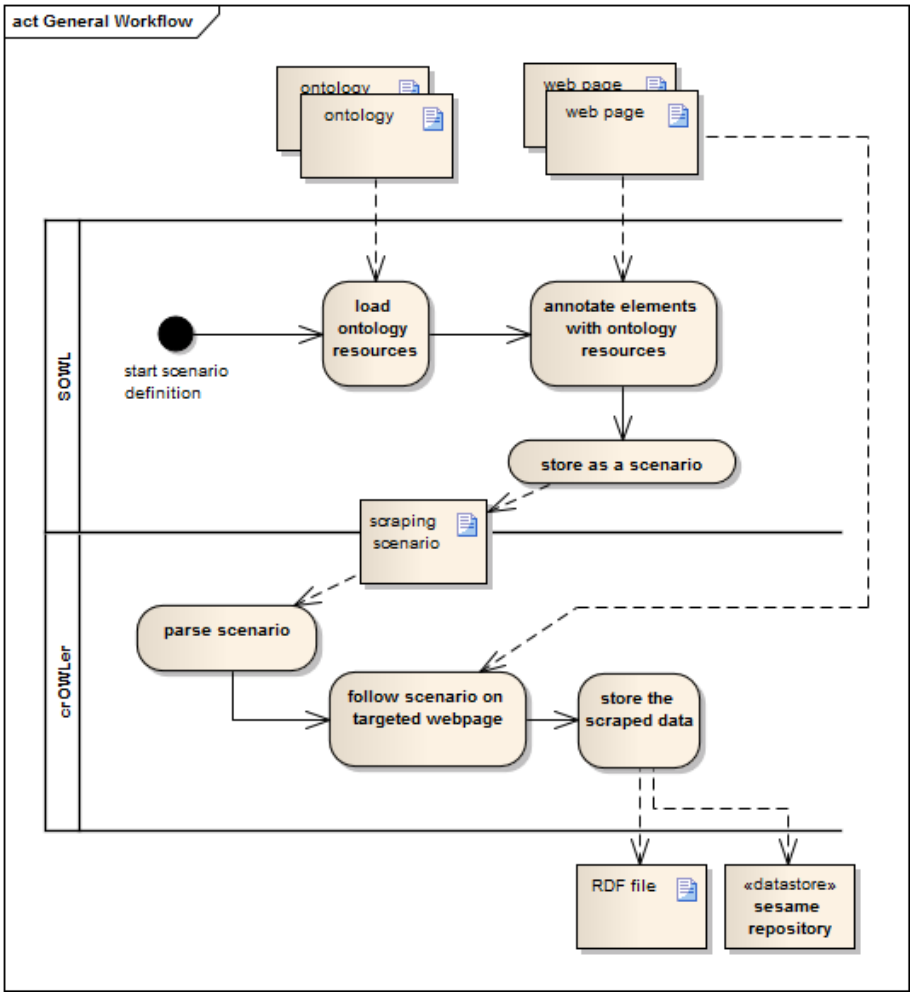



**Figure 1.2.** Diagram of general workflow as derived from presented use case

- Store results into RDF files.

This use case defines the simplest functionality that have to be implemented by both programs. It covers resources handling, scenario creation and running and finally storage of the results. It helps to define the proper behavior of the program as it is written in simple, valid HTML5 code without any JavaScript and all elements can be simply targeted by CSS or XPath selectors.

### 1.2.2 Use Case 2 – National Heritage Institute

`http://monumnet.npu.cz/pamfond/hledani.php`

The webpage of National Heritage Institute of Czech Republic 1.3 gives a public access to a table of damages of national monuments. This is of interest for project MONDIS [1]) partially developed on our school. Its main purpose is a documentation and analysis of damages and failures of cultural heritage objects.

The data were successfully crawled by the original implementation of crOWLer. The goal of following development is to replicate the behavior with new implementation using scenario driven crawling process instead of process driven by hardcoded configuration.

[1]) `https://mondis.cz`


**Figure 1.3.** Partial view at data on National Heritage Institute webpage

The main challenge of this use case lays in JavaScript. Each row of the data table has the `onclick` attribute defined. Unlike the classical "link" (also known as the anchor or `a` tag) the onclick attribute does not contain URL, but rather a JavaScript function content, that handles the click event. After closer investigation 1.4 we can observe that in this case, the function advances to the detail page of the clicked record by modifying a value of a hidden `input` tag and by submitting a `form` parametrized by the value.


**Figure 1.4.** A preview of HTML source analysis on National Heritage Institute webpage

If possible, we would simply simulate the user "click" action to advance to the detail page and the "back" action (usually performed by the Back button of browser or Alt+left keyboard shortcut) to get back and follow on next line. This approach will be analyzed further in this work.

If the stated approach can not be implemented to give the expected results the original approach will be simulated by the new scenario driven structure. This means crOWLer will be getting the content of the `onclick` attribute, parsing it using regular expression and combining it with a predefined pattern into an URL to be directly called using `call-template`.

Additionally this use case hides one more pitfall that, this time, challenges the selector creation. The web page uses JavaScript to colorize table rows when user hovers them with mouse cursor. Using a deeper analysis, we can figure out, that table lines are given additional CSS on certain mouse events. This is often a sign of poor web practices as the same behavior can be achieved by `:hover` CSS selector without a need of additional class, but it is an example of a challenge that our tool need to overcome. In this very case, we probably will not be able to generate selectors using CSS classes and will rely only on tag names, positions and other identifiers, if applicable.

Additional requirement on SOWL to those in Use Case 1 1.2.1:

- allow manual resources creation
- record the `click` event

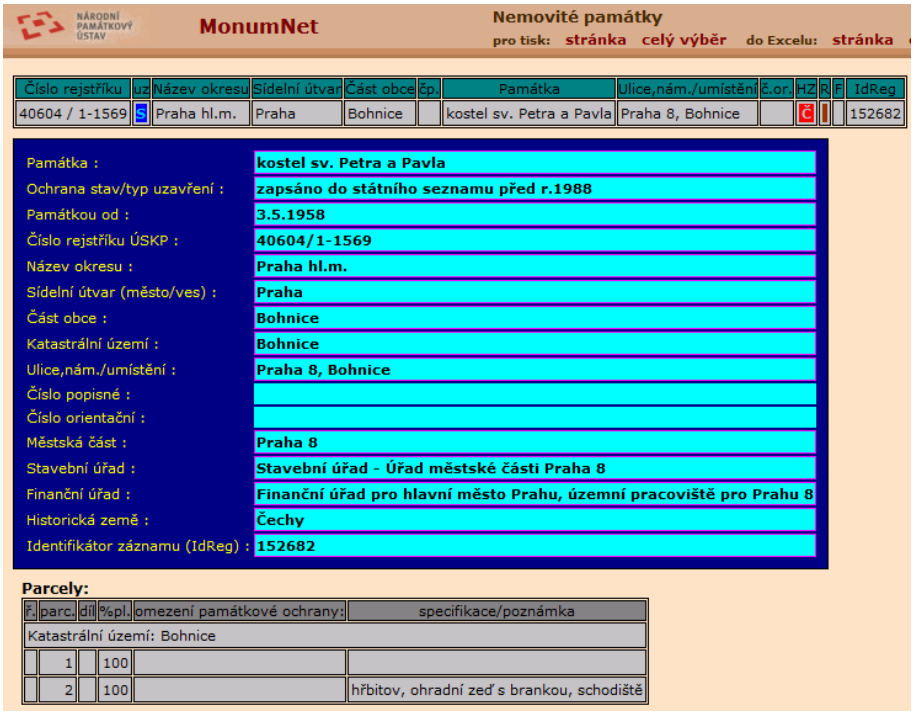



**Figure 1.5.** View on detail page on National Heritage Institute webpage.

- OR
- access the `onclick` attribute
- enable string handling using regular expressions
- record a `call-template` on the resulting URL

Additional requirements on crOWLer

- simulate the `click` event
- OR
- handle the attribute according to the string filters
- do `call-template` on the result as URL

The outcome of this use case and its analysis brings an important message. In many cases we will have to dive into the implementation of the processed webpage to find out how it behaves. In a vast majority of these cases it will require a web developer or coder to correctly and exhaustively define the scraping scenario.

### 1.2.3 Use Case 3 – Air Accidents Investigation Institute

`http://www.uzpln.cz/cs/ln_incident`

This is basic use case with a table, a detail page and a pagination. Everything is present in a clear HTML form without any interruption by JavaScript.

In this case we might consider replacing repetitive values by an object instance carrying the information. For example the table shows column “Event type” (in Czech original: “Druh události”). It contains constant values of “Incident”, “Flight accident” and several more. A resource can be created to denote these types of accidents. The resource corresponding to the string scraped from table would than be used as a value of object property instead of the original string literal. The original literal is assigned to this resource as a “label”.

For example we can use (in turtle syntax 2.8):

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix rlp:  <http://kub1x.org/dip/rlp#>
<rlp:event-xFuHbjA5> a <rlp:event>;
          <rlp:hasEventType> <rlp:flightAccident>.

<rlp:flightAccident> <rdfs:label> "Letecká nehoda"@cs.
```

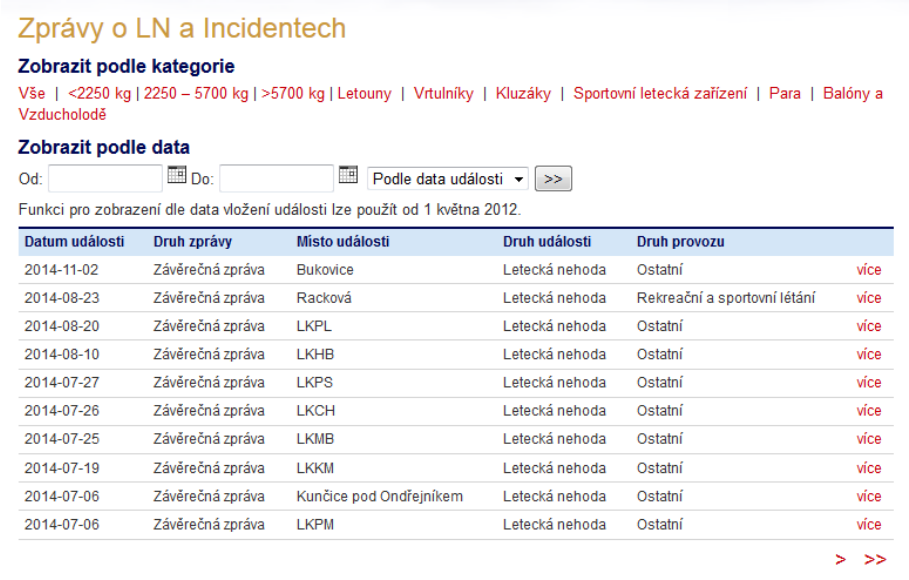
Instead of:

```
@prefix rlp: <http://kub1x.org/dip/rlp#>
<rlp:event-xFuHbjA5> a <rlp:event>;
          <rlp:hasEventType> "Letecká nehoda"@cs.
```

Motivation for the previous instantiation lays in the following use case. As it uses the same domain – flight accidents – it might use some of the resources previously defined here. For the *event type* it would probably use exactly the same instances, and would only add the English label to them.

This should not be much of a problem as long as we can specify an URI identifier when creating an instance of an ongological object. In the example above the identifier is: `<rlp:flightAccident>`. Another identifier in the example is the URI of the event: `<rlp:event-xFuHbjA5>`. This one was chosen from an URL of a PDF file on the page.

From previous paragraph, we define another useful functionality: conditioning on string literals and specifying URIs of instances directly in the scenario either as a constant string or obtained by combining other string values probably in a form of a pattern.



Zprávy o LN a Incidentech

**Zobrazit podle kategorie**

Vše | <2250 kg | 2250 – 5700 kg | >5700 kg | Letouny | Vrtulníky | Kluzáky | Sportovní letecká zařízení | Para | Balóny a Vzducholodě

**Zobrazit podle data**

Od: Do: Podle data události >>

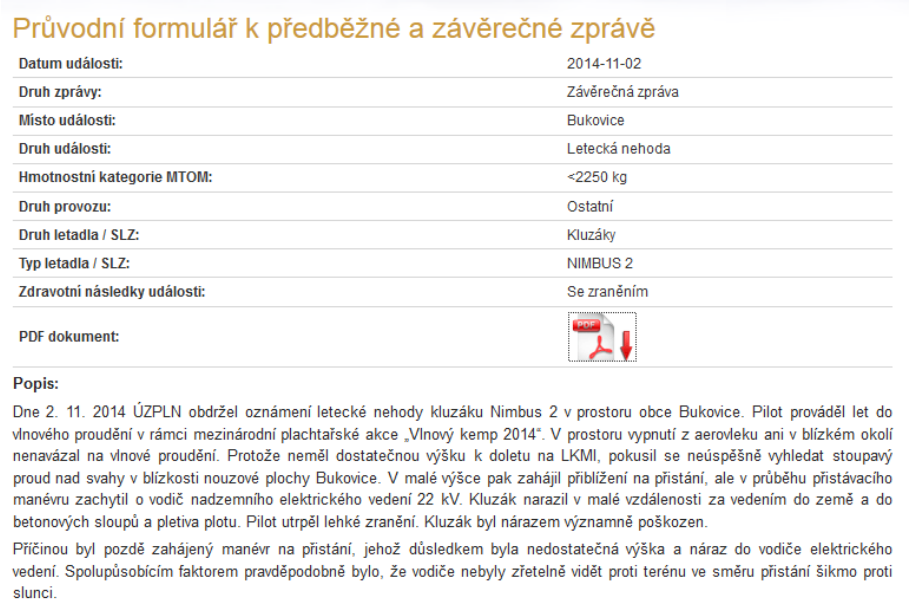
Funkci pro zobrazení dle data vložení události lze použít od 1 května 2012.

| Datum události | Druh zprávy | Místo události | Druh události | Druh provozu | |
|---|---|---|---|---|---|
| 2014-11-02 | Závěrečná zpráva | Bukovice | Letecká nehoda | Ostatní | více |
| 2014-08-23 | Závěrečná zpráva | Racková | Letecká nehoda | Rekreační a sportovní létání | více |
| 2014-08-20 | Závěrečná zpráva | LKPL | Letecká nehoda | Ostatní | více |
| 2014-08-10 | Závěrečná zpráva | LKHB | Letecká nehoda | Ostatní | více |
| 2014-07-27 | Závěrečná zpráva | LKPS | Letecká nehoda | Ostatní | více |
| 2014-07-26 | Závěrečná zpráva | LKCH | Letecká nehoda | Ostatní | více |
| 2014-07-25 | Závěrečná zpráva | LKMB | Letecká nehoda | Ostatní | více |
| 2014-07-19 | Závěrečná zpráva | LKKM | Letecká nehoda | Ostatní | více |
| 2014-07-06 | Závěrečná zpráva | Kunčice pod Ondřejníkem | Letecká nehoda | Ostatní | více |
| 2014-07-06 | Závěrečná zpráva | LKPM | Letecká nehoda | Ostatní | více |

> >>

**Figure 1.6.** View on list page on Air Accidents Investigation Institute.

- specifying a pattern for creation of URI of each instance
- adding language tag to all string values
- possible usage of geographical ontology
- possible usage of enumeration



Průvodní formulář k předběžné a závěrečné zprávě

| | |
|---|---|
| **Datum události:** | 2014-11-02 |
| **Druh zprávy:** | Závěrečná zpráva |
| **Místo události:** | Bukovice |
| **Druh události:** | Letecká nehoda |
| **Hmotnostní kategorie MTOM:** | <2250 kg |
| **Druh provozu:** | Ostatní |
| **Druh letadla / SLZ:** | Kluzáky |
| **Typ letadla / SLZ:** | NIMBUS 2 |
| **Zdravotní následky události:** | Se zraněním |
| **PDF dokument:** | |

**Popis:**

Dne 2. 11. 2014 ÚZPLN obdržel oznámení letecké nehody kluzáku Nimbus 2 v prostoru obce Bukovice. Pilot prováděl let do vlnového proudění v rámci mezinárodní plachtařské akce „Vlnový kemp 2014". V prostoru vypnutí z aerovleku ani v blízkém okolí nenavázal na vlnové proudění. Protože neměl dostatečnou výšku k doletu na LKMI, pokusil se neúspěšně vyhledat stoupavý proud nad svahy v blízkosti nouzové plochy Bukovice. V malé výšce pak zahájil přiblížení na přistání, ale v průběhu přistávacího manévru zachytil o vodič nadzemního elektrického vedení 22 kV. Kluzák narazil v malé vzdálenosti za vedením do země a do betonových sloupů a pletiva plotu. Pilot utrpěl lehké zranění. Kluzák byl nárazem významně poškozen.

Příčinou byl pozdě zahájený manévr na přistání, jehož důsledkem byla nedostatečná výška a náraz do vodiče elektrického vedení. Spolupůsobícím faktorem pravděpodobně bylo, že vodiče nebyly zřetelně vidět proti terénu ve směru přistání šikmo proti slunci.

**Figure 1.7.** View on detail page on Air Accidents Investigation Institute.

### 1.2.4 Use Case 4 – National Transportation Safety Board

`http://www.ntsb.gov/investigations/AccidentReports/Pages/aviation.aspx`

This use case serves mainly to demonstrate usage of the same ontology vocabulary on two different data sources. Additionally we might fill default values in place of missing ones in this table. For example the country value isn't specified for majority of the event records, but we can determine by the "State" field, that they happened in United States.

We will have to deal with JavaScript again. As we can see from the URL of the site (having the ".aspx" suffix), we are dealing with Active Server Pages, created by ASP.NET server. The whole table with all its sorting functionality and pagination is generated by the server and defined by the framework used on the server side. The pagination is of our consideration as it loads data into the table using AJAX call. This means data are loaded dynamically and we do not have easy access to the low level network communication happening in behind.

The options we have are analogical to those in second use case 1.2.2. We can either simulate the user action of "clicking on the next page button" or deeply analyze the JavaScript behind the pagination and perform the AJAX call manually.

The situation here is slightly different from the one in UC2 1.2.2 though. If we successfully emulate the user action for both use cases, in UC2 we will have to perform it for each line in the table (thus "during" creation of consistent ontological object and within iterating the table) whereas in this use case, we only perform the "click" when we need to load completely new set of data. The difference might not seem so essential at a first glance but the devil is in the detail: user action modifies or replaces current DOM object and the original information is lost. This does not apply to regular transfer to a new page using URL because we can use completely separate REST call. Technically it is identical to clicking a ling versus opening it in a new tab in your browser, only in crOWLer these operations are performed internally on lower level.

- adding default value if no content is found

| Report Number | NTSB Title | Accident Date | Report Date | City | State | Country | Other | Report |
|---|---|---|---|---|---|---|---|---|
| MAB1422 | Fire on Board Towing Vessel *Shanon E. Settoon* | 3/12/2013 | 12/10/2014 | Bayou Perot | LA | USA | 29°38.03 N, 90°10.63 W | PDF |
| RAB1414 | Collision of BNSF Railway Company and Union Pacific Railroad Trains Near Keithville, Louisiana | 12/30/2012 | 12/1/2014 | Keithville | LA | | | PDF |
| AIR1401 | Auxiliary Power Unit Battery Fire Japan Airlines Boeing 787-8, JA829J | 1/7/2013 | 11/21/2014 | Boston | MA | | | PDF |
| AAR1404 | Crash Following In-Flight Fire Fresh Air, Inc. Convair CV-440-38, N153JR | 3/15/2012 | 11/17/2014 | San Juan | PR | | | PDF |
| RAR1402 | Collision of Union Pacific Railroad Freight Train with BNSF Railway Freight Train | 5/25/2013 | 11/17/2014 | Chaffee | MO | USA | | PDF |
| MAB1421 | Marine Accident Brief: Breakaway of Tanker *Harbour Feature* from its Moorings and Subsequent Allision with the Sarah Mildred Long Bridge | 4/1/2013 | 11/12/2014 | Portsmouth | NH | | | PDF |

**Figure 1.8.** View on list page on National Transportation Safety Board webpage.

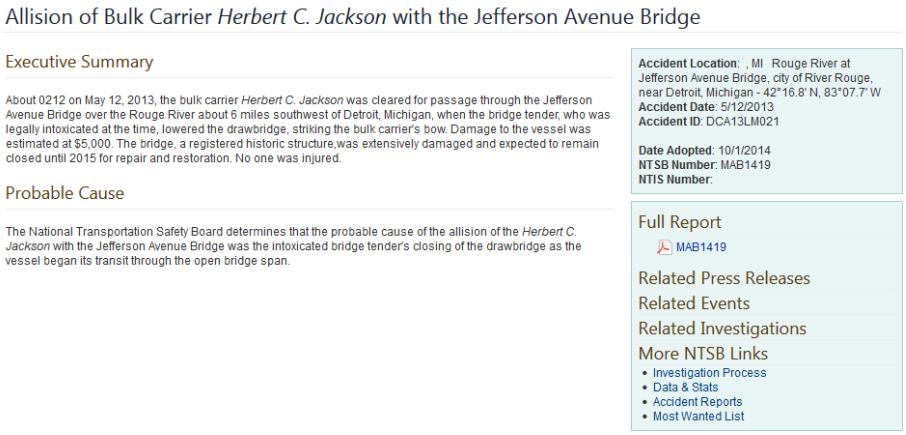



**Figure 1.9.** View on detail page on National Transportation Safety Board webpage.

## 1.3 Current solution crOWLer

The suggested base-technology is being developed on our faculty. Crawler called crOWLer serves the needs of extracting data from web. It follows the workflow of scraping data using manually created scenario with given structure and user-defined set of ontological resources.

In previous implementation, both, the scenario, followed by the crawler, and the ontology structure/schema are hard-coded into the crOWLer code. This requires unnecessary load of work for each particular use case, whilst in practice all the use cases share the same workflow.

1. load the ontology
2. add selectors to specific resources from the ontology
3. implement the rules to follow another page
4. run the crawling process according the above

In the original crOWLer implementation it was necessary to fulfill the first three steps with an actual programming. In order to perform this task, we needed to have a programmer with knowledge of Java programming language, and several technologies used on the web. Moreover a knowledge of the domain of data being scraped is needed in order to correctly choose appropriate resources for annotation. There is also a huge overload in preparation of development environment and learning time of the crOWLer implementation. The need of more elegant and generic solution is evident.

## 1.4 Proposed Solution and Methodology

To simplify the creation of guidelines – scenarios – for crOWLer, we will propose a tool that allows user to select elements directly on the crawled web page, with all the necessary settings, pass the scenario created to the crOWLer and obtain the results in a form of an RDF graph.

## 1.5 Specific goals of the thesis

- define use-cases for the semantic data creation
- create syntax for scenario used by crOWLer
- implement a web browser extension for creating these scenarios
- this extension shall
  - load and visualise ontology
  - join page structure and ontology resources in a form of scenario
  - serialize scenario and necessary ontological data
- parse the scenario by crOWLer
- run crOWLer following the scenario
- store the extracted data

## 1.6 Work structure

Next part of this work 2 will cover tools and technologies (and the related lingo) used in this work and in the field.

Chapter 3 will describe research on existing solutions and how they influenced results of this work.

Chapter 4 is the main part and describes the proposed design.

Chapter 5 gives details about the prototype implemented according to proposed design.

Both, design and implementation, will then be confronted against the real life use cases1.2.

# Chapter 2
# Principles and technologies

In following chapter we will provide basic information about technologies of Semantic Web and Knowledge Representation. The terminology often used in the field will be defined to help full understanding before we proceed to the design and implementation.

## 2.1 Technology of Semantic Web


**Figure 2.1.** Logo of Semantic Web

Wikipedia defines Semantic Web as a collaborative movement led by international standards body the World Wide Web Consortium (W3C) [4]. W3C itself defines Semantic Web as a technology stack to support a "Web of data," as opposed to "Web of documents," the web we commonly know and use [12]. Just like with "Cloud" or "Big Data" the proper definition tends to vary, but the notion remains the same. It is collaborative movement led by W3C and it does define a technology stack. It also includes users and companies using this technology and the data itself. Technologies and languages of Semantic Web such as RDF, RDFa, OWL, SPARQL are well standardized and will be described in following sections of this chapter.

As a general logical concept of the technology, languages of Semantic Web are designed to describe data and metadata, give them unique identifiers – so that we can address them – and form them into oriented graphs. The metadata part define a schema of types (or classes) and properties that both can be assigned to data and also relations between this types and properties themselves. Wrapped together this metainformation is being presented in a form of *ontology*. When some data are annotated by resources from such an ontology we gain power to *reason* on this data, i.e. resolve new relations based on known ones, and also to *query* on our data along with any data annotated using the same ontology.

On low level of the implementation we deal with simple *oriented graph*. The graph structure is defined in a form of *triples*. Each triple consists of three parts: *subject*, *predicate* and *object*, which all are simply *resources* listed by their identifiers (URIs). In this very general form we can express basically any relationship between two resources. On a level of classes and properties, we can define hierarchies, or set a class as a domain of some property. On lower, more concrete level we can assign a type to an *individual*. On a level of ontologies, in a way a "meta–meta" level, we can specify for instance an author, description and date it was released. Each of the relations is described using triples and together form one complex graph.

## 2.2 Linked Data

Wikipedia defines Linked Data as "a term used to describe a recommended best practice for exposing, sharing, and connecting pieces of data, information, and knowledge on the Semantic Web using URIs and RDF." Just like Semantic Web it is a phenomena, a community, a set of standards created by this community, tools and programs implementing these standards and people willing to use these tools and, of course, the data being presented. Linked data effort strives to solve the problem of unreachability of majority of the knowledge present on the web, as it is not accessible in machine readable form, doing so by defining standards and supporting implementation of those standards.

To imagine current state of the Linked Data we can take a look on the Linking Open Data cloud diagram[13]. The visualisation [1]) contains a node for each ontology and shows known connections between ontologies. The data originate from `http://datahub.io`, a popular web service for hosting semantic data. Current diagram visualises the state of linked data cloud in April 2014. As we can see in the center, many data resources are linked to DBpedia [2]), the semantic data extracted from Wikipedia. This best describes the notion of Linked data. When two datasets relate to the same resource, they can be logically linked together through this connection, as this way they state, they relate to the same thing.

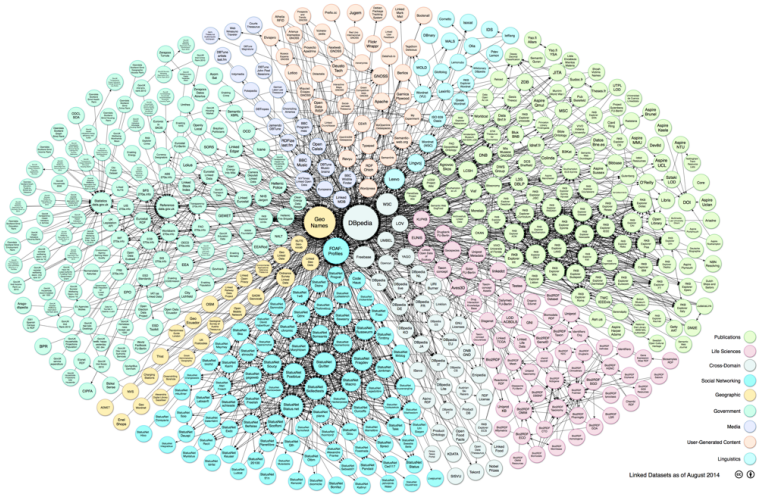



**Figure 2.2.** The Linking Open Data cloud diagram

## 2.3 RDF and RDFS

RDF is a family of specifications for syntax notations and data serialization formats, meta data modeling, and vocabulary used for it [14].

We will look closely on URI, the resource identifier, vocabularies and semantics defined by RDF, RDFS, and OWL, and serialization into Turtle and RDF/XML formats.

### 2.3.1 URI

In order to give each resource an unique identifier a Uniform Resource Identifier is used. This is mostly in a form of URL as we commonly know it as "web address" (e.g.

[1]) `http://lod-cloud.net/versions/2014-08-30/lod-cloud_colored.svg`
[2]) `http://dbpedia.org`

`http://www.example.org/some/place#something`). This literally specifies address of resource and in many cases can be directly accessed in order to obtain the related data. In some cases we can use URN as well. URN as opposed to URL allow us to identify a resource without specifying its location. This way we can for example use ISBN codes when working with books and records, or UUID[1]), a Universally Unique Identifier widely used to identify data instances of any kind.

### 2.3.2 RDF and RDFS vocabulary

In order to work with data properly, RDF(S) vocabulary defines several basic resources along with their semantics.

These are the basic building blocks of our future RDF graphs. The semantics defined in the specification [2]) allows us to specify class hierarchy, properties with domain and range as well as use this structure on individuals and literals. This is the most general standard that lays under every ontology out there.

## 2.4 OWL

Additionally to RDF and RDFS the OWL – Web Ontology Language, is a family of languages for knowledge representation. OWL extends syntax and semantics of RDF, brings in notion of subclasses and superclasses, distinction between *datatype properties* and *object properties*, defines transitivity, symmetricity and other logical capabilities of properties. When querying an OWL ontology, it allow us to use unions or intersections of classes or cardinality of properties. All this capabilities come in with well defined semantics. Usage of each feature brought in by OWL semantics extends requirements on resolver being used for reasoning on our ontology and brings in necessary computational complexity.

Including some more readings on OWL:

- `http://www.w3.org/TR/owl2-primer/`
- `https://en.wikipedia.org/wiki/Web_Ontology_Language`
- `http://www.w3.org/TR/2012/REC-owl2-quick-reference-20121211/`

## 2.5 RDFa

RDFa technology defines a concept of embedding content of a web document defined in HTML with resources from some ontology. Technically we create a invisible layer of annotations over the data that turns our content into machine readable record. This is accomplished by embedding the original HTML with custom attributes. Tools can be used to visualise this data [3]).

## 2.6 SPARQL

SPARQL is a semantic query language for data stored in RDF format [15]. Using SPARQL syntax we define a pattern of the RDF graph using triples and as a result we obtain such a nodes that form a subgraph of the original graph matching the given

[1]) `https://en.wikipedia.org/wiki/Uniform_resource_identifier`
[2]) Major part of the vocabulary is described in appendix C
[3]) `http://rdfa.info/play/`

pattern. So called SPARQL endpoints are the main entry points through which users can obtain data from openly available datasets [1)][2)].

Below you can see a simple example of a SPARQL query that returns a list of all resources from database that have a rdf:type associated to it.

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?target ?type
WHERE {
  ?target rdf:type ?type;
}
```

## 2.7 RDF/XML syntax

RDF/XML is one of formats into which we can serialize our RDF data [16]. It is a regular XML document containing elements and attributes from the RDF(S) vocabulary. RDF/XML is one of the most common formats for RDF data serialization. An example from popular FOAF ontology can be found in appendix D.

## 2.8 Turtle syntax

Turtle syntax is another popular syntax for expressing RDF. It allows an RDF graph to be completely written in a compact and natural text form, with abbreviations for common usage patterns and datatypes [17]. Its syntax suits more naturally to RDF data as it conforms the triple pattern. Follows an example about author of this work.

```
@base <http://kub1x.org/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#me> a foaf:Person;
      foaf:name "Jakub Podlaha".
```

[1)] http://dbpedia.org/sparql Dbpedia SPARQL endpoint
[2)] http://linkedgeodata.org/sparql LinkedGeoData SPARQL endpoint

# Chapter 3
# Existing solutions

In this chapter we will describe the research made on existing solutions for given task (scraping and annotating data from a web). The performed search was focused on tools directly targeting the problem, as well as libraries and technologies that could be included in the solution or existing open source programs we could build the solution on.

## 3.1 Semantic and non semantic crawlers

By researching existing solutions, there is currently no open source or openly available solution that would directly follow the required workflow and fulfill the requirements.

Existing tools named as "Ontology-based Web Crawlers" refer mostly to crawlers that "rank" pages being crawled by guess-matching them against some ontology. In those programs user can not specify data that are being retrieved. Moreover, there is no way to get involved in the crawling process. The tool is solely used to automatically rank the relevance of documents. They are solving different task where input is several documents and possibly an ontology and output is the best matching document.

In case we are trying to solve the input is one or more documents and one or more ontologies and the result is data retrieved from the documents and annotated with resources from the ontologies.

### 3.1.1 Advantages and pitfalls of Semantic crawlers

To properly target the benefits the semantification of the scraped data brings to the user, let us quickly follow an evolution from the most primitive technologies for scraping data to the advanced ones. The ultimate goal is to effectively search in data and maximally utilize the knowledge it carries.

The simplest approach is manual searching for keywords, or even simple browsing the web. That might be useful in some cases, but when there is a lot of data, it becomes exhausting.

Crawling data using simple tools like `wget --mirror` allows us to load data and then write a program or script to retrieve a relevant information. This approach takes a lot of energy for one time only solution of a given problem.

By storing such crawled data into database we obtain persistent database, possibly automatically obtained by the script from previous case. Such data is static, but can be queried over and over and possibly re-retrieved when becomes obsolete. Its structure is, however, based on programmers imagination an needs to be described in order to understand and handle the data properly.

When a triple store is used as the database in previous case we obtain one-time solution to our problem. This is technically equal to original state of crOWLer.

When using Ontology-based solution, tailor made for crawling and annotating data from web, we obtain several benefits "for free". The tool designed specially for this purpose makes it easy. Once the data is annotated, we can not only query on them, but

also automatically reason on them and obtain more or more specific/narrow results than with general data. The attributes and relations within ontology, that allow reasoning, are usually part of the ontology definition and as such comes in naturally without any extra effort.

Last for benefits: using ontology from public resource as a schema for our data can give us correct structure without need for building it from scratch. Also by using some common ontology, we can join together any accessible data structured according to this ontology and simply query on resulting super set.

Semantic crawling is not a silver bullet. The technology is only finding its place and uses and it is being shaped by the needs of its users.

For instance there is always a threat of inconsistency of an ontology when some data do not fit the rules or break structure of an ontology. In its state from April 2014 DBpedia states, there is 3.64 million resources, out of which 1.83 million are classified in a consistent Ontology [18]. That is only half of the data being arguably consistent with each other. That does not say the rest is bad. Only that it might cause a inconsistency and prevent us from reasoning if we include a wrong subset of the data.

Just like with "hardcoded" crawling technique, the semantic crawling is tightly bound to the structure of the crawled web. The web is being matched against some pattern described by selectors and the matching element, when found, is accepted for further processing. Any change on a webpage structure can lead to broken selectors or links during the crawling process and make the scenario partially or completely invalid.

Many web pages load their data dynamically using AJAX queries. Some pages simply change its content frequently (e.g. news pages, forums, user content pages, like video or music servers and social web applications). Crawling content on such servers would require almost constant crawling and would cause growth into massive ontology of, oftentimes, questionable quality.

The semantic crawling is an useful way to effectively obtain and query on data from the web, but it still have its challenges to overtake.

## 3.2 Analysis of crOWLer

A thorough analysis of the current program shall precede creation of the final design. We will focus on architecture, dependencies and components that will have to be reimplemented.

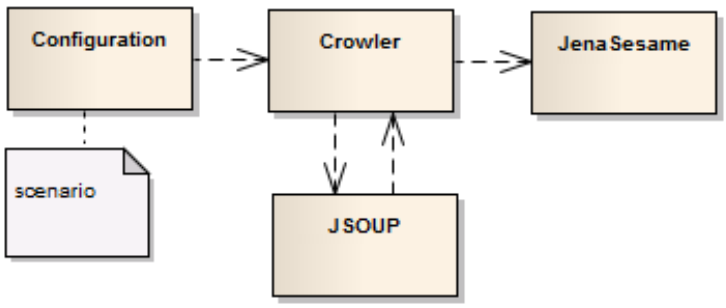



**Figure 3.1.** General architecture of the original crOWLer implementation

In original implementation crOWLer is a prototype of console Java application. It uses Apache Jena library [19] for handling ontological data and JSOUP library [20] for

accessing webpages and addressing elements. Instead of scenario file, crOWLer accepts Java `.class` files containing an implementation of `ConfigurationFactory` class. This factory class builds a `Configuration` object. In appendix E you can see definition of classes defining the configuration component of crOWLer. The class diagram in appendix F describes the `InitialDefinition` and the `Selector` classes that are main building blocks of configuration. Configuration defined using this structure specifies all the information needed for crawling process:

- webpages to be crawled in a form of list or pagination description
- way to address data on each page using JSOUP selectors
- definition of ontology resources used to annotate the obtained data
- setting of how URI will be created for each individual

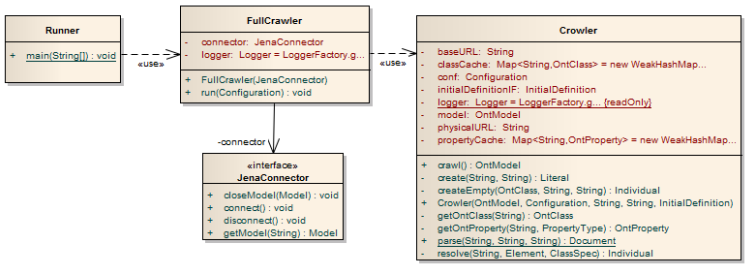



**Figure 3.2.** Core classes of original crOWLer implementation

The main program flow of crOWLer lays upon few core classes. The pair `FullCrawler`, `Crowler` (diagram 3.2) form the crawling process loop. In this loop `FullCrawler` fetches the source web pages and passes them one by one to the `Crowler`. The `NextPageResolver`, which defines list of pages to be crawled is structure implemented within the configuration and thus is specific for given problem instance. Results are stored in the outer loop after each scraped page. According to input parameters data are uploaded into Sesame repository using `JenaSesame` library, or locally in an RDF file.

The inner loop performed by `Crowler` finds a set of HTML elements as defined by the `InitialDefinition` class. Each of these elements serve as a root for a tree of ontological individuals linked by their properties. The tree is in configuration defined using `ClassSpec` and `PropertySpec` classes that hold definition of type of the individual and the assigned property respectively. The spec-classes also carry information about selector, used to find the corresponding HTML element. A collection of `Selector` classes is available and can be extended. JSOUP selector handling is implemented as well as selector chaining or resolving data from a link target.

In Crowler an individual of an ontological object is created after all his defined properties values are scraped within the inner loop, as the URI of the individual can be formed using one or more of these values. This way we can refer to the same object if we create individual of the same URI on two different pages for example.

### 3.2.1 Issues of crOWLer configuration

From deeper analysis of the original crOWLer source we can observe, that the whole scraping process rely on the configuration defining it – a set of Java classes, implementing the predefined interfaces and using the API provided.

This reveals the issue being addressed. Writing a crOWLer configuration requires knowledge of Java programming language along with knowledge of RDF technologies.

Programmer also gets into the position of ontological engineer when designing ontological resources used in the configuration. Knowledge of WEB technologies is needed in order to properly target elements on the webpage using JSOUP selectors. This is one of the hardest task as the selectors have to be manually extracted using for example browser console.

The scenario based approach, focused in this thesis, will enable user to bypass the Java programming and focus only on matching web structure with an ontology.

### 3.2.2 Confrontation with use cases – technical issues

In this section the capabilities of the original crOWLer implementation will be confronted with use cases specified for this work 1.2.

For all use cases a separate configuration would have to be created. We will mainly focus on problems specific for each case.

The first configuration of crOWLer was created for the *Monumnet* webpage of the National Heritage Institute, the UC2 1.2.2. Stating that the UC2 can be and was solved using the hardcoded configuration.

First we will focus on the structure of configuration. Following code is a simplified snipped of actual configuration building code of original crOWLer implementation. It uses `NPU` class as simple static storage for URIs used in our ontology. According to this configuration a *monumnetRecord* object is created for each table row as defined by the `initialDefinition`. The second part create *district* object with its label (found in third table column denoted by the `td:eq(2)` JSOUP selector) and assigns it to the record using *hasDistrict* object property. The `conf` object holds the configuration being passed to the actual crawler.

```
ClassSpec chObject = Factory.createClassSpec(NPU.monumnetRecord.getURI());
conf.addInitialDefinition(
       Factory.createInitialDefinition(
         chObject,
         Factory.createJSoupSelector("table tbody tr.list")));

ClassSpec sDistrict = Factory.createClassSpec(NPU.district.getURI());
chObject.addSpec(
           Factory.createOPSpec(
             Factory.createJSoupSelector("td:eq(2)"),
             NPU.hasDistrict.getURI(),
             sDistrict));
sDistrict.addSpec(true, Factory.createDPSpec(Vocabulary.RDFS_LABEL));
```

This pattern is, with some variation, repeated for all data properties and object properties. The interesting part is how crOWLer handles the detail page link. Just to remind a situation in UC2 1.2.2, each table row of the page uses unique `onclick` attribute in following form:

```
document.listpf.IdReg.value='131164'; document.listpf.submit();
```

The numerical value `IdReg` corresponds to last column of the row and holds the identification number of the national monument in the MonumNet system. As crOWLer handles every page as a static HTML document, there is no way to execute this code as a JavaScript handler. Instead it is being parsed by a regular expression and the result is used to fill in a format-string creating a URL. This URL locates the detail page for each table record.

```
Factory.createNewDocumentSelector(
  conf.getEncoding(),
  Factory.createAttributePatternMatchingURLCreator(
    "onclick",
    ".*([0-9]+).*",
    MONUMNET\_URL+"pamfond/list.php?IdReg=" + "{0}"));
```

Technically this is a form of a workaround, rather than systematic solution of the given problem. We can not securely rely on JavaScript code within the attribute as a part of data. It is important to realize, that the technique used on the webpage is rather non-standard and can not be effectively covered with general purpose tool without a need of problem specific solution.

Understanding the configuration implementation we will now briefly analyze the rest of use cases. crOWLer would solve the UC1 1.2.1 with quite basic configuration. Here we present a short example:

```
ClassSpec chObject = Factory.createClassSpec("foaf:Person");

conf.addInitialDefinition(
       Factory.createInitialDefinition(
         chObject,
         Factory.createJSoupSelector("tr")));

// First name
chObject.addSpec(
           Factory.createDPSpec(
             Factory.createJSoupSelector("td:eq(0)"),
             "foaf:firstName"));

// Analogically for the rest of properties

// Link to detail page
chObject.addSpec(
  Factory.createDPSpec(
    Factory.createChainedFirstElementSelector(
      Factory.createNewDocumentSelector(
        conf.getEncoding(),
        Factory.createAttributePatternMatchingURLCreator(
          "href", ".*", KUB1X\_URL + "{0}")),
      Factory.createJSoupSelector(".nick")),
      "foaf:nickname"));
```

This example is using only classes from the original crOWLer. Note at the bottom How we define following a link to the detail page. In proper implementation we would probably simplify the new document selector creation by wrapping it in a single factory method `createLinkTargetSelector`, which would internally create selector for the address targeted by `href` attribute of the link tag either absolute or relative to current document (so that we could avoid the explicit specification of URL using `KUB1X_URL` constant.

If we wanted to get more properties from the resulting page, we would reuse the `NewDocumentSelector` in combination with selector targeting value of each property. crOWLer always relates selectors to the document currently referenced by the outer

loop in the `FullCrawler`. Whenever a selector containing `NewDocumentSelector` is applied during the crawling process, a REST call is performed to fetch the targeted document. On a MonumNet webpage this means hundreds of thousands of calls for each run of the crOWLer (over 40 000 records each with 16 properties on detail page). Caching system can be implemented to reduce this amount to the necessary minimum. We are still bound by the double loop architecture though.

The UC3 1.2.3 is equal to UC1 according to configuration complexity. All links are implicitly specified in a form of hyperlinks without any interruption or dynamic content change. Moreover in crOWLer configuration we can specify what properties will, combined together, form URI of ontological object we are building. This is exactly the additional functionality required by UC3.

The specificity of fourth use case 1.2.4, as described, lays in AJAX driven pagination. Every "page change" event dynamically updates the content of the webpage. In this specific case we do not need to be alarmed as the pagination component is created using jQuery DataTables plugin [21]. Using this plugin the pagination is built on top of the data table after it has been completely loaded. In case of crOWLer, the plugin is never executed and the table remains complete and unchanged over the whole scraping process.

This is not always the case, though. Even the DataTables plugin itself supports loading data through AJAX so the alertness is more than appropriate. In the hypothetical situation when AJAX is used for data loading crOWLer would not be able to handle the pagination and would only access the first page. The additional data would have to be loaded using workaround similar to the one in UC2. And even if we successfully load the data we still might be unable to handle them by crOWLer. The AJAX call typically serves only the new chunk of data to be inserted into the page either in HTML or in JSON format. When in HTML, we would have to extend the configuration to correctly target elements in the reduced form of AJAX update. In case of JSON a completely new selector system would have to be added to crOWLer.

The situation dramatically changes if we use full stack web environment with JavaScript engine. In that case we would be able to ignore the background functionality of pagination and simply simulate click on the "Next" button. Enabling JavaScript has huge consequences and will be analyzed in a separate section 4.3.

### 3.2.3 Result form crOWLer analysis

The original implementation of crOWLer can solve tasks defined by specified use cases 1.2. The requirements on users of crOWLer are too high and the usability is very limited. The options of extending the configuration component will be examined during the design part4. The configuration can be either generated using scenario or completely replaced, if the scenario defines different crawling procedure (other than current double loop). The option of incorporating JavaScript will get an extra attention.

Previous section roughly define requirements on scenario for semantic crawler. To fully satisfy all considered use cases in all settings, in addition to the functionality implemented so far, we would have to cover the:

- following hyperlinks on a page,
- firing JavaScript and browser events,

- functions of transforming scraped data using regular expressions or key–value mapping.

## 3.3 Strigil

Strigil is a ontological scraping system developed at Faculty of Mathematics and Physics of the Charles University in Prague [1]). It represents an easily configurable tool that enables users to retrieve data from textual or weak structured documents. [22]

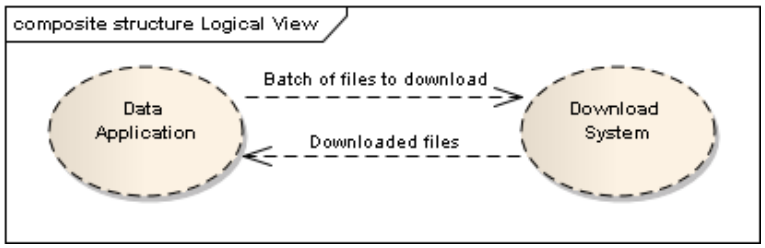



**Figure 3.3.** Overall Architecture of Strigil

It consists of *Data Application* in a form of webserver and backend service providing *Download System* for the application. The webserver offers frontend for configuring the crawling process. The application ten follows the configuration scraping the data and storing results while using the backend handle downloading. Strigil strongly focusses on the download process. Components of the backend conform in a structure of *DownloadManager*, *Downloader*s and *Proxy* servers that help to distribute the load of data being transfered.

The frontend part serves user interface for handling ontological data on top of a web being scraped. It internally creates a scraping script (will be referred to as Strigil Scraping Script or Strigil/XML) which strongly inspired format for scenario used in the actual implementation later in this work and will be closely analyzed in next chapter 4.

### 3.3.1 What problem does it solve?

The architecture of Strigil (more in H) is tailor made for parallel processing of documents. The installation of Strigil requires working Apache2 web server with PHP5, Tomcat, PostgreSQL database, OpenMQ service and several other components before the actual deployment of Strigil into the environment. The system is designed for processing many requests on targeted server, heavy loads of data and long running tasks. Its complicated architecture and installation process prevents it from being effectively used in occasional simple, yet non trivial, scraping tasks.

Moreover its download system fetches only the raw HTML data (just like the original crOWLer implementation) and treats it as static document. This way it can not properly handle dynamic content and temporal changes in documents performed by JavaScript for the exact same reasons that applied for crOWLer.

[1]) `http://xrg.ksi.ms.mff.cuni.cz/software/ld/ldi.html#strigil`

### 3.3.2 Strigil vs crOWLer

Because of the difference in complexity of Strigil and crOWLer, we can't correctly compare them one to one. But we might find a common subset of functionality. Strigil is a server, with frontend, scraping unit and download system. crOWLer is a tool without user interface and with download system reduced to simple REST calls. The common part then is the scraping unit.

The scraping algorithm of crOWLer has been described previously in section 3.2. It consist of outer loop over documents, inner loop over initial definitions and tree of recursive calls forming the ontological structure while scraping data from elements on the page.

Strigil has a slightly different approach. Instead of configuration it is guided by a scraping script. The script will be closely analyzed in the following chapter, but in general it defines a set of templates where one template is called at the beginning and each template can call any other template on some URL (i.e. on document located by the URL). Unlike in crOWLer the processing of each template is performed independently in Strigil. Each template call puts a request into the download system first. The actual execution of each template is fired asynchronously, when download of the targeted document is finished as notified by message from the Download System.

The inner part of a template conforms with structure inside crOWLer configurations. It defines tree structure of ontological classes and properties along with selectors specifying the position of targeted data. Resulting from different document resolving system, there are no `NewDocumentSelectors` in Strigil. In place of this selector, we would simply call another template on the new document. This approach is clearer than using chained selector, especially if we handle two or more nested documents. It is required though, to carry the ontological context from one template to another. This behavior is unfortunately neither mentioned in Strigil documentation, nor in examples examined.

### 3.3.3 Confronting Strigil with use cases

TODO

### 3.3.4 What inspiration it brings for crOWLer

## 3.4 Finding platform for frontend

In order to develop appropriate tool for generating scenarios, several similar tools were inspected for best practices, libraries, and possible extension.

The resulted implementation is named SOWL (short for SelectOWL) and refers to Firefox addon for creating scenarios for crOWLer. In following sections we will refer to SOWL as set of requirements and a envisioned expected result of this work. The actual implementation will be covered in later chapters.

### 3.4.1 InfoCram 6000 – ExtBrain

InfoCram 6000 is part of project ExtBrain [1]) that is developed here on Department of Computer Science. This specific part was implemented by Jiří Mašek and is described as "prototype of user interface for visual definition of extraction rules for ExtBrain Extractor". Its intended usage is very close to the usage of SOWL. It is an Firefox

[1]) `http://www.extbrain.net`

extension that generates rules (scenario) for extractor implemented as another part of the ExtBrain project.

The ExtBrain extractor is implemented in JavaScript as opposed to Java in case of crOWLer. It extracts data according to definitions by InfoCram 6000. The result is stored in JSON format thus not carrying semantic information, but only set of raw data in some form.

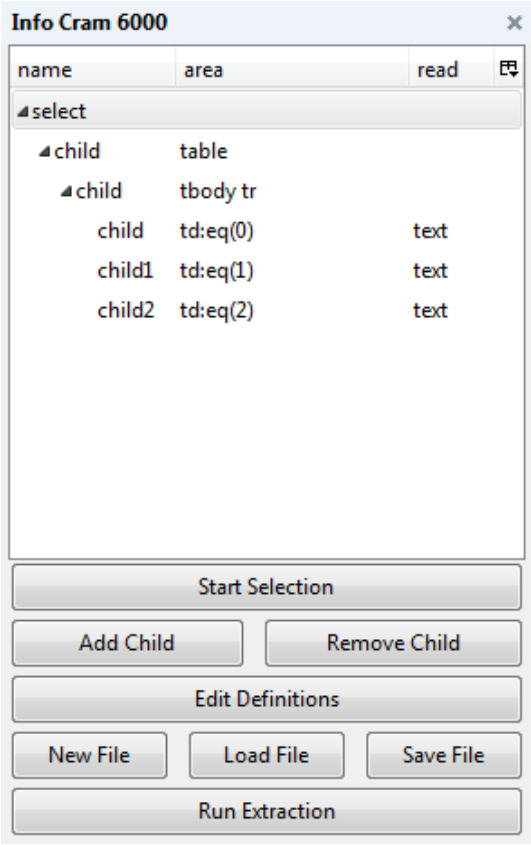



**Figure 3.4.** Main window of InfoCram 6000

Main part of the extension window shows a tree view with rules being edited. This view corresponds to required structure of scenario for crOWLer.

Interesting part it an engine for selection elements of page. Its implementation is based on Aardvark [1]), a Firefox extension that addresses this issue using mouse selection and several keyboard commands.

InfoCram does not use simple CSS or XPath selectors, but include Sizzle library to handle selectors for it. Sizzle is very popular library for handling selectors, which also defines its own selectors like `:eq()`, or `:first`. It is simpler and more expressive than CSS. Its popularity is mainly based on its involvement in jQuery library.

Being so close to required structure and workflow of SOWL, InfoCram 6000 served as the base implementation for it in the early stages. As can be seen at the end of this chapter, the first implementation named SelectOWL caries similar user interface and make use of several modules of the InfoCram implementation.

### 3.4.2 Selenium

[1]) `https://addons.mozilla.org/en-US/firefox/addon/aardvark/`

Selenium is a collection of tools for automated testing of web pages. This tools include:

- Selenium IDE – a Firefox plugin for creating test scenarios
- WebDriver – a set of libraries for various languages capable of running tests generated from Selenium scenarios

A user of Selenium, typically a web designer, programmer or coder, would create a scenario using Selenium IDE, in order to test his web server. From this scenario a unit test can be generated for desired programming language and in desired form (e.g. JUnit test case). Such a test can be simply included it in a set of tests for the web server project. WebDriver library needed for running these tests is available through Maven. There is also a chance to use PhantomJs no-gui web browser for running tests without a need for actual browser, for cases when tests are being executed automatically in background or on server environment without X server or other form of graphical interface. The capabilities of WebDriver make it one of the most popular testing platforms for web servers nowadays XXX.

- IDE - `http://www.seleniumhq.org/projects/ide/`
- plugins - `http://www.seleniumhq.org/projects/ide/plugins.jsp`
- current commands - `http://release.seleniumhq.org/selenium-core/1.0.1/reference.html`
- documentation - `http://docs.seleniumhq.org/docs/index.jsp`
- extending selenium API (blog, tutorial) - `http://adam.goucher.ca/?s=selenium&paged=2`
  - randomString example - `http://adam.goucher.ca/?p=1348`

Selenium IDE is a Firefox plugin that allow us to directly record user actions on webpage such as following links, storing and comparing values, filling in and submitting forms.

An attempt was made to implement SOWL as a plugin for Selenium IDE. This plugin would have two parts:

1. an extension of graphical interface
2. a formatter that would generate scenarios for crOWLer in some desired form

Certain limitations were discovered during development of this plugin. Selenium IDE, as being plugin itself, implements its own plugin system, through which it allows other developers to extend its functionality. The Selenium IDE plugin API allows us to use standard Firefox techniques along with predefined API, to extend the graphical interface and the functionality of the IDE respectively.

Graphical interface is defined using XUL, the standard Mozilla XML format for defining user interface. XUL defines an overlay system using which a new layer is defined and layered over existing part of application layout while extending or modifying it. The overlay system itself comes with Mozilla stack and can be used on IDE by default.

The functionality of IDE is, however, linked with its layout 3.5 and has to be taken in account. Selenium IDE internally defines set of commands that can be used in scenarios. List of default commands can be seen in dropdown on main screen of the IDE. This list can be extended, but the use and structure of commands is implemented internally in Selenium IDE. Addition of new commands is XXX accomplished by extending the `Selenium.prototype` object in registered plugin. After the extension is processed through internal command loader, a new set of commands is added for user to use.

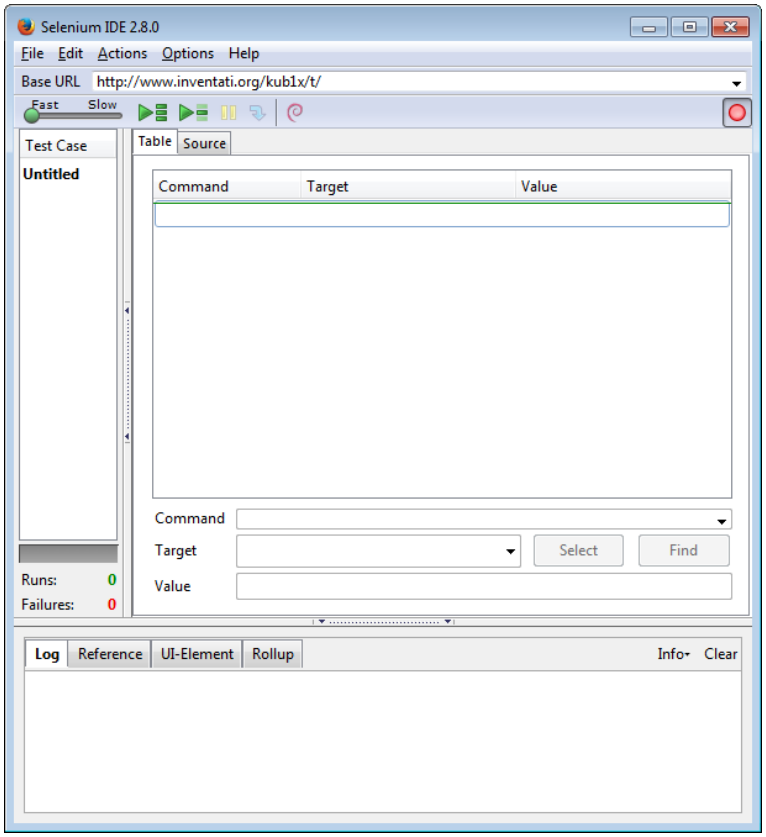



**Figure 3.5.** GUI of Selenium IDE showing the Command, Target and Value fields.

Commands in this system are recognized by their names as they are assigned on the prototype object the prefixes used are:

- do – the action commands – for performing user actions
- get and is – the accessor commands – for testing and/or waiting for a values on page and potentially storing it
- assert – the assertion commands – for performing actual tests

When command is generated the prefix is being stripped and according to type, multiple versions commands can be created. For example do commands have always "immediate" and "patient" version and in this principle `Selenium.prototype.doClick` will generate the `click` and `clickAndWait` command. Accessor commands are even more complex and generate eight commands for every single method (positive and negative assertion, store method, waitFor, etc.). Implementation of the command method defines, how Selenium IDE would behave when "replying" the scenario recorded. Technically it is possible to leave the implementation empty in the IDE and use it only as a command for WebDriver unit test.

None of the original command types corresponds to format of commands for handling the semantic annotation, like adding URI to element, recording creation of individual, assigning literal to its property etc. A new set of commands was suggested and partially implemented having the prefix "owl". This led to changes in core sources of Selenium IDE, which itself is a bad sign as it technically creates a new branch of the program. CommandBuilder had to be extended directly in the Selenium code as it is impossible to change its behavior through native Selenium IDE API. Unfortunately, even though the new command type was implemented, it is not possible to change the more general

concept of all commands. Every command is stored as `(name, target, value)` [1]) triple and from this format everything is derived. It is technically impossible to create command for example for literal along with its language tag as there is simply no field for it. For the same reason we cant create a command to create an ontological object of some type as a property of another object. These commands relate to each other, but such a behavior is not supported by the scenario editor in its current architecture. There is also no way to alter editor GUI for specific command. For instance, we can not offer autocomplete for input field when user enters URI of ontological resource. Such a feature would be an essential part of SOWLs workflow, and as a consequence these limitations are critical and disallow us from properly implementing SOWL on top of the Selenium IDE.

## 3.5 Libraries for SOWL

Research on existing JavaScript libraries that handle RDF data resulted in two promising libraries: jOWL and rdfquery. Both are based on the jQuery library and both claim to be capable to parse RDF files, which is the main requirement for us. Additionally the library might be used in SOWL as a storage for the loaded RDF resources.

### 3.5.1 jQuery

jQuery[23] is widely used JavaScript library that simplifies general tasks like DOM manipulation or event handling. A simplified selectors can be used to target DOM elements as jQuery internally uses Sizzle[24] library for selector handling. Compared to Vanilla JavaScript[25], jQuery produces more compact and coherent code.

Developers can extend the jQuery library with their own plugins. This is the case for two most promising JavaScript libraries handling RDF and OWL data, and so jQuery will be necessary if we decide to use either jOWL3.5.2 or rdfQuery3.5.3.

### 3.5.2 jOWL

The jOWL library is a jQuery plugin for navigating and visualising OWL-RDFS documents[26]. It can parse and handle RDF files, store them in its internal storage and query on them using subset of QUERY-DL language[27]. The library was last updated in 2008 [2]).

### 3.5.3 rdfQuery

rdfQuery[28] is a JavaScript library for RDF-related processing. It supports parsing RDFa, RDF, OWL formats for loading data. It can dynamically embed HTML webpage with RDFa data. rdfQuery is written as a jQuery plugin. The intended use of the rdfQuery library is to write queries over data stored in rdfQuery internal datastore in similar way as DOM objects are queried using jQuery. Moreover the whole concept is based on SPARQL and design in a manner that make the resulting JavaScript code look familiar when compared to native SPARQL query.

[1]) `https://code.google.com/p/selenium/source/browse/ide/main/src/content/commandBuilders.js the CommandBuilder implementation`
[2]) `https://code.google.com/p/jowl-plugin/`

### 3.5.4 aardvark

Aardvark is a JavaScript engine for in-place modifications of a webpage. It allows user to select, delete, or highlight part of HTML page. It has been released in two forms: as a bookmarklet and a Firefox extension. The later was used in a modified form in the InfoCram 60003.4.1 and later in one of SOWL (SelectOWL) prototypes??. This library help to implement the selection and serves as a framework for the selector generating algorithm.

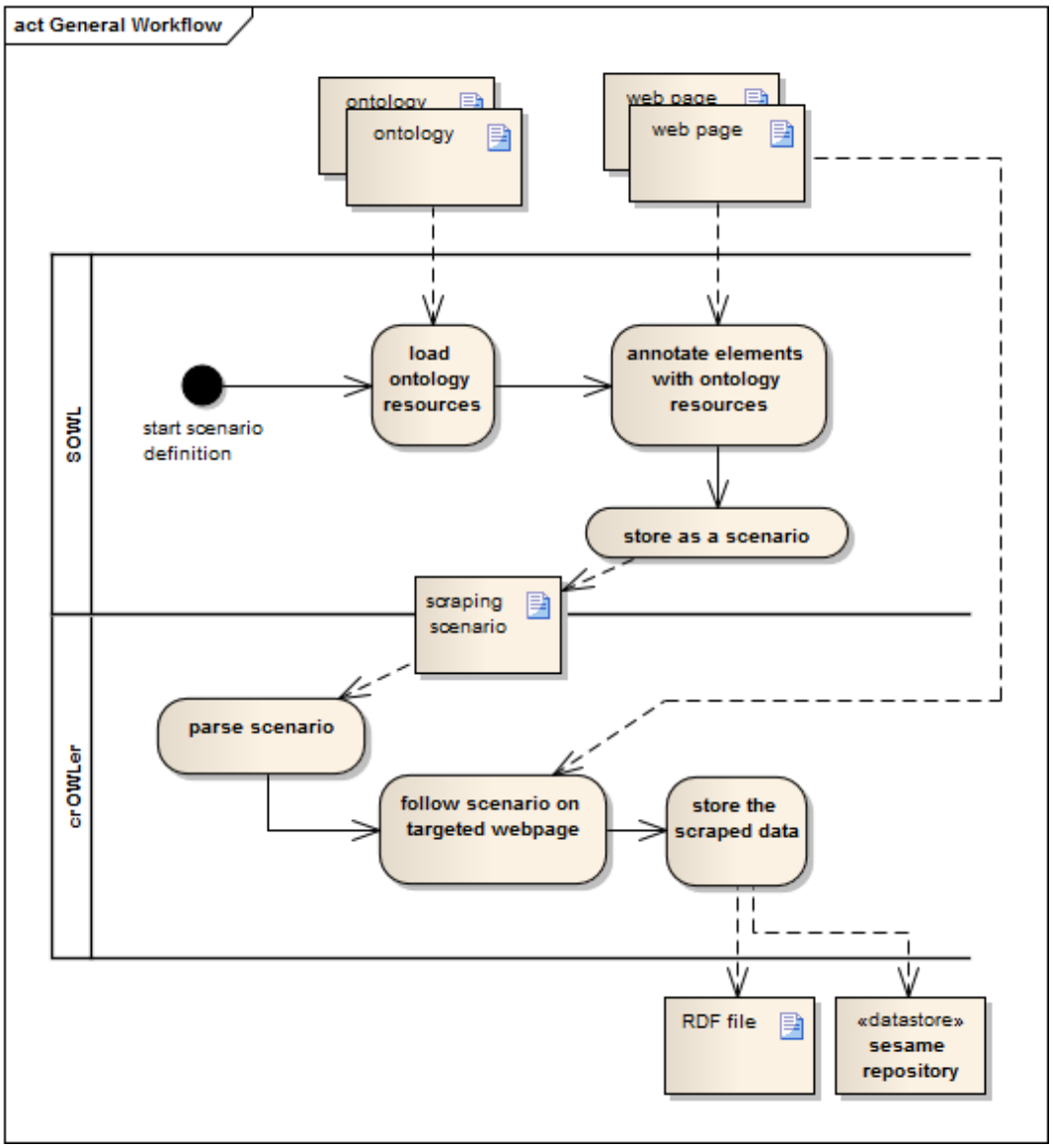
# Chapter 4
# Program stack design

This chapter defines the overall behavior of the program stack derived from presented use cases.

TODO use cases analysis here? XXX XXX XXX

## 4.1 Workflow

From the use cases defined and from analysis performed on existing solution we can derive the general workflow for both SOWL and crOWLer part of the implementation.



### 4.1.1 Main line

- user loads/creates ontology using SOWL
- user opens webpage with data
- user creates scenario using SOWL
  - user adds selectors to scenario steps
  - user adds resources to scenario steps
- SOWL sends scenario to crOWLer
- crOWLer crawls the web according to scenario and stores results in a file or repository

### 4.1.2 Scenario creation

- user starts scenario creation in SOWL
- loop until finished:
  - user creates a step in scenario
  - user selects an element on page, a selector is generated if applicable, on the step
  - user selects a resource, resource is updated on the appropriate field of the step, if applicable

### 4.1.3 Additional branches to Scenario Creation

- user can navigate through scenario by clicking scenario steps
- user can navigate through scenario by clicking ontological context
- user can navigate through scenario by clicking areas on webpage covered by scenario
- when user clicks on a hyperlink:
  - existing template can be assigned to the action (no need to actually follow the link)
  - new template can be created for resulting action (resulting page loaded, new template created)

### 4.1.4 crOWLer scraping

- user runs crOWLer passing it the created scenario
- crOWLer parses the scenario
- crOWLer scrapes data from the webpage following the scenario
- crOWLer stores the results in file or repository

## 4.2 Designing scenario format

One of main tasks of this work was to create format for scenario generated by SOWL and consumed by crOWLer. This scenario will describe information necessary for the crawling process: what operation to do (create ontological object, assign property to such an object, perform task with webpage).

This task is closely related to implementation peculiarity of semantic crawler: we are dealing with two separate contexts at the same time, the ontological and the web context. Ontological context holds current object (individual) to which we assign properties, web context hold current webpage along with currently selected element on that webpage. Scenario have to support operations to change each context separately and/or both at the same time.

### 4.2.1 Strigil/XML

Strigil, the scraping platform in order to solve similar problem as crOWLer introduces its own XML based Scraping Script format. Its documentation can be found here XXX [1]).

Basis of the whole script is system of *templates*. Each template has a name and mime-type declaring type of document the template is designed for. This information is needed as Strigil supports HTML and also Excel spreadsheet files. Templates call each other using `call-template` command anywhere in the script. This command accepts URL as an argument from its nested commands. Each template is called only with new URL, thus on new document. Of course URL of current document can be passed as an argument, but due to nature of Strigil, this would create completely separate context.

Strigil is tailor made for parallel processing. The architecture of the Strigil system contains not only scraping processor, but also a layer for distributed download queue processing and layer of proxy servers that can be used to spread the traffic and scale the download process horizontally. As the downloads are performed asynchronously and can be even delayed due to network lags and timeouts, there is no guaranteed order in which documents will be scraped. Each of Strigil templates create its own context when called. If we want to link data obtained from different template calls we have to use some additional techniques. For example we can assign some properly defined, non-random, unique identifiers to an object. This identifier have to be guaranteed to be the same for the same object through different template calls and potentially on different pages.

To handle ontological data manipulation the commands `onto-elem` and `value-of` are used. First one creates an individual of given type and, if nested into different `onto-elem` relates this new individual to its parent with some property. Literals are assigned to properties of parent object using `value-of` command with property name specified. This command is very powerful with usage regular expressions, selectors or nested calls of itself it can create arbitrary values from constants and data obtained from web page being processed.

Strigil also implements variety of functoins to help with processing of textual data. Function `addLanguageInfo`, for example, is widely used in Strigil scraping scripts to add language tags to string literals. The function call can be seen below.

```
<scr:function name="addLanguageInfo">
  <scr:with-param>
    <scr:value-of select="Hello World" />
  </scr:with-param>
  <scr:with-param>
    <scr:value-of text="en" />
  </scr:with-param>
</scr:function>
```

[1]) `https://drive.google.com/file/d/0B4On-lGb38CgWlAyZDhGbDV2TFk/edit Scraping script documentation`

Similarly we can use function `addDataTypeInfo` to add datatype flag, function `generateUUID` to obtain unique identifier or function `convertDate` to convert Czech and English dates into a common `xsd:date` format and several others. Some functions, like the last one mentioned, cover task-specific issues and Strigil does not define a way to extend the list of functions.

In early stages of SOWL development an attempt was made to use original Strigil/XML as a format of choice. An appropriate, consistent subset was chosen that would cover required use cases. Implementation of simple use cases revealed some pitfalls of this decision and revealed several suggestions for improvements on the approach and the format itself.

### 4.2.2 Adaptation of Strigil/XML format

Strigil creates its scraping script internally hidden under GUI and leaves user unaware of its actual content. It might still serve well, at least for developers, to keep the script compact and easily readable. Addition of language tag as seen in previous chapter, is widely used pattern that polutes the resulting script with unnecessary overload. Suggested improvement would separate this functionality into an extra attribute of the `value-of` tag named `lang`.

The same suggestion can be applied to the data-type specification. Moreover *implicit parsing* of known data types would not only simplify the scraping script, but also help to clean and clear the resulting data.

Let us imagine hypothetical scenario of two similar tables on one page containing two sets of data in the same format. For such a case we would need to define a template on subset of DOM and call it twice with different root node. Creation of `dom-template` and `call-dom-template` tags would solve this issue and would allow scenario creator to narrow down his focus to a subpart of the scraped webpage. This would be particularly useful on complicated pages with a lot of nested HTML. `dom-template` and `call-dom-template` would be defined within a single `template` tag and unlike `call-template`, they would *keep* the ontological context co call of `value-of` within `dom-template` would assign a property to individual created by `onto-elem` wrapping the current `call-dom-template` call.

The architecture of Strigil (distributed downloader) suggests that it uses simple raw HTML pages as they were downloaded and uses JSOUP to extract data from it as JSOUP is the selector system of choice. Many webpages, or even web applications, make use of dynamic AJAX calls to fetch additional data after the presentation layer of the web is shown to the user. Strigil does not handle these cases by default. The internal AJAX code could be analyzed and simulated using `call-template` call, but this requires deep knowledge of the webpage being processed. In crOWLer we opted to switch from JSOUP to WebDriver library and use PhantomJs, a no-GUI web browser. This technology allow us to handle webpages the same way as user sees them.

Usage of actual full-stack web browser with JavaScript engine long with WebDriver allows us to inject and execute arbitrary JavaScript code into the processed webpage. In order to make full use of this feature we can define `function-def` tag which would define JavaScript function with name and parameters and contain its code. To execute this function we would call `function-call` and identify it by its name. Return value of this function can be then used the same way as the one from `value-of` tag.

From the experience with development on Strigil/XML we can derive, that it is tied with its intended use for distributed downloader and it lacks some functionality. In SOWL we would almost necessarily modify its formal definition and thus it is of consideration if we can not make use of more appropriate format.

### 4.2.3 SOWL/JSON

As all Firefox extensions, SOWL is written entirely using JavaScript with additional HTML defining the graphical layout. Early stages of implementation generated XML based on Strigi/XML format using hardcoded XML snippets and string formating – approach often used on webpages with dynamically loaded content. A string holds a snippet of HTML or XML structure with placeholder. This placeholder is replaced by either a value or by another already processed snippet. This way piece by piece the whole scenario is generated. This solution is not hard to implement, but brings in poor maintainability and with additional complexity it looses elegance, readability and can even cause performance issues.

Original data of the scenario created by SOWL are stored naturally in JavaScript object. Using standard JavaScript method `JSON.stringify()` we can immediately generate JSON serialization of such object. This way we have structure similar to the original defined by Strigil/XML, but in flexible structure. Obviously some adaptations are necessary. Nesting is recorded using the `steps`, the header section is redesigned for the JSON structure. For example instead of listing prefixes in a single string of XML attribute, we define object ontology with a map of prefix–URI pairs.

The original semantics of `onto-elem` and `value-of` was preserved, only limited to its basic use. `value-of` serves to assign literal properties or to retrieve textual values for its parent scenario step.

An example of the scraping script can be found in appendix I.

### 4.2.4 Consequences of conversion to JSON format

According to difference in syntax between XML and JSON do not have `text content` of elements like XML elements can. In JSON we simply reserve a property for a value that would be otherwise specified this way in corresponding XML. Strigil, however, does not explicitly use the textual values and everything is specified using attributes. Some elements return textual values to their parents to handle, and in these cases it might be suitable to enable textual values as constants instead of the required element.

Another syntactical distinction is that JSON does not explicitly define child nodes. Everything is property in JSON object, so we, again, assign a property to store the child nodes. Child nodes are held in ordered list which in JavaScript corresponds to an array. As we build a structure of scenario steps, the reserved property will be simply called `steps` for every element that allows child nodes (e.g. `onto-elem` or `template`).

Technically each JSON object quacks like a hash map [1]) with a string keys and value of any JavaScript type. We can benefit from this loose structure. For example we can use any key to store a substep, not only the `steps` array.

The `onto-elem` command benefits exactly from this difference between XML and JSON. In original Strigil/XML the `onto-elem` tag allow us to specify URI of the resulting individual (as commonly denoted by the *about* property), by taking it from from its *first child* which is expected to be `value-of` tag. Needles to say, this specification lowers robustness as the position in the XML file is not enforced by the syntax and can be easily unintentionally broken by accidental swap of two elements, although it would not invalidate the files syntax and thus would not be captured by the script parser as an error. In the JSON format we lack the notion of child elements. Even when we simulate it as mentioned before, we would only cause the same indetermination. So instead, we simply reserve a property named `about` exactly for the described use.

[1]) `https://en.wikipedia.org/wiki/Duck_typing`

## 4.3 JavaScript and events support

Special attention have to be payed when dealing with direct interaction with DOM elements and script execution. WebDriver supports injection and execution of JavaScript as well as simulation of user interactions like *click* on element or *back* and *forward* navigation. Even though it brings great power there are considerations and great limitations to be taken in account.

WebDriver supports execution of JavaScript directly on webpage loaded in the driver. This is done by calling `executeScript` or `executeAsyncScript` function on the `driver` object. First argument of these functions is string defining content of JavaScript function we want to execute. Header and actual call of this function will be added for us before it gets attached to the webpage. We can pass any number of accepted arguments to these functions and they will be accessible through standard `arguments` object in on the JavaScript side. Types, corresponding to standard JavaScript types are supported as arguments: number, boolean, String, WebElement or List of any combination of the previous [1]). The second – asynchronous version returns immediately with a `response` object. It provides callback as additional argument to the JavaScript call. This callback is used for synchronization when accessing the result on the `response` object from Java.

```
JavaScriptExecutor exec = (JavaScriptExecutor)driver;
List<WebElement> labels = driver.findElements(By.tagName("label"));
List<WebElement> inputs = (List<WebElement>) exec.executeScript(
    "var labels = arguments[0]," +
    "    inputs = [];" +
    "for (var i = 0; i < labels.length; i++) {" +
    "  var name = labels[i].getAttribute('for');" +
    "  inputs.push(document.getElementById(name));" +
    "} return inputs;", labels);
```

In simple cases we can use JavaScript to extend functionality of crOWLer. It might be used as a complex string formatter, parser for nontrivial values etc. In following example it is used to condition on attribute value of an anchor tag. A document location if the href tag contains a hash symbol `#` (often used when the link is handled by JavaScript function.

```
JavaScriptExecutor exec = (JavaScriptExecutor)driver;
WebElement el = driver.findElement(By.cssSelector('a.detail'));
String result = (String) exec.executeScript(
"var elem = arguments[0];"+
"var href = elem.getAttribute('href');" +
"return (href ==='#' ? window.location.href : href);", el);
```

Previous example is simple, yet if we wanted to cover it with our scenario implementation we would bring a lot of single-problem-specific syntax into the scenario. We would have to use special notation for obtaining current URL and for conditioning on values. Following code demonstrates how this functionality might look like if it was covered only by scenario syntax without usage of JavaScript. The `getCurrentUrl` function is inspired by Strigil.

```
{
  command: "condition",
  condition: "ne",
```

[1]) `http://goo.gl/Hhwq3l Selenium JavaScriptExecutor documentation`

```
    param: "#",
    value: {
      commad: "value-of",
      selector: "a.detail",
    }
    onfalse: {
      command: "function",
      value: "getCurrentUrl",
    }
  }
```

We have declared the `condition` command with implementation of *ne* the "not equal" operator (and for completeness we would implement all the other un/equality operators) and the `function` command with implementation of *getCurrentUrl* which, again, probably is not the last function to be implemented. All this would require update of the scenario parser, the implementation for commands and all their attributes and thus update of the whole backend every single time, new functionality is needed. The advantage of this approach is that user does not have to know JavaScript and understand how it is called in WebDriver in order to use advanced conditioning and/or value formating.

It is disputable if a set of extra commands in scenario syntax and hence extra controls in scenario editor would be more understandable than a single field for JavaScript function. Technically by adding conditioning and function commands, we are inclining towards building a new programming language. To offer the best for the user, implementing both is the option: basic conditioning to easily direct the scenario flow along with a set of functions to format and modify string and other values as well as enabling JavaScript execution for complex problems.

With use of JavaScript the same scenario step as in previous example would look as follows:

```
  {
    command: "value-of",
    selector: "a.detail",
    exec: "var href = elem.getAttribute('href');" +
          "return (href ==='#' ? window.location.href : href);"
  }
```

In this case we embedded only the `value-of` with a single attribute that takes JavaScript function body. From there we have technically unlimited power for extending the functionality of the crOWLer without need of changing the Java implementation.

Note that compared to example in Java the first line of the original JavaScript was omitted:

```
var elem = arguments[0];
```

It will be automatically prepended every time, we exec JavaScript on a single DOM element. It is a simple helper and does not invalidate any users input (as in JavaScript we can redefine variable as many times as we want). Similarly we will predefine variable `elems` when a list of elements is passed, `value` when passing a string or number to a JavaScript function.

But with great power comes a great current squared times resistance [1]) XXX. With usage of JavaScript as suggested in previous paragraphs we have to take in account two major considerations.

[1]) `http://www.xkcd.com/643/`

Firstly, JavaScript function can accept any number of parameters and return an arbitrary value. In both cases, the parameters and the return value can be of any of the allowed types (as JavaScript is not strongly typed language). We thus have to specify what exact parameters are being passed to a function and what result of what type is expected. We also have to implement a robust way of controlling this specification and properly define a fallback-on-error behavior. This is especially important as we might want to use JavaScript function not only as a string filter, but also for example as a universal selector where we struggle with classical selectors. Any additional use have to be described separately before it can be universally used.

More importantly, there is the second consideration. Any DOM element is accessible from any JavaScript function using for example the `document.getElementByTagName` method. When an element is replaced or even removed, it becomes invalid from the Java context. Modification to an element can cause unexpected behavior of its reference in Java too. The same applies for operations on the whole page. When a link is followed, the original DOM tree is dropped and all references are lost.

To better describe the underlying behavior during this issue, below you can see a simple test. When link is clicked, the WebDriver follows the link in current window and the reference to the original DOM is lost.

```
WebDriver wd = new FirefoxDriver();
wd.navigate().to("http://www.inventati.org/kub1x/t/");
WebElement a = wd.findElement(By.cssSelector("a"));
System.out.println(a.getText()); // Prints "detail"
a.click();
System.out.println(a.getText());
// throws org.openqa.selenium.StaleElementReferenceException:
// { "errorMessage":"Element does not exist in cache", ... }
```

The previous can be partially solved by sandboxing the code in a closure. By doing so we can hide some essential object in global scope like `window` or `document` and make it harder to do inappropriate operations on the DOM. In following example we create the described partial sandbox:

```
JavaScriptExecutor exec = (JavaScriptExecutor)driver;
WebElement elem = driver.findElement(By.css("div.wewant"));
exec.executeScript(
"return (function(elem, window, document) {" +
funStr +
"})(arguments[0])", elem);
```

This technique is not completely secure (for example the element passed as an argument does have reference to its parent which is already leak of intended sandbox). Proper sandboxing would require implementing whole JavaScript engine in JavaScript [29] which is probably too much for our intentions.

In crOWLer, we can now distinguish between two ways of ascending to another HTML page:

1. using `call-template` command
2. using JavaScript or user event such as “click” or “back”

The `call-template` is always called on an URL and always creates new web context, keeping the original one untouched. It actually behaves like call stack, so when we return from the template call, we can follow on the original DOM tree. Just to note:

compared to corresponding Strigil command, crOWLer persists the ontological context throughout this call, and so we can relate to it when assigning properties.

Direct interaction with current window in any way that changes page location will, however, irreversibly invalidate all the elements of current DOM. This does not have to mean we can not use this functionality all together. Probably the best solution would be to only allow DOM modifying operations on the bottom level of templates (i.e. within the `steps` property of the `template` command in scenario). At this place we only hold the `body` of current document and as such we can simply replace it with the newly loaded content. In the original crOWLer implementation, this would be the spot between two "Initial Definitions".

Even thought the JavaScript is sandboxed in WebDriver, it is still running in a browser in your computer and could technically submit some data on a web. Security issues have not been considered so far, but might become a point of interest when we take in account an option of obtaining and executing scenarios from unknown sources.

## 4.4 User Interface

Here the required structure of user interface is described.

### 4.4.1 SOWL user interface

The user interface of SOWL shall be presented in a form of sidebar. The sidebar shall have two parts: a scenario editor and resources list. Scenario editor shall contain a tree shaped structure of steps of the scenario being created along with panel for editing the general settings of the scenario. The resources list shall accept dropping of ontology files which would load it into current dataset. Addition of resources manually shall be possible using a button. The list shall show all currently loaded resources and allow textual filtering.

SOWL shall enable tag selection on the webpage being processed by clicking or other user action.

### 4.4.2 crOWLer user interface

CrOWLer is a console application. It shall accept scenario as one of its parameters. Following settings shall be enabled using parameters as well:

- setting of target directory for RDF files
- setting of sesame repository for the result storage

## 4.5 Model

Presenting proposed design of the two programs the SOWL Firefox addon and the crOWLer Java application.

### 4.5.1 SOWL model

Current recommendation of Mozilla Developer Network suggests developing new addons using their native SDK. It allows creation of restartless addons, uses new API and limits usage of older libraries or low level calls by wrapping it in consistent API.

The SDK based addons have partially predefined structure. The *background script* runs in its own scope and uses the SDK API to control the addons behavior. The

*content script* is a JavaScript code that is injected into a webpage but runs in its own sandboxed overlay, while having access to pages DOM and JavaScript content. In SOWL, the scenario editor will be placed into a sidebar. Sidebar holds standard HTML window object in which the JavaScript code is running. All three components communicate via textual messages using `port` object offered internally by by Firefox.

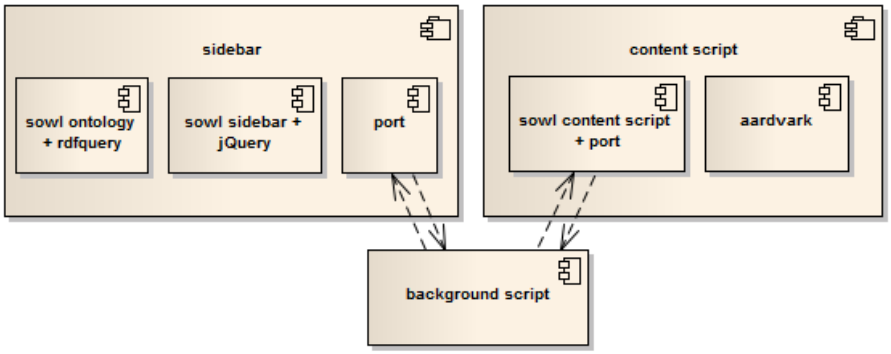



**Figure 4.1.** A component structure of the SOWL Firefox addon.

### 4.5.2 crOWLer model

In the new implementation of the scraping backend the original JSOUP component will be replaced by WebDriver. WebDriver, with its support for JavaScript, will help to handle dynamic content and brings in new possibilities for the crOWLer itself. The original configuration component is replaced by parser for the SOWL/JSON scenario format. The core crOWLer is also reimplemented according to new set of instructions (i.e. commands in the scenario) and the new web interface (i.e. the WebDriver instead of the native Java Jsoup library).

The overall architecture then looks as follows:

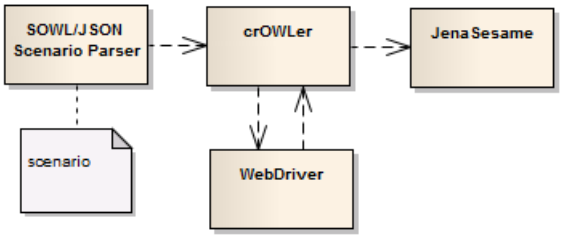



**Figure 4.2.** A new overall architecture of the crOWLer implementation.

# Chapter 5
# Program Implementation

This chapter describes the implemented prototype of SOWL–crOWLer tool stack. The relation between tools can be seen on diagram 5.1.

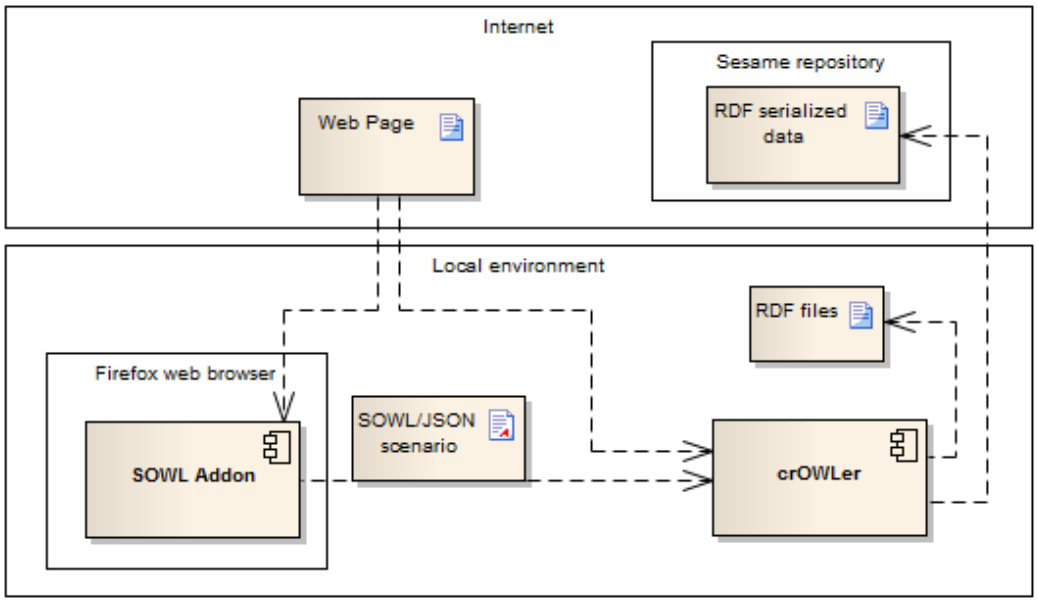



**Figure 5.1.** Overview of the whole stack and files exchanged

## 5.1 SOWL implementation

During testing of various technologies and frameworks several prototypes of the scenario creator was built. The first one called SelectOWL was native Firefox addon build on XUL and calls to Firefox low level API. Development of SelectOWL was discontinued in favour of new addon with shortened name SOWL. The new addon is based on Firefox addon SDK. The structure of the addon is completely different from the original one and the JavaScript of the addon runs in different context too. The new SDK is the recommended approach now and offers more flexible functionality and more intuitive code structure as the user interface is defined using classical HTML instead of XUL. The original version is kept in the repository for reference [1]).

### 5.1.1 Parsing Ontologies in JavaScript

Both jOWL vs rdfQuery were tested on common ontologies (FOAF, Dublin Core, Good Relations). Results shown that the newer rdfQuery library more accurately implements the standard behavior for handling RDF resources.

[1]) `https://github.com/kub1x/selectowl/tree/master/ff-extension`

Specifically in jOWL all resources have only one type. This type is determined when parsing input XML fileby a lookup cascade: if the type is not determined by the explicit RDF type property, the parser would look into the overlying tag name.

rdfQuery, on the other hand, properly stores all the data in form of triples in its internal dataset implementation. By using this approach it offers correct results and is our library of choice.

Even though rdfQuery currently serves for parsing of input files only, we might consider utilizing its reasoning capabilities in future development.

### 5.1.2 Targeting elements on webpage and generating selectors

Inspired by the InfoCram project we decided to use Aardvark code in order to target elements on webpage and obtain their selectors.

In early stages the native addon code for Aardvark was used. Unfortunately this code uses some internal Firefox API and had to be replaced when new Firefox SDK was used for the SOWL development.

In current Implementation of SOWL we create different type of Firofox extension using new SDK. Moreover the aardvark code is injected directly into the webpage using the Content Script feature of the Firefox SDK. According to these differences the bookmarklet version better fits the needs and is used. The aardvark code is included in the addon files extended with features necessary for SOWL. Namely the event handling was extended by drag/drop events and selector creation algorithm was added.

The selector creation was implemented as specified in XXX design-¿selector creation XXX. Even though it was rewritten it behaves almost identically as in InfoCram. We simply bubble up the DOM tree until we meet our context. On each element we try to generate unique selector according to the elements parent element. The last method to try is the `:nth-child()` selector which always exists and targets the correct element, but is also most prone to failures due tu structure changes. If possible ID or class attributes are used to target the element.

As use case 2 XXX shown we can not always rely on class selectors as they are often dynamically modified by pages JavaScript. For this reason the class selector are disabled by default, but they are supported by crOWLer and can be manually specified in the selector field. Aardvark shows the class of a hovered element on its label to simplify this task.

## 5.2 crOWLer implementation

The current implementation of crOWLer forms the architecture on picture 5.2. Even though the overall architecture holds visually the similar structure as the original implementation, the result is technically brand new program. Change from configuration system to scenario changed the input handling and influenced structure of the core algorithm from loop-based to template-based. The main library for web communication was changed from JSOUP to WebDriver which combined with scenario led to complete reimplementation of the core. The only part derived from original crOWLer are the Jena and JenaSesame libraries for handling the ontological models and storage of RDF data. The complete architecture can be better seen on the component model in appendix .

XXX XXX XXX
XXX XXX XXX
XXX XXX XXX

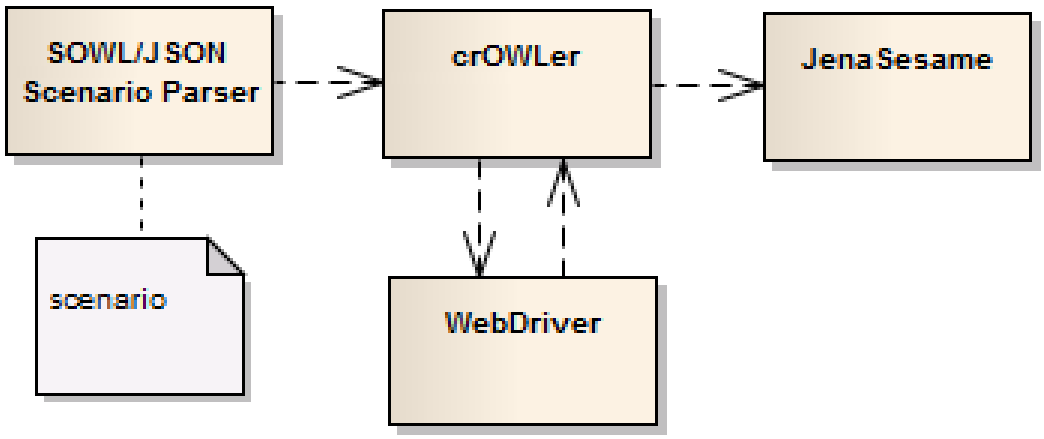



**Figure 5.2.** The overall architecture of new crOWLer implementation

A new structure was implemented holding a Scenario object with its steps. In this form the Scenario is passed to main loop. Instead of `FullCrawler` based on JSOUP we created WebDriver based solution, the `WebDriverCrawler`.

### 5.2.1 architecture

XXX include class diagram..?

# Chapter 6
# Results and Tests

A design was proposed.
A prototype was made.
XXX What else do you need?

# Chapter 7
# Conclusion

TBD

It is suitable to notice, that in many cases the intentions and activities of semantic web community focus on government data [30]. The common goal leads us to turn the web into an open, accessible source of knowledge and data of all kinds, linking the data together where possible. Naturally the governmental data and statistics get the most attention. Government handles, collects and is ofthen obliged to publish in some form a lot of data and statistics. Not always this form complies with standards of semantic web. Sometimes it might even be the case of intentional presenting of malformed data or obfuscation. In the big picture, misinformation of people seems to be the major thread to democracy as we usually envision it. By supporting the creation of semantic data we are naturally taking part in this movement. The hope is to bring government data closer to people to help overcome the information gap that prevents each one of us from being adequately informed about how our resources are being spent an how our countries are truly led and offices driven. I hope this and any follow-up work will serve to support this common vision.

# References


[1] *Web Ontology Language – Wikipedia.*
https://en.wikipedia.org/wiki/Web_Ontology_Language.

[2] *Google Knowledge Graph – Wikipedia.*
https://en.wikipedia.org/wiki/Google_Knowledge_Graph.

[3] *Search Engine Optimization – Wikipedia.*
https://en.wikipedia.org/wiki/Search_engine_optimization.

[4] *Semantic Web – Wikipedia.*
https://en.wikipedia.org/wiki/Semantic_Web.

[5] *Linked Data – Connect Distributed Data Across Web.*
http://linkeddata.org/.

[6] *HTML5 – Wikipedia.*
https://en.wikipedia.org/wiki/HTML5.

[7] *Microformats.*
http://microformats.org/.

[8] *HTML + RDFa 1.1 – Support for RDFa in HTML4 and HTML5.*
http://dev.w3.org/html5/rdfa/.

[9] *Google Structured Data Testing Tool.*
http://www.google.com/webmasters/tools/richsnippets.

[10] *RDFa Play – the RDFa data visualisation tool.*
http://rdfa.info/play/.

[11] Robert Isele, Jürgen Umbrich, Chris Bizer, and Andreas Harth. *LDSpider: An open-source crawling framework for the Web of Linked Data.* In: *Proceedings of 9th International Semantic Web Conference (ISWC 2010) Posters and Demos.* 2010 .
http://iswc2010.semanticweb.org/pdf/495.pdf .

[12] *Semantic Web – W3C.*
http://www.w3.org/standards/semanticweb/.

[13] *Linking Open Data diagram.*
http://lod-cloud.net.

[14] *Resource Description Framework – Wikipedia.*
https://en.wikipedia.org/wiki/Resource_Description_Framework.

[15] *SPARQL Protocol and RDF Query Language – Wikipedia.*
https://en.wikipedia.org/wiki/SPARQL.

[16] *RDF/XML – Wikipedia.*
https://en.wikipedia.org/wiki/RDF/XML.

[17] *Turtle – Terse RDF Triple Language – W3C.*

[18] *DBpedia – the Datahub.*
http://datahub.io/dataset/dbpedia.

[19] *Apache Jena*.
http://jena.apache.org/.

[20] *JSOUP – Java HTML parser*.
http://jsoup.org.

[21] *DataTables – Table plug-in for jQuery*.
http://www.datatables.net.

[22] Nečaský M. Stárka J., Holubová I.. Strigil: A Framework for Data Extraction in Semi-Structured Web Documents. 2013, paper submitted to 15th International Conference on Information Integration and Web-based Applications & Services, Vienna, Austria, 2013..

[23] *jQuery*.
http://jquery.com.

[24] *Sizzle JavaScript selector library*.
http://sizzlejs.com.

[25] *Vanilla JS*.
http://vanilla-js.com.

[26] *jOWL – Ontology Online*.
http://jowl.ontologyonline.org.

[27] Petr Kremen. *Towards SPARQL-DL Evaluation in Pellet*. 2007.
http://weblog.clarkparsia.com/2007/10/26/towards-sparql-dl-evaluation-in-pellet.

[28] *rdfQuery – RDF processing in your browser*.
https://code.google.com/p/rdfquery.

[29] *JavaScript in JavaScript (js.js): Sandboxing Third-Party Scripts*.
http://sns.cs.princeton.edu/2012/04/javascript-in-javascript-js-js-sandboxing-third-party-sc

[30] *Open Government Data*.
http://opengovernmentdata.org/.

# Appendix A
# Assignment

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

## DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Jakub Podlaha**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Platform for Semantic Crawling of the Web**

Guidelines:

1. Become familiar with semantic web technologies, namely with Linked Data concept, and RDF(S), OWL 2, SPARQL languages.
2. Design a platform for creating semantic extraction scenarios of the web. The platform will provide a UI for searching a suitable ontological schema, construction of the extraction scenarios, their execution using existing tools and visualization.
3. Test the platform on the selected public data sets and evaluate its potential for semantic web authoring.

Bibliography/Sources:

[1] Tom Heath and Christian Bizer (2011) Linked Data: Evolving the Web into a Global Data Space (1st edition). Synthesis Lectures on the Semantic Web: Theory and Technology, 1:1, 1-136. Morgan & Claypool.
[2] OWL Primer. http://www.w3.org/TR/owl2-primer/, cit. 19.12.2013

Diploma Thesis Supervisor: Ing. Petr Křemen, Ph.D.

Valid until the end of the summer semester of academic year 2014/2015


doc. Ing. Filip Železný, Ph.D.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, March 3, 2014

# Appendix B
# Abbreviations

| | |
|---|---|
| MDN | Mozilla Developers Network |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| URN | Uniform Resource Name |
| RDF | Resource Description Framework |
| RDFS | RDF Schema - set of classes and properties providing basic elements for the description of ontologies |
| OWL | Web Ontology Language |
| SPARQL | SPARQL Protocol and RDF Query Language - query language for semantic databases/triplestores |
| foaf | friend of a friend - a popular ontology for describing personal information and relationships |

# Appendix C
# RDF and RDFS vocabulary

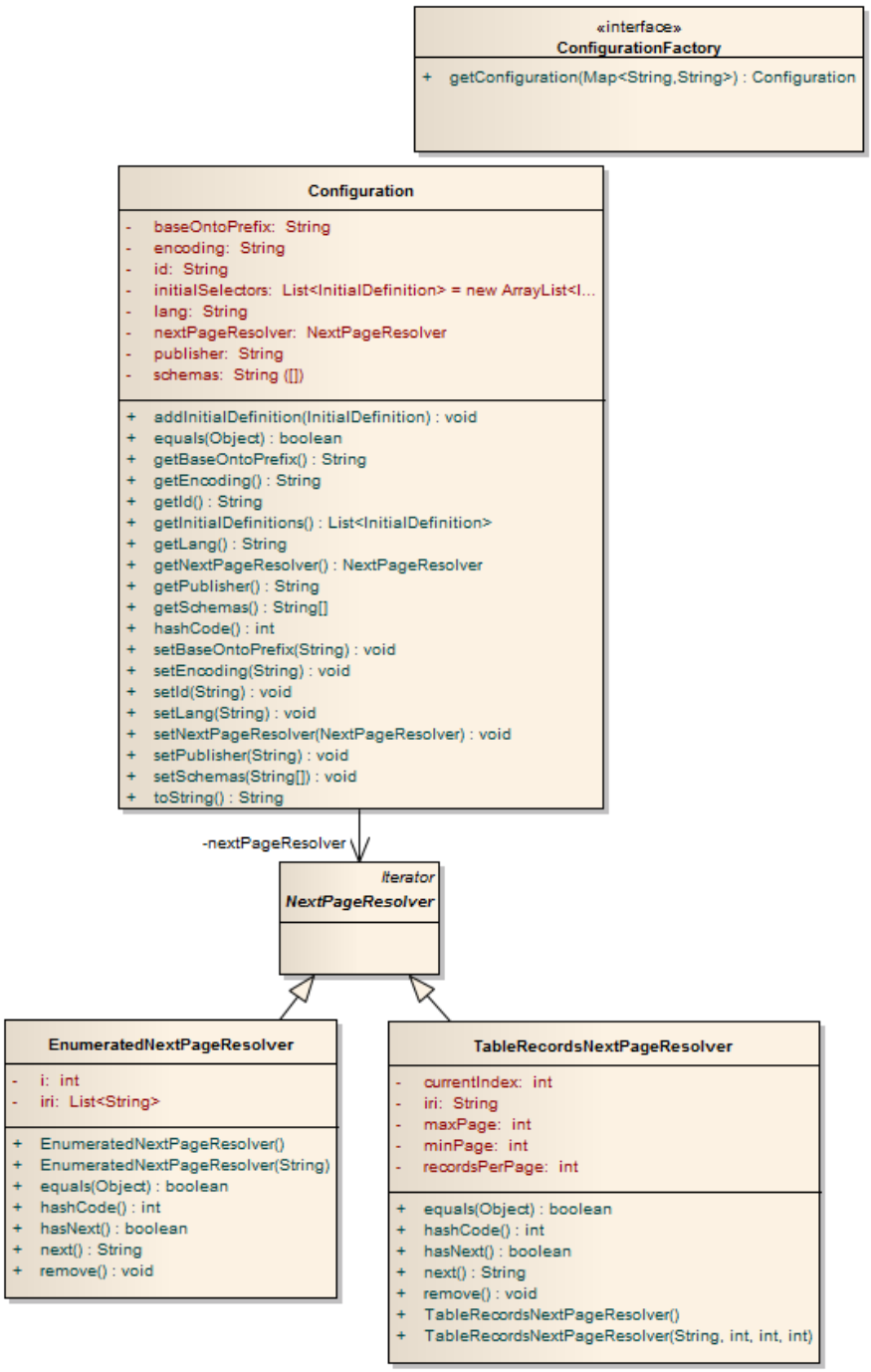
| resource | description |
|---|---|
| rdf:type | a property used to state that a resource is an instance of a class<br>a commonly accepted qname for this property is `a` |
| rdfs:Resource | the class of everything; all things described by RDF are resources |
| rdfs:Class | declares a resource as a class for other resources |
| rdfs:Literal | literal values such as strings and integers<br>property values such as textual strings are examples of RDF literals<br>literals may be plain or typed |
| rdfs:Datatype | the class of datatypes<br>rdfs:Datatype is both an instance of and a subclass of rdfs:Clas<br>each instance of rdfs:Datatype is a subclass of rdfs:Literal |
| rdf:XMLLiteral | the class of XML literal values; rdf:XMLLiteral is an instance<br>of rdfs:Datatype (and thus a subclass of rdfs:Literal) |
| rdf:Property | the class of properties |
| rdfs:domain | (of an rdf:predicate) declares the class of the subject in a triple<br>whose second component is the predicate |
| rdfs:range | (of an rdf:predicate) declares the class or datatype of the object in a triple<br>whose second component is the predicate |
| rdfs:subClassOf | allows to declare hierarchies of classes |
| rdfs:subPropertyOf | an instance of rdf:Property that is used to state<br>that all resources related by one property are also related by another |
| rdfs:label | rdf:Property used to provide a human-readable version of a resource's name |
| rdfs:comment | rdf:Property used to provide a human-readable description of a resource |

**Table C.1.** RDF and RDFS vocabulary

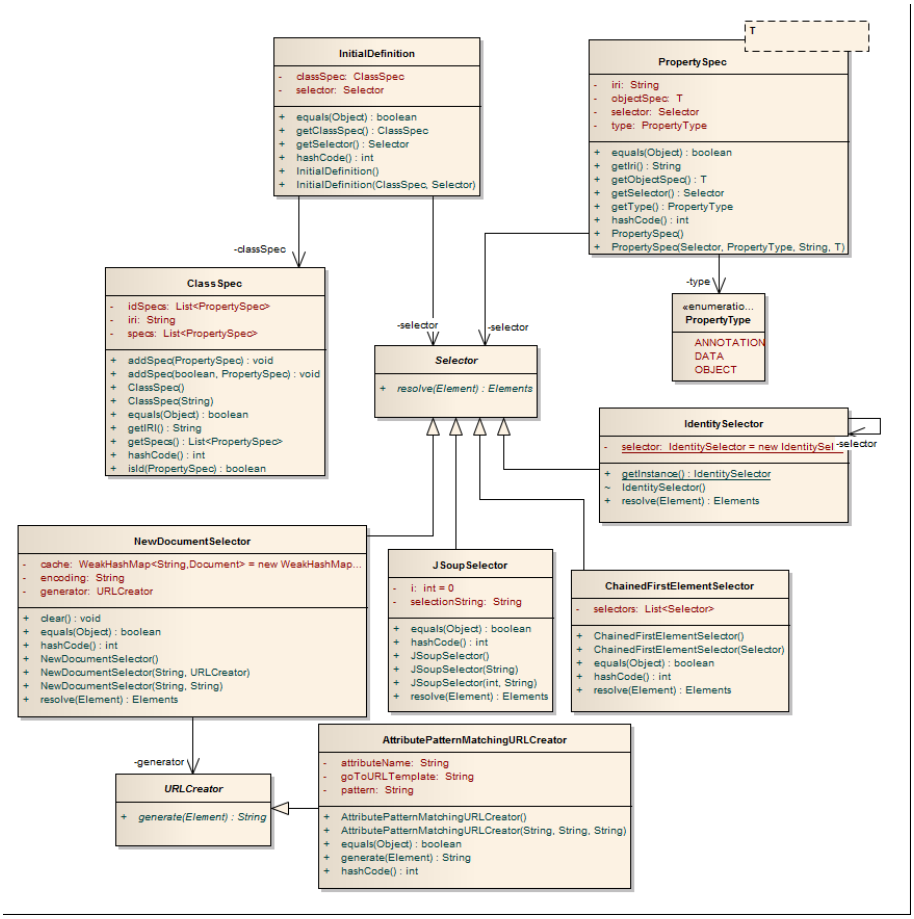
# Appendix D
# Example of RDF/XML syntax

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmlns:vs="http://www.w3.org/2003/06/sw-vocab-status/ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
  <!-- Here we describe general characteristics
       of the FOAF vocabulary ('ontology'). -->
  <owl:Ontology rdf:about="http://xmlns.com/foaf/0.1/"
                dc:title="Friend of a Friend (FOAF) vocabulary"
                dc:description="The Friend of a Friend (FOAF) RDF
                                vocabulary, described using
                                W3C RDF Schema and OWL the Web
                                Ontology Language." >
  </owl:Ontology>
  <rdfs:Class rdf:about="http://xmlns.com/foaf/0.1/Person"
              rdfs:label="Person"
              rdfs:comment="A person."
              vs:term_status="stable">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
    <owl:equivalentClass
      rdf:resource="http://schema.org/Person" />
    <owl:equivalentClass
      rdf:resource="http://www.w3.org/2000/10/swap/pim/contact#Person"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="http://xmlns.com/foaf/0.1/Agent"/>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Class
        rdf:about="http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing"
        rdfs:label="Spatial Thing"/>
    </rdfs:subClassOf>
    <rdfs:isDefinedBy
      rdf:resource="http://xmlns.com/foaf/0.1/"/>
    <owl:disjointWith
      rdf:resource="http://xmlns.com/foaf/0.1/Organization"/>
    <owl:disjointWith
      rdf:resource="http://xmlns.com/foaf/0.1/Project"/>
  </rdfs:Class>
  <!-- (...) -->
</rdf:RDF>
```

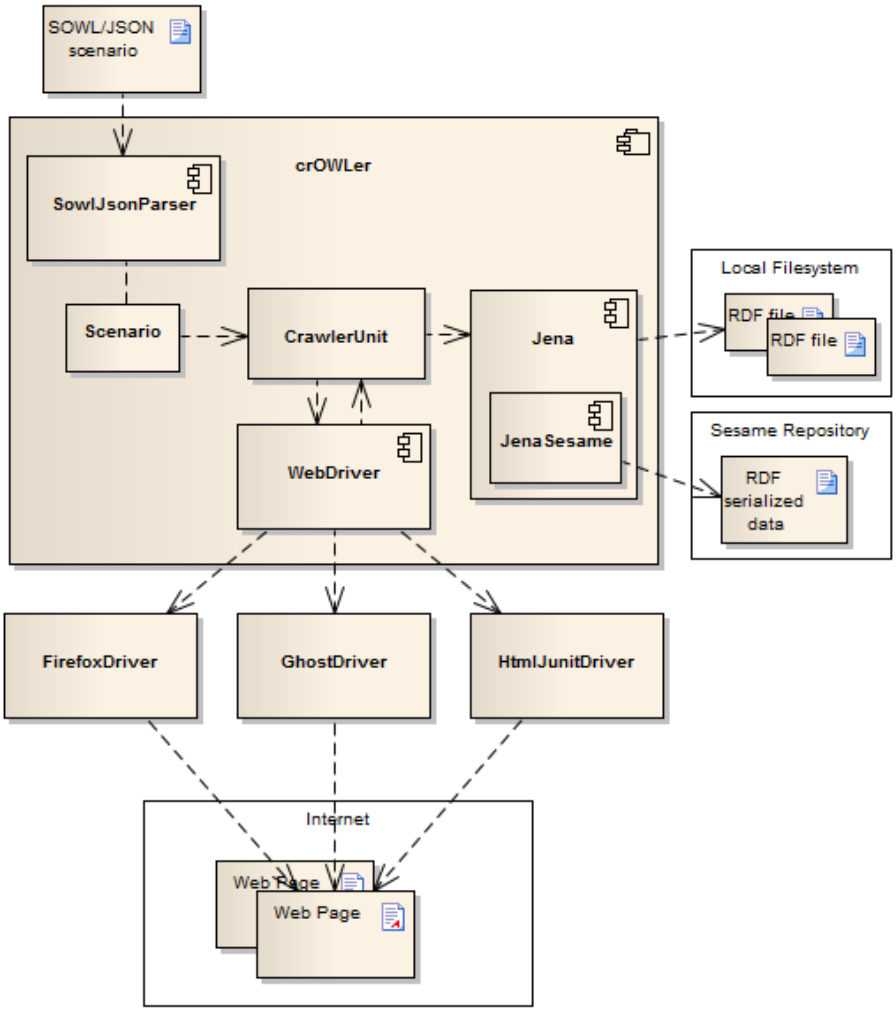
# Appendix E
# Configuration component of original crOWLer



# Appendix F
## Selector component of original crOWLer



# Appendix G
## crOWLer architecture



# Appendix H
# Detailed architecture of Strigil platform

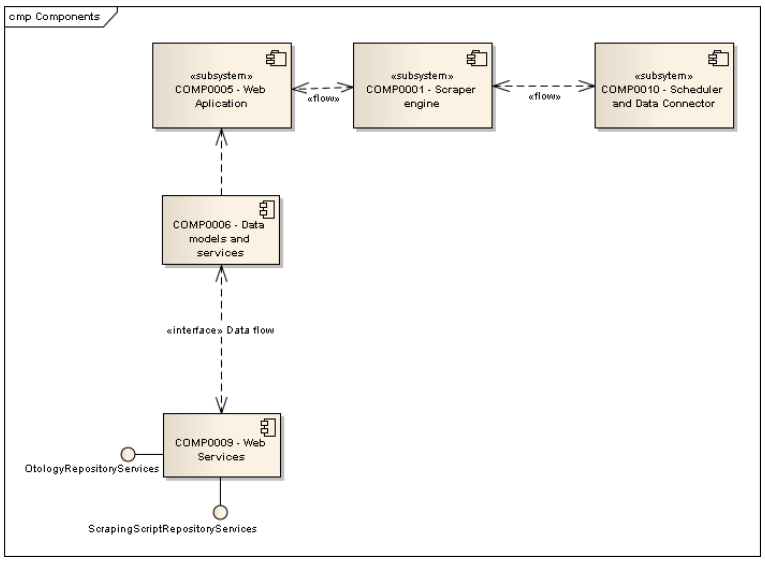



**Figure H.1.** Components of Data Application part of Strigil

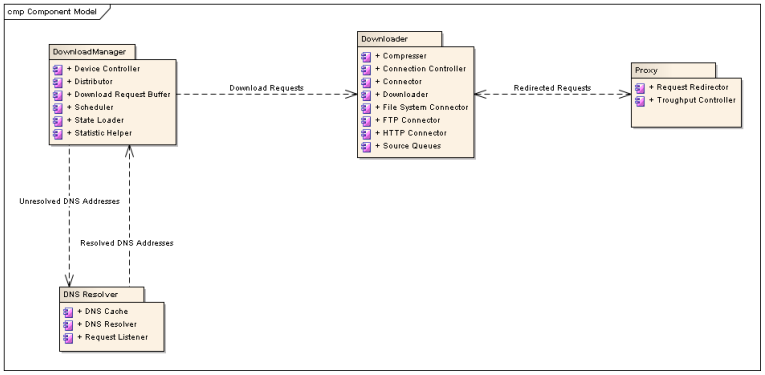



**Figure H.2.** Components of Download System part of Strigil

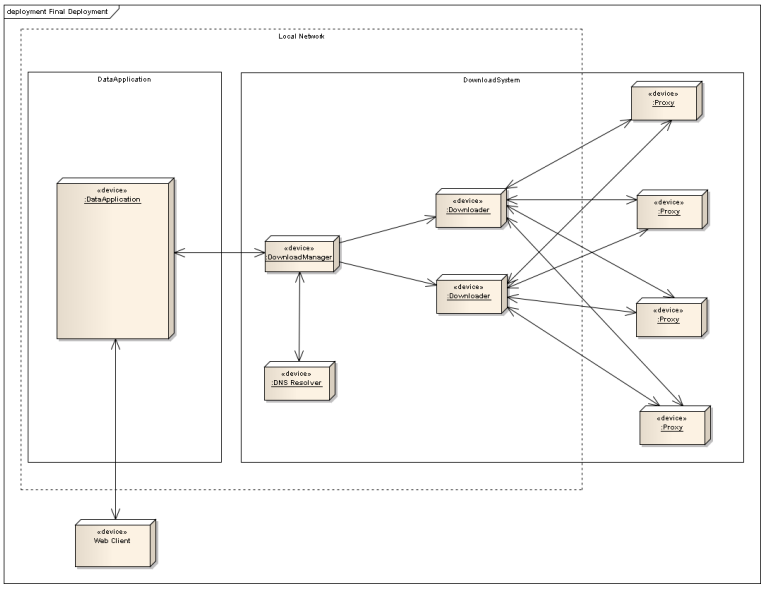



**Figure H.3.** Example deployment structure of Strigil

# Appendix I
# SOWL/JSON scenario solving Use Case 1

```
{
  type: "scenario",
  name: "manual",
  ontology: {
    base: "http://kub1x.org/onto/dip/t/",
    imports : [
      {
        prefix: "foaf",
        uri: "http://xmlns.com/foaf/0.1/",
      },
      {
        prefix: "kbx",
        uri: "http://kub1x.org/onto/dip/t/",
      },
    ],
  },
  creation-date: "2014-11-30 12:40",
  call-template: {
    command: "call-template",
    name: "init",
    url: "http://www.inventati.org/kub1x/t/",
  },
  templates: [
    {
      name: "init",
      steps: [
        {
          command: "onto-elem",
          typeof: "http://xmlns.com/foaf/0.1/Person",
          selector: {
            value: "tr",
            type: "css",
          },
          steps: [
            {
              command: "value-of",
              property: "http://xmlns.com/foaf/0.1/firstName",
              selector: {
                value: "td:nth-child(1)",
                type: "css",
              },
            },
            {
              command: "value-of",
```

```
              property: "http://xmlns.com/foaf/0.1/lastName",
              selector: {
                value: "td:nth-child(2)",
                type:  "css",
              },
            },
            {
              command: "value-of",
              property: "http://xmlns.com/foaf/0.1/phone",
              selector: {
                value: "td:nth-child(3)",
                type:  "css",
              },
            },
            {
              command: "call-template",
              name: "detail",
              selector: {
                value: [
                  {
                    value: "td.detail a",
                    type:  "css",
                  },
                  {
                    value: "@href",
                    type: "xpath",
                  },
                ],
                type: "chained",
              },
            },
          ],
        },
      ],
    },
    {
      name: "detail",
      steps: [
        {
          command: "value-of",
          property: "http://xmlns.com/foaf/0.1/nickname",
          selector: {
            value: ".nick",
            type:  "css",
          },
        },
      ],
    },
  ],
}
```