UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Enhancement of a software chrestomathy for open linked data

Masterarbeit

zur Erlangung des Grades eines Master of Science
vorgelegt von

Martin Leinberger

Erstgutachter:	Prof. Dr. R. Lämmel Institut für Softwaretechnik
Zweitgutachter:	M. Sc. A. Varanovich Institut für Softwaretechnik

Koblenz, im Juli 2013

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

....

(Ort, Datum) (Unterschrift)

Abstract

A software chrestomathy collects small software systems as an aid in learning a subject. As many languages, technologies and concepts might be involved, it will contain a heterogeneous mix of code artifacts, documentation and relationships. This is problematic, as knowledge resources should convey their data in a structured manner. The data should be conveniently explorable, easily discoverable and as accessible for humans as well as machines.

This thesis tries to tackle the problems created by the heterogeneousness of the data by applying Linked Data principles. The 101companies chrestomathy is enriched with these principles, meaning that every important entity is seen as a resource and dereferencable through HTTP, which results in meaningful data about this entity. Additionally, the entities are linked with each other, so that all available data is reachable.

It is shown that by embracing the Linked Data principles, the problems created by the diverse data sets can be alleviated. Furthermore, examples are given on how the approach can even enable further research options, such as clone detection, that were previously difficult if not unfeasible.

Acknowledgements

The 101companies system is a joint work of the Softlang Team at the University of Koblenz-Landau. At the time of writing, it relies on contributions of Kevin Klein, Aleksey Lashin, Ralf Lämmel, Arkadi Schmidt, Thomas Schmorleiz and Andrei Varanovich. This thesis would not have been possible without them.

Contents

List of Figures

Chapter 1

Introduction

1.1 101companies

The 101companies project[1] (also 101companies, 101project or just 101) [FLSV12] is a software chrestomathy. According to Wikipedia, a chrestomathy "is a collection of choice literary passages, used especially as an aid in learning a subject" [unk13]. Likewise, a software chrestomathy can be seen as a collection of small software systems, which 101 calls contributions. The intention behind is to provide the software engineering community a valuable new knowledge resource for learning and comparing languages, technologies and concepts [FLL$^+$12].

All contributions in the 101project deal with the same task - implementing a small Human-Relations system, usually dealing with a structure consisting of companies, departments and employees. Besides this basic task, all contributions choose from a predefined feature set which features they want to implement, ranging from basic requirements like being able to total or cut the salaries to more advanced ones like doing things in parallel.

Through its contributions, the 101project can actually highlight the various combinations of languages, technologies and concepts, making the comparison and learning based on known examples easier for users.

[1]http://101companies.org

1.2 The problem

As storing contributions and data for as many languages, technologies and concepts as possible is a necessity for a software chrestomathy, one can easily imagine it as a collection of highly heterogeneous code artifacts, documentation and relationships. This is problematic, as 101 is aimed at representing and conveying knowledge about these things in a structured manner. In particular, all code artifacts and documentation should be conveniently explorable, relationships and all available data should be discoverable. The data should also be consumable by humans as well as machines.

The thesis tries enrich the 101companies chrestomathy with a Linked Data approach to tackle these problems. By applying the principles as described by Tim Berners-Lee [BL07], the data supposedly becomes more structured and easier to consume, both from a human perspective as well as from the perspective of a machine operating over the data set.

A summary of the results presented in this thesis and co-authored by Kevin Klein, Ralf Lämmel, Thomas Schorleiz and Andrei Varanovich, has been submitted for publication [KLL$^+$13].

Chapter 2

Related work

2.1 Previous work within the 101companies project

Within the 101companies project, with and without participation of the present author, relevant work has been published. The concept of the 101companies chrestomathy was introduced in [FLSV12]. The idea of linking entities to resources like languages and technologies was presented in [FLV12] and extended to the automatic recovery of such links from source artifacts in [FLL$^+$12].

2.2 Related work inside a Linked Data context

Exposing of heterogeneous data through Linked Data is not a new problem. A related approach has been described by [KFH$^+$12] and [KFRC11]. There, Linked Data enabled software repositories are used to expose software artifacts as well as the results of preprocessing and analysis steps on these artifacts in the context of software repository mining.

Other research on applying Linked Data principles onto software repositories includes the linking and documentation of data in different repositories through RDF as described by [How08]. The goal is to overcome heterogeneous documentation techniques used by the different repositories, such as documenting in wikis or database schemas, in order to improve usability of the data. Another approach is the Linked Data Driven Software Development methodology as described by [IUHT09]. There, the goal is to transform "data from version control systems, bug tracking tools and source code into linked data". Linked Data has also been used to for enhancing tracebility by augmenting source code repositories with developer related information [IH12].

Other cases include vocabularies, like the Asset Description Metadata Schema (ADMS) [Dek13], used to describe "semantic assets, defined as highly reusable metadata and reference data" also exist. [Per11] took ADMS to the context of eGovernment to tackle the semantic interoperability of systems. Also, the Asset Description Metadata Schema for Software (ADMS.SW) [Goe11] exists, and was applied to the Debian Package Tracking system by [Ber12]. The goal was to "generate RDF meta-data documenting the source packages, their releases and links to other packaging artifacts" and to link the packages to other Open source software and derivative distributions for traceability. Linked Data principles were again applied to Open Source repositories by [ICH12], with the purpose of simplifying the integration of data multiple code forges.

2.3 Related work outside a Linked Data context

There is also relevant work outside of a Linked Data context. Especially as most data sources in the 101companies project are not in a Linked Data format by nature, but still need to be combined and exposed, the field of "mashing up" non Linked Data sources is highly relevant. The software architecture used here is related to general "mash up" architectures such as described by [MG08]. A architecturally similar approach has been applied to Software Engineering by [GTS10] with the goal of improving SE tools by enriching them with information from different sources such as web-based APIs and information repositories.

As the referencing of entities is one of the major problems tackled in the thesis, other identification mechanisms such as Uniform Resource Names (URN) [Moa97] and Digital Object Identifier (DOI) systems, as described by [Pas05], are also relevant. However, none of these identification approaches provide the integration into a global data space as Linked Data does and were therefore not considered any further.

Chapter 3

Background

3.1 101companies

As presented in the introduction the 101companies system stores contributions as well as their documentation. Additionally, it also stores data about the concepts, languages and technologies used in these contributions. In overview, the 101project is defined by three major systems [FLL$^+$12]:

– 101repo for storing code artifacts.

– 101wiki for storing 101-specific documentation.

– 101worker for creating derived resources containing metadata about 101repo artifacts.

3.1.1 101repo

As the contributions are small, self-running and independent software systems, it makes sense to store them in repositories. 101 uses the 101repo, a GitHub based distributed repository, to store all software artifacts related to the chrestomathy. The confederation is necessary to enable a smooth collaboration process, in which new contributions can be added easily without authors having to check out the complete repository. 101repo does not only store contribution data. As sometimes code artifacts are created for highlighting special concepts or "Hello World" programs for languages, it also has to store this data.

3.1.2 101wiki

The code artifacts in 101repo are documented in two ways. For one, they are treated as regular source code and should therefore be documented with regard to software engineering best practices. Code comments and Readme files should provide guidance on what the code does and how it does it. However, this documentation does not consider the 101-specific concerns, as it would be to disrupting to store this with the code artifacts. A wiki based approach is used for the 101-related documentation, which highlights the interesting code parts and integrates the contribution in the 101companies ecosystem.

3.1.3 101worker

The 101worker system is tasked with deriving knowledge about the code artifacts in 101repo. This is done in several preprocessing and analysis steps, such as fact extraction, tokenization, metrics computations or metadata association based on a rule set. The worker also tries to compute the links that exist between the code artifacts and the documentation. For example, a file might use a certain language or technology, introducing a link from that file to the documentation for the language or technology.

The worker system is completely based on file I/O, meaning that it will serialize results of every analysis step in files. These files then contain the valuable derived knowledge and are referred to as derived resources.

3.2 Linked Data

Linked Data can best be described as a "set of best practices for publishing and connecting structured data on the Web" [BHBL09]. The goal is not only to provide data, but also create typed links between the data of different, possibly very diverse, data sources. The interoperability of these systems is achieved by a machine-readable description language used to encode the informations. Through the adoption of these best practices, a global data space has been created already containing billions of assertions [BHBL09].

3.2.1 Principles

Linked Data is directly build on the Web architecture and applies this architecture to the task of sharing data [HB11]. As the web architecture is build around a few mechanisms, mainly the use of URIs as an identification mechanism, HTTP as an access mechanism and HTML as a content format, the Linked Data principles are fairly similar. These principles, defined by [BL07], that became known as the official Linked Data principles are:

1. Use URIs as names for things.

2. Use HTTP URIs so that people can look up those names.

3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL)

4. Include links to other URIs, so that they can discover more things

In general, Linked Data systems deal with the "items of interest" of a specific domain, which are called resources. These resources are identified through an URI, as these provide the advantage to create globally unique identifiers in a decentralized fashion [HB11]. As HTTP is a well known form of access, HTTP URIs should be used as identifiers. This allows users to directly access the resource, which is called dereferencing. The user then obtains a description of the resource, which is modeled with RDF. The last principle adds links to other data sets. This is important as it connects the individual data sets with each other to create one global data space. In contrast to many traditional web services, who may also use standardized descriptions and identification by an URI, the Linked Data approach is the only one who can provide typed links into other data sets and therefore break up the traditional data silos.

These standards are elemental, when taking several data sources and combining them into a new information sources. This is generally called a "mashup".

3.2.2 Data model

As mentioned in the previous section, the data model behind Linked Data is RDF (Resource Description Framework) [KC04]. It represents information as an node-and-arc-labeled directed graph [HB11]. Figure 3.1 shows an example for such a graph. In this example, it is described that a resource with the name of "Joe Hackaton" knows another resource called "Olga Subbotnik".

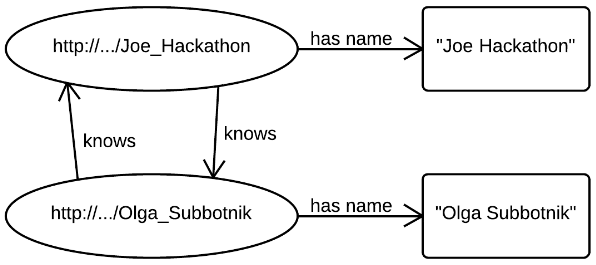


Figure 3.1: Simple RDF graph example.

These RDF descriptions are represented as triples, consisting of a subject, predicate and object, making basic assertions about a resource. Subjects are always resources and therefore URIs. Objects can either be literal values, like strings and numbers, or they can be links to other resources. Predicates express the type of relationship that exists between subject and object. They are basically URIs pointing to their definition in vocabularies. Figure 3.2 shows the same graph as figure 3.1, just this time in triple form.

```
ex:Joe_Hackathon   ex:has_name  "Joe Hackathon"
ex:Joe_Hackathon   ex:knows     ex:Olga_Subbotnik
ex:Olga_Subbotnik  ex:has_name  "Olga Subbotnik"
ex:Olga_Subbotnik  ex:knows     ex:Joe_Hackathon
```

Figure 3.2: Examples of RDF Triples (URIs replaced by "ex").

Vocabularies can be expressed in RDF schema (RDFS). This is a "declarative, machine-processable language", that "can be used to formally describe an ontology or metadata schema as a set of classes (resource types) and their properties" [Jac03]. It is also used to specify relations between classes and properties as well as to specify some constraints on these properties. It supports basic inheritance through the "subClassOf" predicate, while "domain" can be used to state the class of the subject. The predicate "range" is used to state the "object" class of a property. Figure 3.3 shows an example of RDFS statements that fit the previous examples. It is important to notice that RDFS is a relatively simple ontology language, meaning that it can define the right usage of a predicate, but it has no advanced restrictions like cardinality.

```
ex:Person     a          rdfs:Class

ex:has_name   a          rdfs:Property
ex:has_name   rdfs:domain  ex:Person
ex:has_name   rdfs:range   rdfs:Literal
```

Figure 3.3: Examples for RDFS statements.

RDF is just a data model and can be serialized in several ways [HB11]. One of the most common formats, that is also used in this thesis, is RDF/XML [BM03], where the RDF statements are serialized as XML. A central root node exists, while every resource described in this language is a child of this root node. The children of every resource node make up the stated triples for this resource. Figure 3.4 shows the previously introduced example in such a serialization, while 3.5 shows the RDFS for the example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ex="...">

  <rdf:Description rdf:about=".../Joe_Hackathon">
    <rdf:type rdf:resource="...#Person"/>
    <ex:has_name>Joe Hackathon</ex:has_name>
    <ex:knows rdf:resource=".../Olga_Subbotnik"/>
  </rdf:Description>

  <rdf:Description rdf:about="../Olga_Subbotnik">
    <rdf:type rdf:resource="...#Person"/>
    <ex:has_name>Olga Subbotnik</ex:has_name>
    <ex:knows rdf:resource=".../Joe_Hackathon"/>
  </rdf:Description>
</rdf:RDF>
```

Figure 3.4: Example RDF/XML document.

All RDF data and schemas shown in this thesis are displayed in their serialized RDF/XML form.

```
<rdf:RDF xmlns:rdf="..." xmlns:rdfs="...">
  <rdfs:Class rdf:ID="...#Person"/>

  <rdf:Property rdf:about="...#has_name">
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema
        #Literal"/>
    <rdfs:domain rdf:resource="...#Person"/>
  </rdf:Property>

    <!--- ... other properties omitted ... -->
</rdf:RDF>
```

Figure 3.5: Example RDFS serialized in RDF/XML.

3.3 JSON schema

The second schema language used in this thesis is JSON schema. This is a "JSON based format for defining the structure of JSON data" [GZC13]. It is a full fledged schema language capable of, among other things, defining required fields and datatypes for fields as well as restricting values for these fields. Figure 3.6 shows a short example of a schema for a person. It shows that the JSON data is an object, having the properties "name", "age" and "website". The "website" property has the restriction that it needs to be a string in an URI like format. Only "name" and "age" are required in this object, the "website" property can be missing.

```
{
  "title": "Person schema",
  "type" : "object",
  "required": ["name", "age"],
  "properties": {
    "name" : { "type": "string" },
    "age" : { "type" : "integer"},
    "website" : { "type": "string", "format" : "uri" }
    }

}
```

Figure 3.6: Example for a JSON schema.

Chapter 4

Linked Data requirements

4.1 Linked Data principles and 101companies

The basic idea of this thesis is to tackle the problems created by the diversity of the data and the heterogeneity of the systems through the Linked Data principles described in section 3.2.1. Applying the principals to the 101companies project implies that entities in 101, which means all wiki pages, all code artifacts, all derived resources and all ontological entities, must be referable through HTTP URIs. Dereferencing those must reveal meaningful data - this includes the actual content of the entity (if applicable) as well as available metadata. It should be possible to display all data in the machine readable RDF data format. Also, everything should be as interlinked as possible meaning that every entity should provide links to all other, for this entity relevant entities, in the chrestomathy.

4.2 Further requirements

As the description in the previous section is rather general and hard to validate, more specialized requirements, that can be evaluated later, are formulated in the next sections.

4.2.1 Navigate from wiki to repo

As everything has to be interlinked, it must be possible to navigate from the wiki to the repository to get from the documentation to the actual source code. This is complicated by the fact that the repository is distributed, meaning that the distribution mechanism must be exploited to actually create the links to the physical repositories.

4.2.2 Navigate from repo to wiki

The navigation should be bidirectional, meaning that it should also be possible to navigate from the files and folders of the repository to associated wiki pages. As the physical repositories are not under control of 101companies, a view that can act as an replacement for direct repository access shall be used.

4.2.3 Referencing source code on wiki pages

As documentation on the wiki has to highlight source code from 101repo, it will refer to certain parts of the code. The unambiguous identifier through URIs, as required by Linked Data principles, shall be exploited so that wiki pages can reference source code parts. The wiki can then dereference the URI and directly display the source code on the wiki page.

4.2.4 Displaying of metadata for 101repo entities

As the Linked Data principles require meaningful data when dereferencing an URI, the resources in 101repo shall be extended with metadata derived through 101worker. A example for this meaningful metadata is the language a file uses.

4.2.5 Associate derived resources with primary resources

The derived resources that 101worker creates can be difficult to use, as it is hard to discover what data exists and where it can be found. Therefore, in accordance with the Linked Data principle of providing links to all other relevant resources, derived resources should be linked to their source code artifacts. Additionally, it should be linked how the metadata was derived. Thus:

– For every source code artifact, link to the derived metadata.

– For the derived metadata, link to the source code artifact which it was taken from.

– For derived metadata, link who produced it.

4.2.6 Operate the wiki like a graph

Conceptually, 101wiki is a graph. In this graph, pages are represented as nodes while the links between them are the edges. As one goal of Linked Data is to provide typed links, these edges should have types associated with them.

4.2.7 Operate the repo like a tree

Conceptually, 101repo is a tree. As everything should be linked, a User should be able to browse through the repository tree, transitioning from folders to files and back to its parent folder. Again, as 101repo is a distributed repository and not completely under control of 101companies, a View shall be used to enable this experience.

4.2.8 Querying for data

The Linked Data principles as described in section 3.2.1 also encourage the use of SPARQL. This extends the possibilities of browsing and allows a more precise way of information gathering. Therefore, 101wiki and 101repo should both be queryable through SPARQL.

4.2.9 Human and machine readable data

Technically, the Linked Data principles only require data in a RDF format. But as 101 is primarily designed as a knowledge source, human interaction is expected. Therefore, all data should also be exposed as HTML. For 101repo, a View shall be used for this. The HTML representation of 101wiki data is the wiki itself. Additionally, RDF/XML and JSON shall be provided for machine consumption.

Chapter 5

Data modeling

5.1 101repo data

As all code artifacts in the 101companies project are stored in 101repo, this section takes a look at the distribution system and the data model behind 101repo.

5.1.1 Mounting of repositories

101repo is a virtual repository consisting of many physical GitHub repositories, who are actually storing the files. This is enabled by a 101 specific mechanism comparable to the Linux file system table (fstab). A registry of contributing repositories exists. Every entry of this registry has additional information that defines how the data in these repositories is mounted in the 101repo tree. This registry is maintained by submission and administration services.

```
registry ::= entry* .
entry ::= repository sourceDirectory mountDirectory [ options ] .

repository ::= URL .
sourceDirectory ::= relativePath .
mountDirectory ::= relativePath .
options ::= ("normal" | "subdirsonly") .
```

Figure 5.1: Registry for mounting repositories in 101repo.

5.1.2 Data model behind the stored data

Even though the repository is in reality confederated, it has a virtual folder layout that resembles a tree. Figure 5.2 shows the data model behind 101repo as a UML diagram. 101 uses a namespace structure for its data. The highest possible namespace is the namespace called "Namespace", which acts as the root of the 101repo tree. All other namespaces, like the "Language" and "Contribution" namespaces are members of this namespace. They themselves have members again, like the language "Java" in the "Language" namespace. A special form of members are Modules, which are detailed further in section 5.3.3. Although namespaces are represented as folders on the file system, only folders resembling the members of this namespace are expected in them. Members on the other hand are considered to be real folders, which can have sub-folders and files. Files break into smaller units, so called fragments. These fragments can be composed of smaller fragments again.

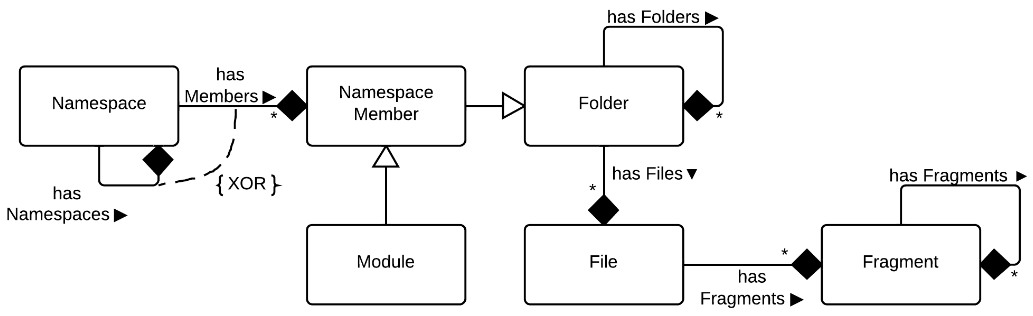


Figure 5.2: Data model for 101repo.

Every entity in this data model has metadata assigned to it. As most metadata is available for files, the schema for file entities[1] is shown in figure 5.3. The metadata is often directly inferred from the entity itself. An example is the language of a file. But some metadata, like a wiki headline that only namespaces and namespace members have, is passed along by higher entity. Other associated metadata includes the links to the physical GitHub repository and links to derived resources. Also, the content for a file is returned.

Figure 5.4 shows an excerpt of the RDFS schema for RDF data available on 101repo entities, which defines the different types of resources[2] and the metadata

[1]For brevity, the definition which are required is omitted.
[2]These are the same as in the UML diagram in figure 5.2.

that every resource can have.

```
{
  "title": "File schema",
  "type" : "object",
  "properties": {
    "wiki" : { "type": "string", "format": "uri" },
    "github" : { "type": "string", "format": "uri" },
    "name" : { "type": "string" },
    "headline" : { "type": "string" },
    "triplestore" : { "type": "string", "format": "uri" },
    "classifier" : { "type": "string", "enum": ["File"] },
        "language": { "type" : "string" },
        "content" : { "type" : "string" },
        "derived" : {
            "type" : "array",
            "items": {
                "type" : "object",
                "properties" : {
                    "headline" : {"type":["string", "null"]},
                    "resource" : { "type" : "string",
                      "format" : "uri" },
                    "name" : { "type" : "string" },
                    "producedBy" : { "type" : "string",
                      "format":"uri" }
                }
          }},
    "fragments" : {
      "type": "array",
      "items": {
        "type": "object",
        "required" : ["resource", "classifier", "name"],
        "properties": {
          "resource" : { "type": "string", "format" : "uri"},
          "classifier" : {
            "type": "string", "enum" : [ "Fragment" ]
          },
          "name" : { "type": "string" }
        }
      }}}
}
```

Figure 5.3: JSON schema for file entities in 101repo.

```
<rdf:RDF xmlns:rdf="..." xmlns:rdfs="...">
  <rdfs:Class rdf:ID="http://101companies.org/schemas/repo#
      Namespace"/>

  <rdfs:Class rdf:ID="http://101companies.org/schemas/repo#
      NamespaceMember">
    <rdfs:subClassOf rdf:resource="http://101companies.org/schemas
        /repo#Folder"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="http://101companies.org/schemas/repo#Folder
      "/>

  <rdfs:Class rdf:ID="http://101companies.org/schemas/repo#File"/
      >

  <rdfs:Class rdf:ID="http://101companies.org/schemas/repo#
      Fragment"/>

  <rdfs:Class rdf:ID="http://101companies.org/schemas/repo#Module
      ">
    <rdfs:subClassOf rdf:resource="http://101companies.org/schemas
        /repo#Member"/>
  </rdfs:Class>

  <rdf:Property rdf:about="http://101companies.org/schemas/repo#
      headline">
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema
        #Literal"/>
    <rdfs:label>Headline</rdfs:label>
    <rdfs:comment>The headline that is used on the wiki</
        rdfs:comment>

    <rdfs:domain rdf:resource="http://101companies.org/schemas/
        repo#Namespace"/>
    <rdfs:domain rdf:resource="http://101companies.org/schemas/
        repo#Member"/>
    <rdfs:domain rdf:resource="http://101companies.org/schemas/
        repo#Folder"/>
    <rdfs:domain rdf:resource="http://101companies.org/schemas/
        repo#File"/>
    <rdfs:domain rdf:resource="http://101companies.org/schemas/
        repo#Fragment"/>
  </rdf:Property>
    <!--- ... other properties omitted ... -->
</rdf:RDF>
```

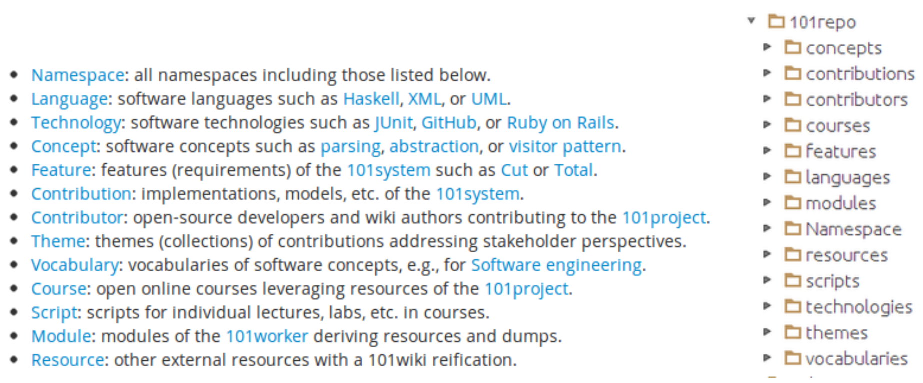
Figure 5.4: RDF schema for metadata in 101repo (excerpt).

5.2 101wiki data

As all 101-specific documentations is stored in 101wiki, this section takes a closer look at it. Generally, 101wiki is a semantic wiki, consisting of wiki pages, which in turn break down into URI addressable sections.

5.2.1 Relation between 101wiki and 101repo

Just like 101repo, 101wiki uses a namespace categorization for its content. This namespace-based organization of the 101repo layout and the 101wiki are designed to be in sync. The top level folders of 101repo correspond to the namespaces of 101wiki, while the second-level folders correspond to the member pages on the wiki. Figure 5.5 shows some top level namespaces of 101wiki and their associated folders in 101repo. Beyond the namespace member level, files and folders are associated with the member page.



- Namespace: all namespaces including those listed below.
- Language: software languages such as Haskell, XML, or UML.
- Technology: software technologies such as JUnit, GitHub, or Ruby on Rails.
- Concept: software concepts such as parsing, abstraction, or visitor pattern.
- Feature: features (requirements) of the 101system such as Cut or Total.
- Contribution: implementations, models, etc. of the 101system.
- Contributor: open-source developers and wiki authors contributing to the 101project.
- Theme: themes (collections) of contributions addressing stakeholder perspectives.
- Vocabulary: vocabularies of software concepts, e.g., for Software engineering.
- Course: open online courses leveraging resources of the 101project.
- Script: scripts for individual lectures, labs, etc. in courses.
- Module: modules of the 101worker deriving resources and dumps.
- Resource: other external resources with a 101wiki reification.

```
▾ 101repo
  ▸ concepts
  ▸ contributions
  ▸ contributors
  ▸ courses
  ▸ features
  ▸ languages
  ▸ modules
  ▸ Namespace
  ▸ resources
  ▸ scripts
  ▸ technologies
  ▸ themes
  ▸ vocabularies
```

Figure 5.5: Wiki namespaces (left) and folders in 101repo (right).

5.2.2 Links in 101wiki

The pages stored in the wiki can refer to each other via plain links or more specific semantic properties. Semantic properties, which are essentially RDF triples declared on the page, are used for creating typed links between the pages. Through these, special relationships between the pages are expressed. An example is the "uses" property, which links a contribution to languages or technologies, indicating that a contribution uses them in its source code. Figure 5.6 gives an overview over all semantic properties linking to pages inside the wiki.

Predicate	**Meaning**
uses	A resource uses a language or technology.
implements	A contribution implements a feature.
instanceOf	"instance of" relationship.
isA	"is-a"" relationship on concepts.
developedBy	A contribution is developed by a contributor.
reviewedBy	A contribution is reviewed by a contributor.
relatesTo	A resource relates to (ontological) to another resource.
mentions	A resource mentions another resource (weak internal link).

Figure 5.6: Properties for typed links internal to the wiki.

In accordance with Linked Data practices, the wiki should also offer links outside the own dataset. Similar to internal links, semantic properties are used for that. The two available predicates are "identifies" and "linksTo". While "identifies" is used for external resources that are ontologically about the same resource, "linksTo" expresses that the external resource is highly relevant to the wiki page, but they cannot considered to be the same. Figure 5.7 lists the external properties. Figure 5.8 shows an excerpt of the schema used for 101wiki.

Predicate	**Meaning**
identifies	External resource is designated to the resource at hand.
linksTo	External resource is concerned with the resource at hand.

Figure 5.7: Properties (types) for typed links to external resources.

```
<rdf:RDF xmlns:rdf="..." xmlns:rdfs="...">

  <rdfs:Class rdf:ID="http://101companies.org/schemas/wiki#
      Namespace"/>
  <rdfs:Class rdf:ID="http://101companies.org/schemas/wiki#
      Language"/>
  <rdfs:Class rdf:ID="http://101companies.org/schemas/wiki#
      Technology"/>
  <!-- Further classes (namespaces) omitted. -->

  <rdf:Property rdf:about="http://101companies.org/schemas/wiki#
      uses">
    <rdfs:range rdf:resource="http://101companies.org/schemas/wiki
        #Technology"/>
    <rdfs:range rdf:resource="http://101companies.org/schemas/wiki
        #Language"/>
    <rdfs:domain rdf:resource="http://101companies.org/schemas/
        wiki#Contribution"/>
  </rdf:Property>

  <rdf:Property rdf:about="http://101companies.org/schemas/wiki#
      implements">
    <rdfs:range rdf:resource="http://101companies.org/schemas/wiki
        #Feature"/>
    <rdfs:domain rdf:resource="http://101companies.org/schemas/
        wiki#Contribution"/>
  </rdf:Property>

  <!-- Further properties omitted. -->

</rdf:RDF>
```

Figure 5.8: RDF schema of 101wiki.

5.3 101worker data

As all derived resources are created by 101worker, this section takes a look at how they are created and associated with their source artifacts.

5.3.1 Metadata derivation for code artifacts

The metadata and derived resources, that are linked with the entities as described in section 5.1.2, are derived through various means. Besides relatively simple processes, such as the calculation of metrics, more complex systems like the language

101meta are used. This language is basically a rule set, defining when to assign metadata to an entity.

It consists of rules, which break down into a precondition and a conclusion. Should the precondition be fulfilled, the conclusion is assigned. The preconditions have constraints of various nature. Typical example are the suffix of a source code artifact, which could lead to the assignment of a language or the content which could lead to the assignment of a technology. It is also possible to execute an arbitrary command (called predicate) to determine if a constraint holds. Figure 5.9 shows an overview over the rules and precondition syntax.

```
RuleSet ::= Rule⋆.
Rule ::= Condition → Assignment⋆.
Condition ::= Constraint | ¬ Constraint |
  Constraint ∧ Constraint | ... .
Constraint ::=
 suffix String
 | ...
 | content (String | RegExp)
 | predicate Command .
```

Figure 5.9: Rules and preconditions (excerpt) in 101meta.

If the complete precondition holds, then the metadata associated with the conclusion is assigned. Assignable metadata contains information that represents links to other 101 entities, like languages or technologies, but also attributes like the relevance of a file or support information like the syntax highlighting. Figure 5.10 shows an overview over the assignments in 101meta.

5.3.2 Enabling of fragments

A important step in 101meta is the assignment of the language specific fact extractors and fragment locators. These programs are necessary to enable the usage of fragments in 101repo - however, due to their language specific nature, they need to be assigned to the files first. As described in section 5.1.2, files break down into a fragment tree. These are extracted as part of a fact extraction phase, after the extractor has been assigned. Figure 5.11 shows a UML diagram of the available information as gathered through fact extraction.

When referring to a fragment, a specific fragment description is used. This can be seen as a simplified form of XPath queries against the fragment tree. They

```
Assignment ::=
  LinkAssignment
  | AttributeAssignment
  | MethodAssignment .
LinkAssignment ::=
  language LanguageName // equal to wiki predicate "use"
  | dependsOn TechnologyName // equal to wiki predicate "use"
  | concept ConceptName
  | feature FeatureName // equal to wiki predicate "implements"
  ... .
AttributeAssignment ::=
  relevance // relevance of file in system context
  | geshi // used for syntax highlighting
  ... .
MethodAssignment ::=
  extractor Command // Fact extractor assignment
  | locator Command // Fragment locator assignment
  ... .
```

Figure 5.10: Assignments (excerpt) in 101meta.

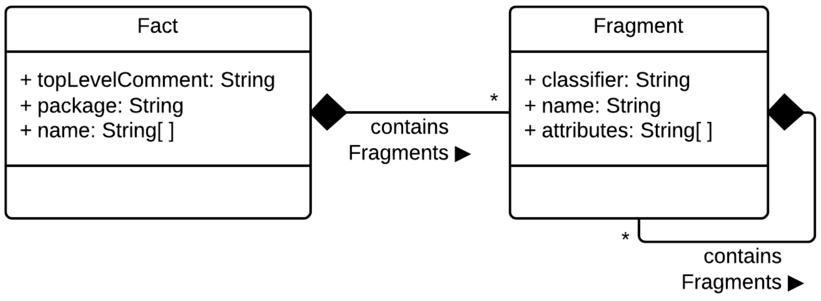


Figure 5.11: Model for facts.

are URI like selectors, stating the type and name of a fragment while traversing the fragment tree. Figure 5.12 shows a grammar for these descriptions. They are evaluated by fragment locators, which return the line range of the fragment in the file.

5.3.3 Creation of derived resources

The analysis processes on 101worker are implemented by so called modules, which serialize their results into files, making them the derived resources. There are basically three different ways a module can serialize results - based on files,

```
FragmentQuery ::= Type "/" Name [ "/" Index ] [ "/" FragmentQuery ] .

Index ::= Number .
Name ::= String .
Type ::= (JavaType | HaskellType | PythonType | ... ) .

JavaType ::= ("class" | "method") .
HaskellType ::= ("pattern" | "type" | "data" | ...) .
...
```

Figure 5.12: Query language for fragment locators.

folders or in big dumps. If a module produces derived resources based on files, it means that for every source artifact in 101repo, there will be a derived resource produced by this module. These derived resources are linked to their artifact through the filename as every module that produces resources in this manner uses an own extension. As many modules exist that work on individual files, every artifact f in 101repo will have a number of derivatives as described in figure 5.13.

Derived file for source file	**Contains**
f.commitInfo.json	Information about GitHub commits.
f.extractor.json	Results of fact extraction.
f.matches.json	Simple metadata units.
f.predicates.json	More advanced metadata units.
f.metrics.json	Metrics (lines of code, etc.) about a file.
f.tokens.json	Results of the tokenization process.
f.validator.json	Results of a validation process.
f.summary.json	Summary of previously computet metadata.

Figure 5.13: Derived files for every source artifact in 101repo.

Modules can also create resources based on folders. Some restrictions, like only producing resources for specific folders, are possible. Figure 5.14 lists all folder-based resources that currently exist.

Derived resource	**Restriction**	**Contains**
index.json	None	Gives overview over file-based metadata.
fileindex.json	None	Lists all files of folder.
members.json	namespace and members	Lists namespaces and members in folder.

Figure 5.14: Derived files for folders in 101repo.

The third kind of derived resources are large dumps of data. These are used to store aggregated results or larger data sets. Figure 5.15 lists several relevant dumps.

Dump	**Contains**
wiki.json	Serialized form of 101wiki and its links.
repository.json	Information about 101repo.
matches.json	Aggregated f.matches.json files.
predicates.json	Aggregated f.predicates.json files.
rules.json	Aggregated 101meta rules.
moduleDescriptions.json	Summary of module descriptions.
explorer.rdf	Summary of 101repo in a Linked Data format.

Figure 5.15: Derived dumps.

As all derived resources need to be linked to their producing module and source file or folder, this information needs to be captured. Module descriptions are used to describe what module created which derived resource, either by stating the suffix or a filename. Also, the language used for serialization needs to be specified. Figure 5.16 shows the schema for these descriptions.

```
{
   "title": "Module description schema",
   "type" : "object",
   "required": [ "tagets" ],
   "properties": {
      "targets" : {
         "type" : "array",
         "items": {
            "type": "object",
            "required": ["headline", "scope", "language"],
            "properties": {
               "headline" : {
                  "type": "string"
               },
               "scope" : {
                  "type": "string",
                  "enum": ["file", "folder", "dump"]
               },
               "suffix" : {
                  "type": "string"
               },
               "filename": {
                  "type": "string"
               },
               "language" : {
                  "type": "string",
                  "enum": ["JSON", "RDF"]
               }
            }
         }
      }
   }
}
```

Figure 5.16: JSON schema for module descriptions.

Chapter 6

Implementation

6.1 101wiki

On the technical side, 101wiki relies on the interaction of several components arranged in a three layer architecture as displayed in Figure 6.1.

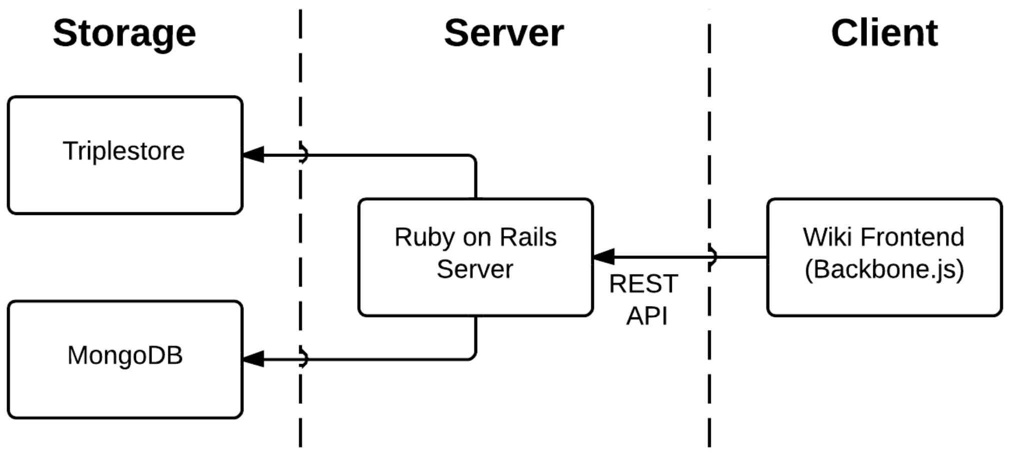


Figure 6.1: Overview over the wiki.

The back end is a MongoDB which can store the actual content of pages. Semantic capabilities are added through the usage of a Triplestore. The logic of 101wiki is implemented in a Ruby on Rails Server. There, versioning of pages and extraction of semantic properties is implemented. Before pages are written into the MongoDB storage, the server first inspects all links for semantic properties, as described in section 5.2. If he finds any, they are extracted and written as RDF triples into the Triplestore. The server exposes actual page data as well as the triples of a page, including backlinks, in a RESTful API. The server is also capable of exporting the data into different formats, like RDF. Figure 6.3 shows the data as exposed by the server in JSON, while figure 6.2 shows the triples of that page, that can be accessed separately.

```
[
  [
    "http://101companies.org/resources/languages/Haskell",
    "http://101companies.org/property/instanceOf",
    "http://101companies.org/resources/concepts/
        Functional_programming language"
  ],
  [
    "http://101companies.org/resources/languages/Haskell",
    "http://101companies.org/property/instanceOf",
    "http://101companies.org/resources/namespaces/Language"
  ],
  ...
]
```

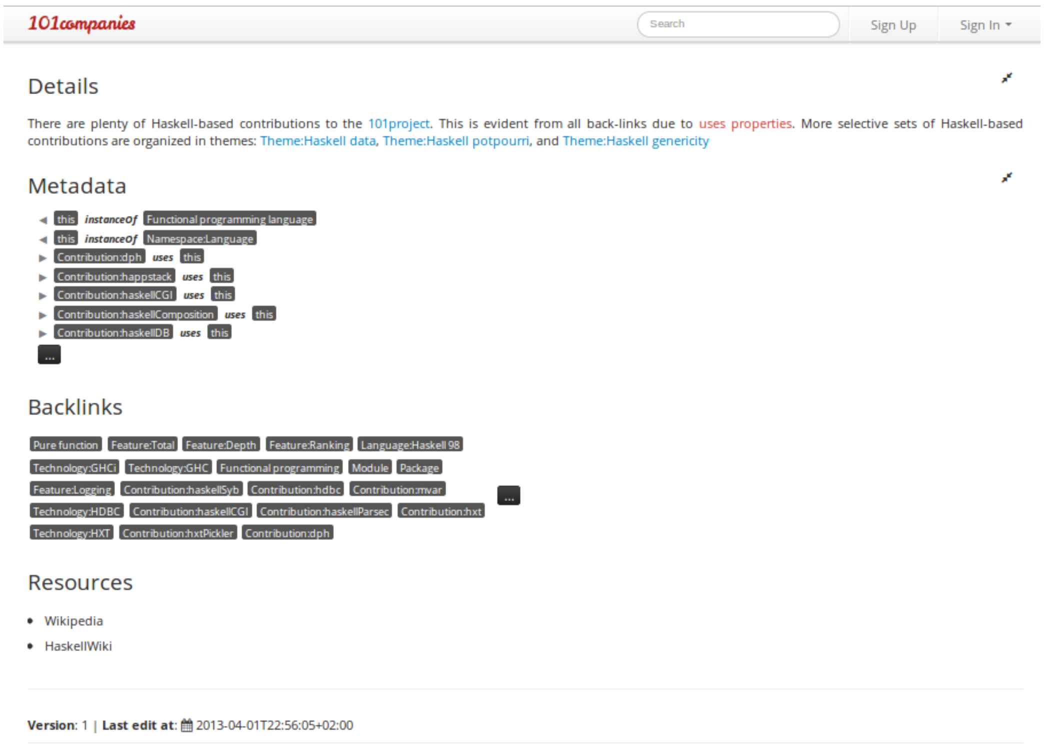
Figure 6.2: Triples for the language Haskell as exposed by the sever (excerpt).

This API is consumed by a JavaScript client written with the Backbone.js Framework. The task of this front end is the actual rendering of 101wiki. Figure 6.4 shows the rendered page of the language Haskell, while figure 6.5 shows the triples as they are rendered on the wiki page[1].

[1]"this" is a shortcut for the actual page, in this case the page for the language Haskell.

```
{
  id: "Language:Haskell",
  content: "...",
  sections: [
    {
      title: "Headline",
      content: "= Headline =An advanced purely-[[functional
          programming language]]"
    },
    {
      title: "Details",
      content: "= Details =There are plenty of Haskell-based
           contributions ..."
    {
      title: "Metadata",
      content: "= Metadata =* [[identifies::http://www.
          haskell.org/]] * [[identifies::http://en.wikipedia.
          org/wiki/Haskell_(programming_language)]] ..."
    }
  ],
  history: {
    _id: "5159f465ef8e3cf9bd000001",
    created_at: "2013-04-01T22:56:05+02:00",
    page: "Language:Haskell",
    updated_at: "2013-04-01T22:56:05+02:00",
    user_id: null,
    version: 1
  },
  backlinks: [
    "Pure_function",
    ...
  ]
}
```

Figure 6.3: A page as returned by the server (excerpt).


101companies Search Sign Up Sign In

Details

There are plenty of Haskell-based contributions to the 101project. This is evident from all back-links due to uses properties. More selective sets of Haskell-based contributions are organized in themes: Theme:Haskell data, Theme:Haskell potpourri, and Theme:Haskell genericity

Metadata

- ◀ this *instanceOf* Functional programming language
- ◀ this *instanceOf* Namespace:Language
- ▶ Contribution:dph *uses* this
- ▶ Contribution:happstack *uses* this
- ▶ Contribution:haskellCGI *uses* this
- ▶ Contribution:haskellComposition *uses* this
- ▶ Contribution:haskellDB *uses* this

...

Backlinks

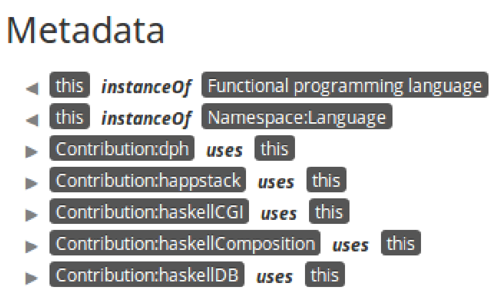
Pure function Feature:Total Feature:Depth Feature:Ranking Language:Haskell 98
Technology:GHCi Technology:GHC Functional programming Module Package
Feature:Logging Contribution:haskellSyb Contribution:hdbc Contribution:mvar ...
Technology:HDBC Contribution:haskellCGI Contribution:haskellParsec Contribution:hxt
Technology:HXT Contribution:hxtPickler Contribution:dph

Resources

- Wikipedia
- HaskellWiki

Version: 1 | **Last edit at:** 2013-04-01T22:56:05+02:00

Figure 6.4: Rendered page for the language Haskell.


Metadata

- ◀ this *instanceOf* Functional programming language
- ◀ this *instanceOf* Namespace:Language
- ▶ Contribution:dph *uses* this
- ▶ Contribution:happstack *uses* this
- ▶ Contribution:haskellCGI *uses* this
- ▶ Contribution:haskellComposition *uses* this
- ▶ Contribution:haskellDB *uses* this

Figure 6.5: Triples for the language Haskell (excerpt) as rendered in 101wiki.

6.2 101worker

101worker, as described in 3.1.3 is charged with the synthesization and derivation of resources based on code artifacts. As mentioned in section 5.3.3, this functionality is implemented in so called modules. Additionally, 101worker needs to expose this data on the web.

6.2.1 Module Implementations

The modules on 101worker can be seen as a collection of programs that are organized in a pipeline and controlled through a "Runner" program. This form is necessary, as they are often dependent on preprocessing and analysis results of previous modules.

The pipeline begins with the assembly of 101repo on the file system of the worker machine. As some support technology, like fact extractors or fragment locators are also stored in 101repo and need to be built before they can be used, this is the next step. After that, some metadata can already be assigned using the 101meta rules. Then, several preprocessing steps are applied such as tokenization, the computation of metrics and fact extraction. The more complex 101meta rules, that might operate on the results of the preprocessing, can be applied to the 101repo artifacts in the next step. In the end of the process the module descriptions are aggregated into a dump and the data stored in 101wiki is downloaded. The process[2] is summarized in figure 6.6.

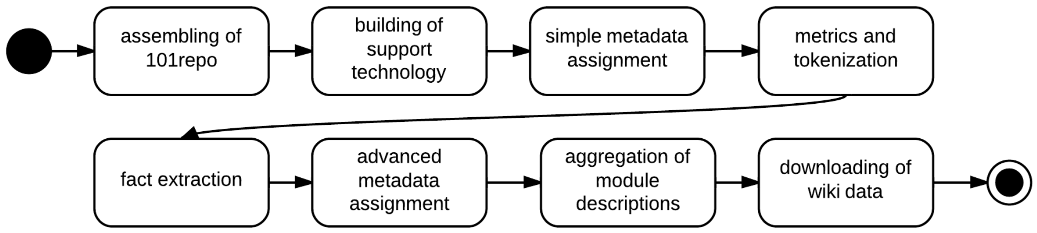

assembling of 101repo
building of support technology
simple metadata assignment
metrics and tokenization
fact extraction
advanced metadata assignment
aggregation of module descriptions
downloading of wiki data

Figure 6.6: Modules executed as a batch process.

[2]For the sake of clarity, some steps are omitted.

6.2.1.1 Assembly of 101repo through a module

For assembling 101repo, the central registry that was described in section 5.1.1 needs to be interpreted. This registry is serialized in a JSON format. Figure 6.7 shows an excerpt of it.

```
[
  {
    "sourcerepo":
      "https://github.com/rlaemmel/yapg.git",
    "sourcedir" : "/",
    "targetdir" : "contributions/yapg"
  },
  {
    "sourcerepo":
      "https://github.com/hakanaksu/101android.git",
    "sourcedir" : "/",
    "targetdir" : "contributions",
    "mode" : "subdirsonly"
  },
  {
    "sourcerepo":
      "https://github.com/101companies/101worker.git",
    "sourcedir" : "/services",
    "targetdir" : "services",
    "mode" : "subdirsonly"
  }
  ...
]
```

Figure 6.7: Serialized version of the repository registry (excerpt).

It is located in a base repository, that serves as a starting point and is checked out onto the file system of the worker machine, or, if it already exists, updated. After that, the registry can be interpreted. A special folder exists where the other repositories are checked out or updated. If a repository has new data, then a check is necessary whether the files are already mounted or not. If they are, these need to be removed before the new files can be copied into the base repository folder structure. After a entry has been interpreted, the copy process is logged in a history file. Once the process is almost complete, the history file will be turned into a dump[3]. This enables the association of 101repo folders with physical GitHub repositories and, by extension, the association of 101wiki pages to physical GitHub repositories. Figure 6.9 shows an excerpt from this dump while figure 6.8 summarizes the process.

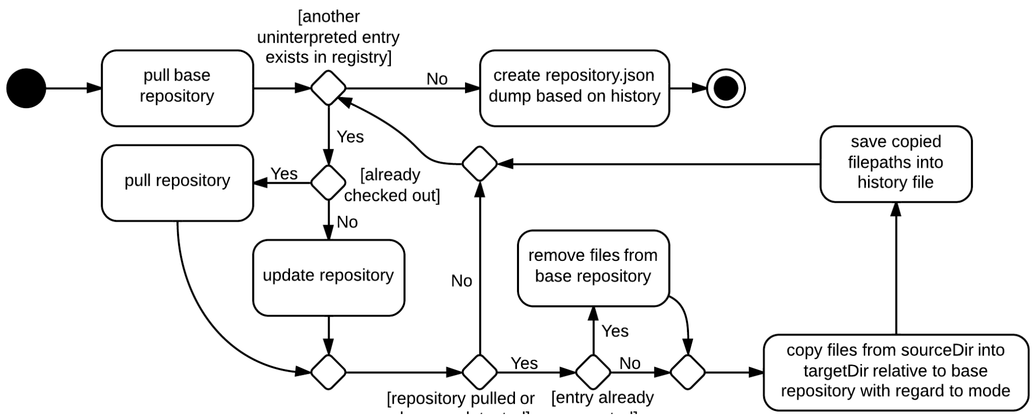

pull base repository
[another uninterpreted entry exists in registry]
No
create repository.json dump based on history
Yes
pull repository
Yes
[already checked out]
No
update repository
save copied filepaths into history file
No
remove files from base repository
Yes
Yes
No
copy files from sourceDir into targetDir relative to base repository with regard to mode
[repository pulled or
[entry already

Figure 6.8: Assembling 101repo.

```
{
  "jqueryDom": "https://github.com/101companies/101repo/
     tree/master/contributions/jqueryDom",
  "javaSyb": "https://github.com/101companies/101simplejava
     /tree/master/contributions/javaSyb",
  "modularLb": "https://github.com/101companies/101datalog/
     tree/master/modularLb",
  "jdom": "https://github.com/101companies/101simplejava/
     tree/master/contributions/jdom",
  "tabaluga": "https://github.com/101companies/101haskell/
     tree/master/contributions/tabaluga",
  "xsdDataSet": "https://github.com/101companies/101repo/
     tree/master/contributions/xsdDataSet",
  ...
}
```

Figure 6.9: The dump created based on assembling 101repo (excerpt).

[3]In figure 5.15, this dump was referenced as repository.json.

6.2.1.2 Assignment of metadata

The assignment of metadata is managed through executing the language 101meta, as described in section 5.3.1, which is serialized in a JSON format. A example displaying the rules for the language Java is shown in figure 6.10. These rules are scattered all over 101repo. Therefore, the first step before interpreting the rules, is to gather them. This results in a dump of all rules[4].

```
[
  {
    "suffix" : ".java",
    "metadata" : [
      {"geshi":"java"},
      {"language":"Java"},
      {"validator": "technologies/JValidator/validator.py"},
      {"extractor": "technologies/JFactExtractor/extractor.
         py"},
      {"locator": "technologies/JFragmentLocator/locator.py"
         }
    ]
  }
]
```

Figure 6.10: Serialized version of 101meta identifying Java files.

Because of the dependencies that exist, the actual interpretation of these rules has to be split up into several modules. Rules of simple complexity can be assigned in the first step. But several rules work on representations of files, as for example created through fact extraction or tokenization, which again need some metadata like language or general nature of the file. The execution of these, more complex rules must be deferred to a later point. The complexity of a rules is tied to the used constraints. Generally, rules that use constraints like the suffix or filename have a low complexity and are for example used to assign the language of a file. Rules that use custom predicates have the highest complexity.

The actual interpretation of the rules is rather simple. The module charged with this task iterates through all code artifacts in 101repo. In every step, it initializes a variable for storing results and goes over all rules matching the current complexity and checks the constraints. If the constraints hold, the metadata is

[4]Earlier referenced as "rules.json"

assigned to the variable. At the end of the process, the variable is saved into a file, which is the derived resource this module creates, and goes on to the next file. At the end of the process, all results are summarized in a dump. Figure 6.11 summarizes the process.

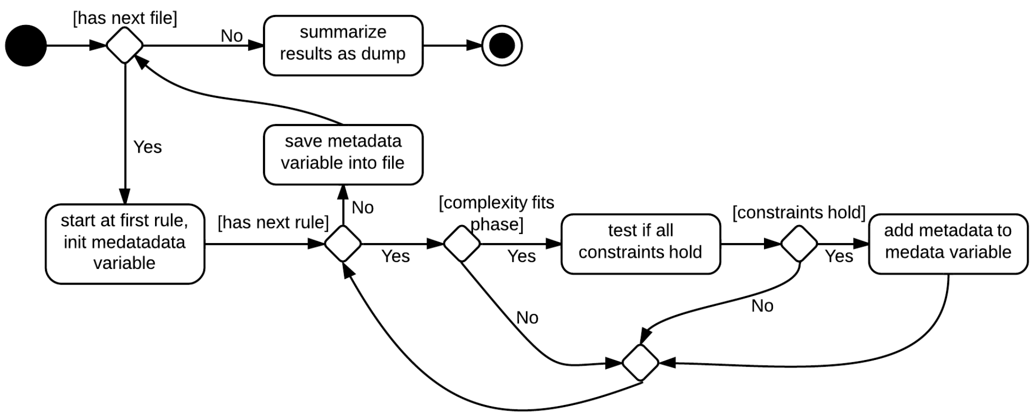

[has next file]
No
summarize results as dump
Yes
save metadata variable into file
start at first rule, init medatadata variable
[has next rule]
No
Yes
[complexity fits phase]
Yes
No
test if all constraints hold
[constraints hold]
Yes
No
add metadata to medata variable

Figure 6.11: Principal workflow of executing 101meta rules.

6.2.2 Web access

101worker provides access to all derived resources through an Apache web server. The web server directly exposes the part of the file system where the resources are stored.

For extended functionality 101worker also provides web services. These have also access to all parts of the file system in which the derivatives or source code artifacts of 101repo are stored. Technically, Django and mod_wsgi are used to enable communication between the Apache server and the Python based services. The request is dispatched to the services through the URL routing implemented in Django. As all services have access to all data, they can be used to filter out specific data or for mashing up several derived resources. Figure 6.12 shows an overview over the service architecture on 101worker.

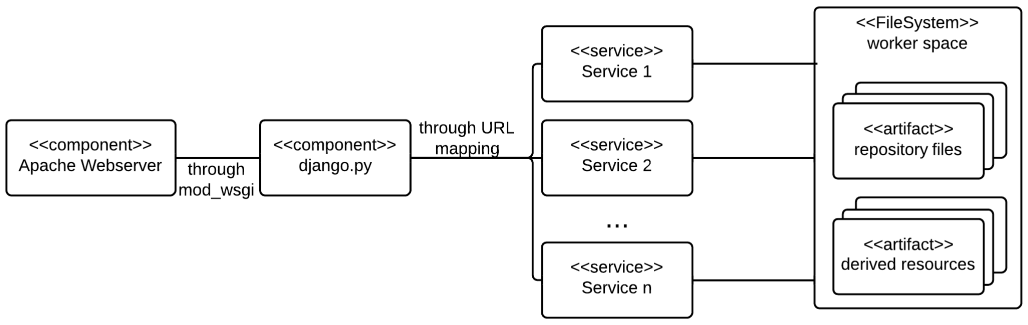


Figure 6.12: Services on the 101worker.

6.3 The exploration service

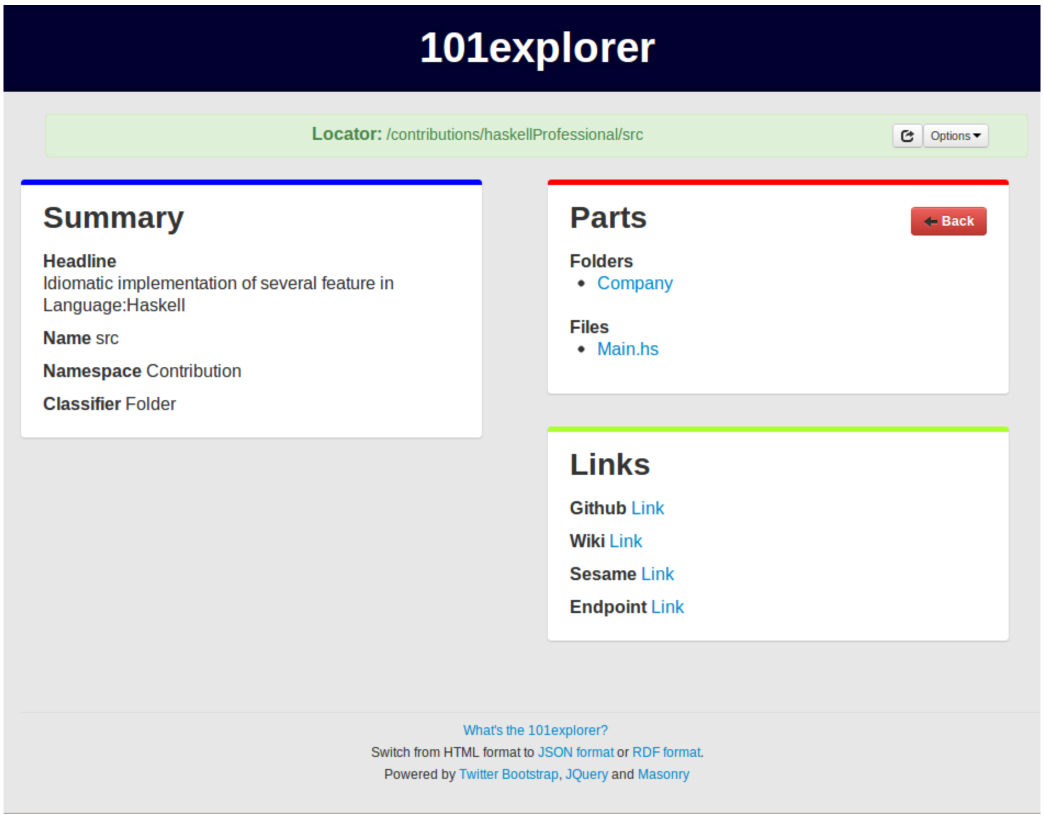
The exploration service (also 101explorer) acts as a view[5] on top of 101repo and enables the appliance of Linked Data principles. Technically, it works through the 101worker web services system. It can serve the repo entities with their metadata and fully linked to their derived resources. This mashing up is possible since, as a service, 101explorer has full access to the assembled repository, the derived resources and the wiki data dumped to 101worker. It can serve data in human readable HTML or machine the machine readable formats RDF and JSON. Figure 6.13 shows a screenshot of this service in the HTML format for a folder in a Haskell-based contribution.

6.3.1 Serving of 101repo entities

As URLs are used as identifiers, the first step of 101explorer when receiving a request is to identify the data class based on the data model described in section 5.1 with the given URL. It can then disassemble and convert the URL into a path on the 101worker file system. By having the file path, the metadata of the entity can be gathered by reading the derived resources. Through that, the service knows for example whether a file is binary or a text file[6] which is important for reading the content. Figure 6.14 shows some of the metadata stored in the derived resources for a Java file.

[5]A view for 101repo was referenced several times in the requirements - in section 4.2.2, 4.2.7 and 4.2.9.

[6]To actually determine this, the "geshi" value is used, which is normally used for code highlighting.



Locator: /contributions/haskellProfessional/src

Summary

Headline
Idiomatic implementation of several feature in Language:Haskell

Name src

Namespace Contribution

Classifier Folder

Parts

Folders
- Company

Files
- Main.hs

Links

Github Link

Wiki Link

Sesame Link

Endpoint Link

What's the 101explorer?
Switch from HTML format to JSON format or RDF format.
Powered by Twitter Bootstrap, JQuery and Masonry

Figure 6.13: Screenshot of the exploration service.

After metadata gathering, the links for the browsing experience are collected. If the entity is a namespace, then the association of wiki namespaces to folders (see section 5.2) is exploited and the links for browsing are constructed based on derived resources[7] created by a module. In the case of a folder or namespace member, the links are constructed by a lookup on the file system. For files and fragments, the links are constructed with the results of fact extraction. The necessary fact extractor can be read via the metadata as shown in 6.14. Also, for those two, an extra for reading the content is necessary. For files, the complete content is read while for fragments only the line range specified by the fragment locator, which is again specified by the metadata (see figure 6.14), is read.

After that, the wiki data can be added. As 101wiki uses singular names for namespaces and 101repo plural names, a mapping is applied to convert the namespace into a 101wiki style namespace before the actual data is read from the

[7]In figure 5.14, this folder based resource was referenced as members.json

```
[
  {
    id: 404,
    metadata: { geshi: "java" }
  },
  {
    id: 404,
    metadata: { language: "Java" }
  },
  {
    id: 404,
    metadata: {
      extractor: "technologies/JFactExtractor/extractor.py"
    }
  },
  {
    id: 404,
    metadata: {
      locator: "technologies/JFragmentLocator/locator.py"
    }
  }
  ...
```

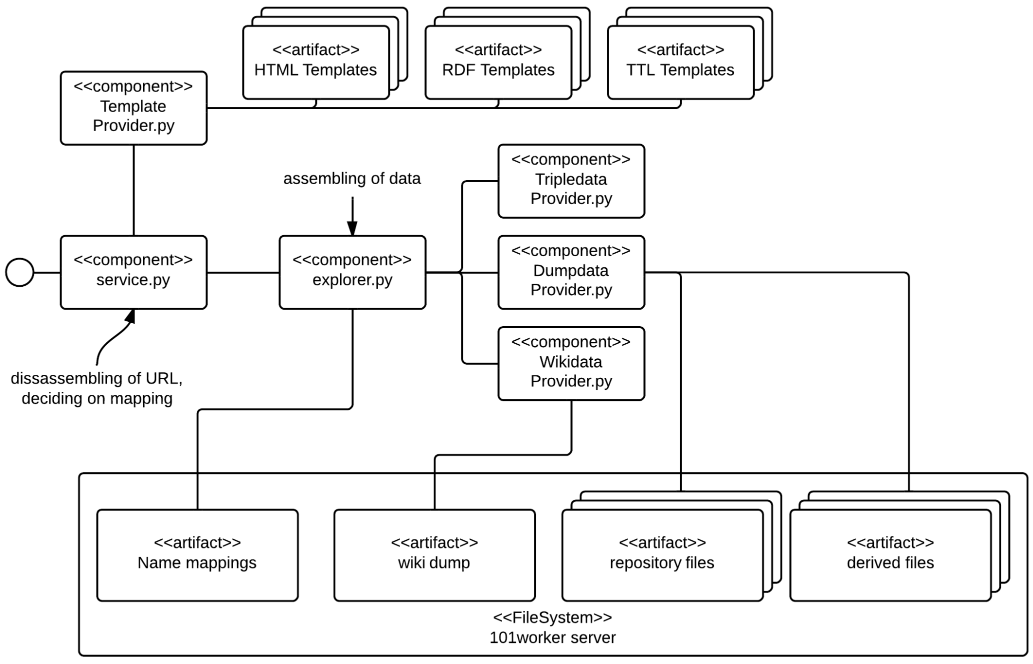
Figure 6.14: Some simple metadata units about a Java file.

wiki dump as shown in figure 6.15.

Then, the links to other data sources are constructed and the data can be returned. If the requested format was JSON, this is achieved through the build in Python methods. For other formats, Template files are applied to the data to convert them into the different formats. Fig. 6.16 shows the general architecture of the service, while 6.17 shows a activity diagram summarizing the workflow for a incoming request.

```
[
  {
    page: {
      type: "Subject",
      page: {
        p: "Language",
        n: "Prolog"
      },
      headline: " A [[programming language]] for [[logic
          programming]]",
      internal_links: [
        "programming language",
        "logic programming",
        "instanceOf::Logic programming language",
        "identifies::http://en.wikipedia.org/wiki/Prolog",
        "instanceOf::Namespace:Language"
      ],
      ...
```

Figure 6.15: Excerpt from the dumped wiki data.



<<artifact>> HTML Templates
<<artifact>> RDF Templates
<<artifact>> TTL Templates
<<component>> Template Provider.py
assembling of data
<<component>> Tripledata Provider.py
<<component>> service.py
<<component>> explorer.py
<<component>> Dumpdata Provider.py
<<component>> Wikidata Provider.py
dissassembling of URL, deciding on mapping
<<artifact>> Name mappings
<<artifact>> wiki dump
<<artifact>> repository files
<<artifact>> derived files
<<FileSystem>> 101worker server

Figure 6.16: Architecture of the exploration service.

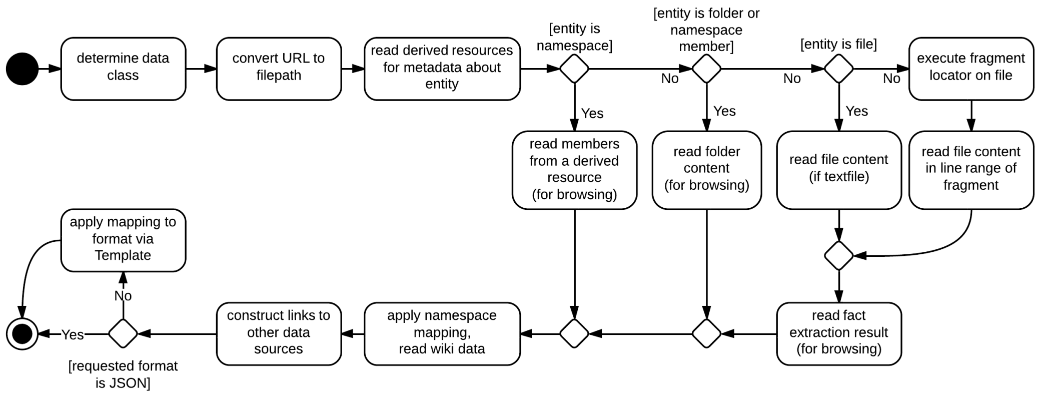

determine data class
convert URL to filepath
read derived resources for metadata about entity
[entity is namespace]
[entity is folder or namespace member]
[entity is file]
execute fragment locator on file
No
No
No
Yes
Yes
Yes
read members from a derived resource (for browsing)
read folder content (for browsing)
read file content (if textfile)
read file content in line range of fragment
apply mapping to format via Template
No
read fact extraction result (for browsing)
apply namespace mapping, read wiki data
construct links to other data sources
Yes
[requested format is JSON]

Figure 6.17: Workflow of the exploration service.

6.3.2 Link construction

Constructing the links to other data sources requires several derived resources. For constructing the GitHub link, the dump created during the repo assembly (see section 6.2.1.1) can be used. As every namespace member should be listed there, the physical repository can be extracted easily. If they are unlisted, they are expected to be in the base repository. Then the remaining file path and, in the case of fragments, the line range can be appended to the GitHub URL.

Constructing links to 101wiki pages and links to directly retrieve the Triplestore data is trivial, as the only problem is the different convention about namespace names which can be bridged through the mapping.

For associating derived the derived files with their modules, all necessary data comes from the module description summary[8]. An excerpt of this file is shown in figure 6.18.

If the 101repo entity is a file, all derived resources can be identified through their suffix. So, for every derived resource, the filename is inspected and linked to a module once a match is found. Folder-based derived resources can be identified and associated with a module through their filename. The complete process is summarized in figure 6.19.

[8]In figure 5.15, this dump was referenced as moduleDescriptions.json.

```
{
  "suffix": {
    ".extractor.json" : {
      "descr" : {
        "headline": "Extracted facts",
        "scope": "file",
        "suffix": ".extractor.json",
        "language": "JSON"
      },
      "name": "extract101meta"
    },
    ...
  },
  "filename": {
    ...
```

Figure 6.18: Summary of module descriptions (excerpt).

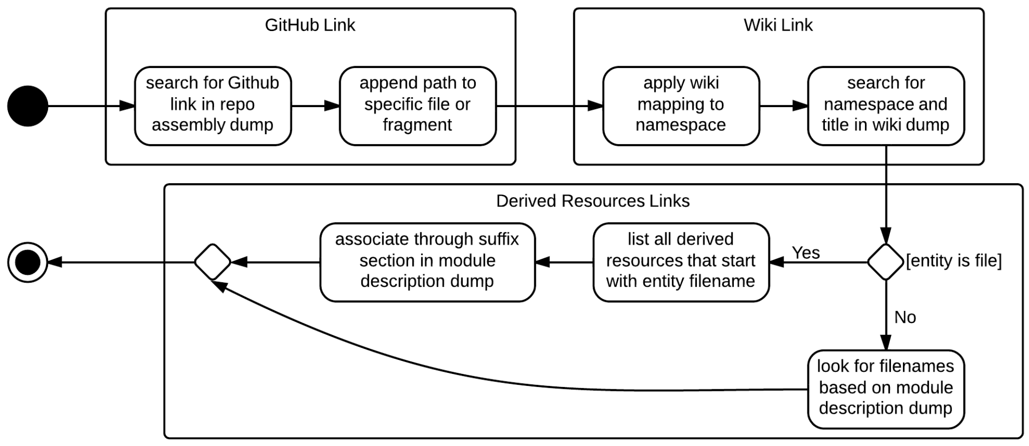

GitHub Link
search for Github link in repo assembly dump
append path to specific file or fragment
Wiki Link
apply wiki mapping to namespace
search for namespace and title in wiki dump
Derived Resources Links
associate through suffix section in module description dump
list all derived resources that start with entity filename
Yes
[entity is file]
No
look for filenames based on module description dump

Figure 6.19: Creation of links to other data sources.

Chapter 7

Evaluation

7.1 Evaluation through the requirements

In order to evaluate the presented system, the requirements are reexamined including a quick schema for the solution.

7.1.1 Navigate from wiki to repo

As required in section 4.2.1, it is possible to go directly to the physical GitHub repositories from the wiki. This is displayed in figure 7.1.

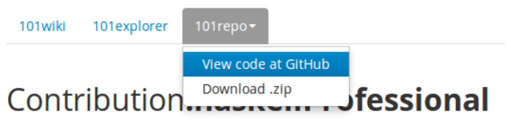
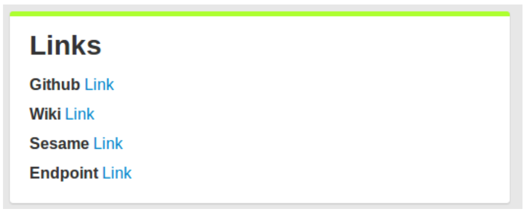


Figure 7.1: Navigating from the wiki to the repo.

It is achieved through a specialized service on 101worker, that works on the dump created after the assembly of 101repo as described in section 6.2.1.1. The basic functionality is similar to the creation of the GitHub link in the exploration service, but as some other, wiki-specific data is involved, the functionality has been separated.

7.1.2 Navigate from repo to wiki

As required in section 4.2.2, this is possible through the 101repo view provided by the exploration service. Reconstruction of the link to 101wiki is simple as folders up to the member level correspond to wiki pages. As folders and files beyond that level are associated with the member page (see section 5.2), the member they belong to is taken when constructing the link. Other than that, it is simply a matter of taking the Mapping as described in section 6.3 to overcome the different conventions, if necessary. The result is shown in figure 7.2.



Links

Github Link
Wiki Link
Sesame Link
Endpoint Link

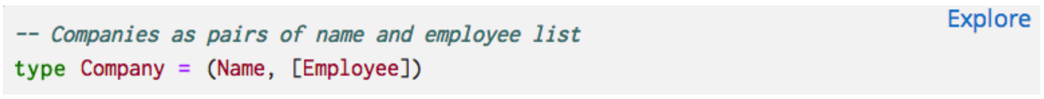
Figure 7.2: Navigating from the exploration view to the wiki.

7.1.3 Referencing source code on wiki pages

Referencing of source code as requested in section 4.2.3 on 101wiki pages is achieved through the JSON based responses of the exploration service (see section 6.3). Markup code is placed in the actual text of the wiki page. The exploration service will return the content of the file or fragment as well as metadata necessary for code highlighting based on the details given in the markup. It can then be replaced with the actual code by markup parser implemented in the wiki server. An example is the following markup

```
<fragment url="src/Main.hs/type/Company"/>
```

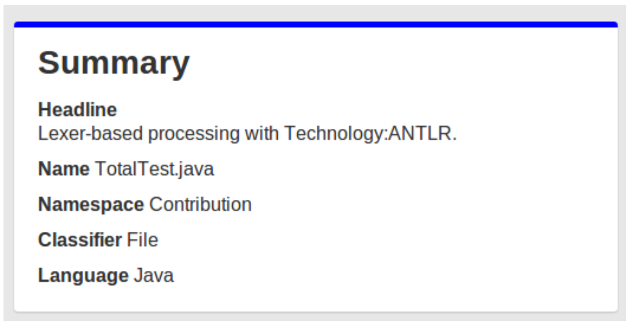
on the wiki page of a simple Haskell-based contribution. As this URL is given relative to a contribution page, it will first be transformed it into an absolute URL and then request the data from the exploration service. The result is then rendered as follows:



```
-- Companies as pairs of name and employee list
type Company = (Name, [Employee])
```

7.1.4 Displaying of metadata for 101repo entities

As demanded in section 4.2.4, every entity accessed through the exploration service provides metadata that were taken from derived resources as described in section 6.3. Figure 7.3 shows an example of this - among other information, every file displays the associated language.



Summary

Headline
Lexer-based processing with Technology:ANTLR.

Name TotalTest.java

Namespace Contribution

Classifier File

Language Java

Figure 7.3: Metadata in the exploration service.

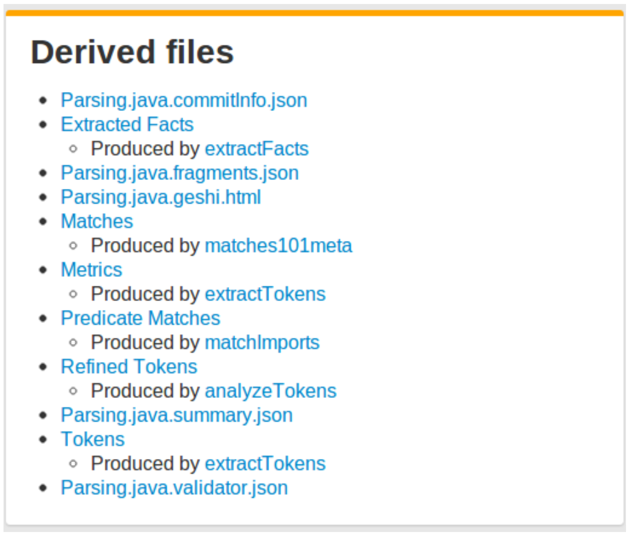
7.1.5 Associate derived resources with primary resources

As demanded in section 4.2.5, every entity accessed through the exploration service provides links to all derived resources. The creation of these links was shortly described in section 6.3. Figure 7.4 shows an example of this - all derived files for a Java source code artifact. Resources for which the producing module is missing currently don't have a module description and therefore can't be associated.

7.1.6 Operate the wiki like a graph

As required in section 4.2.6 101wiki can be seen and treated as a graph. Pages are the nodes while typed links provide the edges. A example for this is shown in figure 7.5, which shows data from a Haskell-based contribution in the graph-based RDF data model[1].

[1]Originally, this data was returned as N-Triples from the wiki server. It was converted into RDF/XML for the sake of consistency in this thesis.



Derived files

- Parsing.java.commitInfo.json
- Extracted Facts
  - Produced by extractFacts
- Parsing.java.fragments.json
- Parsing.java.geshi.html
- Matches
  - Produced by matches101meta
- Metrics
  - Produced by extractTokens
- Predicate Matches
  - Produced by matchImports
- Refined Tokens
  - Produced by analyzeTokens
- Parsing.java.summary.json
- Tokens
  - Produced by extractTokens
- Parsing.java.validator.json

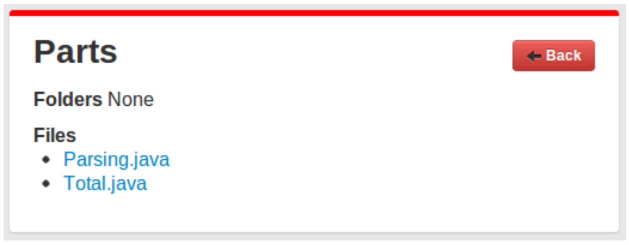
Figure 7.4: Derived resources in the exploration service.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:wik="http://101companies.org/schemas/wiki">
  <rdf:Description rdf:about="http://101companies.org/resources/
      contributions/haskellTree">
    <wik:uses rdf:resource=".../languages/Haskell"/>
    <wik:uses rdf:resource=".../technologies/GHC"/>
    <wik:implements rdf:resource=".../features/Total"/>

    <!--- ... further data omitted ... -->
  </rdf:Description>
</rdf:RDF>
```

Figure 7.5: 101wiki data for a Haskell-based contribution.

7.1.7 Operate the repo like a tree

As required in section 4.2.7, 101repo can be browsed through the exploration service. Figure 7.6 shows the navigation options in the HTML version of the exploration service.



Parts

Folders None

Files
- Parsing.java
- Total.java

Figure 7.6: Browsing through 101repo with the exploration service.

7.1.8 Querying for data

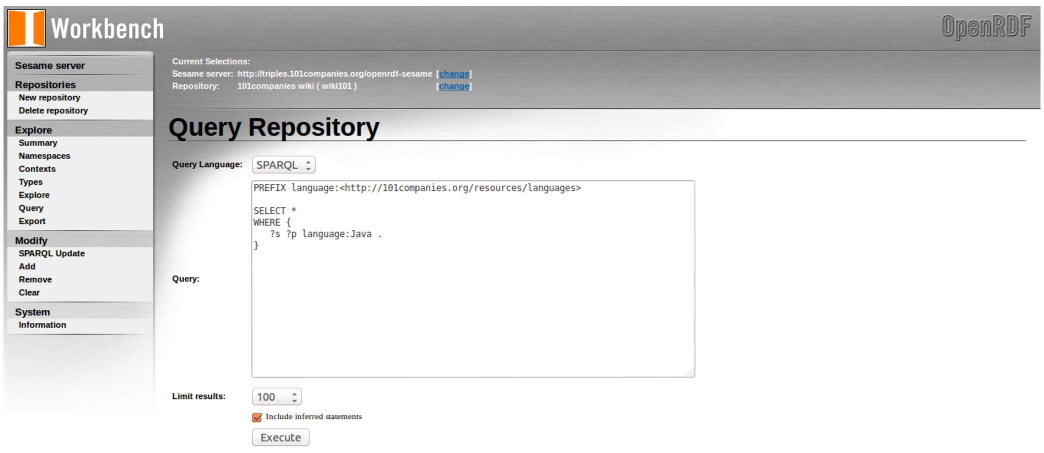
As demanded in section 4.2.8, both 101wiki and 101wiki can be queried via SPARQL. For the repository, this is best done against a precomputed dump as derived by a module based on the exploration service[2]. But it is also possible to directly build the RDF graph in memory and query against that. Figure 7.7 shows a Python program using the library RDFlib to do a simple SPARQL query extracting all Java files from the exploration service dump.

```
g = rdflib.Graph()
g.load('http://data.101companies.org/dumps/explorer.rdf')
namespace =
  rdflib.Namespace('http://101companies.org/schema/repo')

#querying for all java files in the crestomathy
queryResult = g.query("""
                SELECT DISTINCT ?file
                WHERE {
                    ?file co:classifier "File" .
                    ?file co:language "Java"
                }
               """,
               initNs=dict(
                   co=namespace
               ))

for file in queryResult.result:
    print file
```

Figure 7.7: SPARQL query extracting all Java files.

[2]In figure 5.15, this dump was referenced as explorer.rdf

For 101wiki, the data can either be queried via the Triplestore. GUI and via programmatic access are available for that. Figure 7.8 shows a SPARQL query listing all triples where the language Java is the object executed via the GUI.



Current Selections:
Sesame server: http://triples.101companies.org/openrdf-sesame [change]
Repository: 101companies wiki (wiki101) [change]

Query Language: SPARQL

Query:

```
PREFIX language:<http://101companies.org/resources/languages>

SELECT *
WHERE {
   ?s ?p language:Java .
}
```

Limit results: 100

[✓] Include inferred statements

Figure 7.8: SPARQL query on 101wiki.

7.1.9 Human and machine readable data

As required in section 4.2.9, all data is available in JSON, RDF/XML and HTML. As HTML data was already shown, figure 7.9 and figure 7.10 show some data available for a file in a Java based contribution in JSON and RDF/XML.

```
{
  name: "Serialization.java",
  language: "Java",
  headline: "Modular programming with static methods in
     Language:Java",
  namespace: "Contribution",
  geshi: "java",
  classifier: "File",
  ...
}
```

Figure 7.9: JSON serialized data about a file.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:co="http://101companies.org/schemas/repo">
  <rdf:Description rdf:about=".../javaStatic/src/main/java/org/
      softlang/company/features/Serialization.java?format=rdf">
    <rdf:type rdf:resource="http://101companies.org/schemas/repo#
        File"/>
    <co:name>Serialization.java</co:name>
    <co:namespace>Contribution</co:namespace>
    <co:classifier>File</co:classifier>
    <co:geshi>java</co:geshi>
    <co:language>Java</co:language>

    <!--- ... other data omitted ... -->
  </rdf:Description>
</rdf:RDF>
```

Figure 7.10: RDF/XML serialized data about a file.

7.2 Further scenario based evaluation

In order to further evaluate and the system and to demonstrate its capabilities, several scenarios are exercised. These scenarios show that the results of this thesis not only tackle existing problems, but also enable other software engineering research that was previously unfeasible.

7.2.1 Clone detection

As the software chrestomathy contains many small software systems that are similar by design, clones are expected. Therefore, the exploration service can be used to exercise clone detection. To do that, one must basically traverse the 101repo tree with the exploration view to extract all files. Then, the file contents, which will be returned (among other data) when dereferencing the file URIs, can be mapped against the actual file URIs. Through that, perfect clones in 101repo can be detected. Figure 7.11 contains the code for this, while figure 7.12 shows an excerpt of the files that were found. It can be seen that a XML schema is used several times. Also, a data model used in Haskell-based contributions and some Operations in Java-based contributions are used on multiple occasions.

The initial approach presented here is trivial, but it can easily be extended. As every file also returns the language it belongs to, implementing a language filter so specifically search for clones in specific languages is easily doable. Other

```
# Collect files and content recursively
def extractFiles(entity, languages=[]):
   files = []
   data = loadPage(entity['resource'])

   for f in data.get('files', []):
      fileData = getFileData(f)
      files.append({
         'uri' : f['resource'],
         'data': fileData
      })

   for d in data.get('folders', []):
      files += extractFiles(d)

   return files

# Iterate over all contributions
filesList = []
root = 'http://101companies.org/resources/contributions'
contributions = loadPage(root)
for member in contributions['members']:
   filesList += extractFiles(member)

# Hash map content to file URIs
contents = {}
for file in filesList:
   content = file['data']['content']
   if not content in contents:
      contents[content] = []
   contents[content].append(file['uri'])
```

Figure 7.11: Python code for clone detection with the exploration service.

possible extension are a fragment based clone detection or a approach for less than perfect clones.

```
[
 "dom/Company.xsd",
 "jdom/Company.xsd",
 "sax/Company.xsd",
 "scalaXML/Company.xsd",
 "xom/Company.xsd",
 "xquery/Company.xsd",
 "xslt/Company.xsd"
],
[
 "haskellSyb/src/Company/Data.hs",
 "monoidal/src/Company/Data.hs",
 "nonmonadic/src/Company/Data.hs",
 "writerMonad/src/Company/Data.hs"
],
[
 "jaxbChoice/src/test/java/org/softlang/tests/Operations.
    java",
 "jaxbExtension/src/test/java/org/softlang/tests/
    Operations.java",
 "jaxbSubstitution/src/test/java/org/softlang/tests/
    Operations.java"
]
```

Figure 7.12: Results of perfect clone detection (excerpt).

7.2.2 Metrics based comparison of contributions

A common expectation is that a software chrestomathy should not just list programming languages, but also compare them in different dimensions on the grounds its contributions. A possible example is to compare the conciseness of the languages in terms of lines of code (LOC). Clearly several several problems concerning validity need to be solved before this question can be answered scientifically, but the scenario described here shows the technical side of how the chrestomathy and the data model can be used for this.

To determine which languages need the least lines of code, contributions that implement the same feature set are examined. A result for a specific feature set is shown in figure 7.13.

```
{
  "featureSet" : [
      "Hierarchical_company",
      "Total",
      "Cut"
  ]
}
```

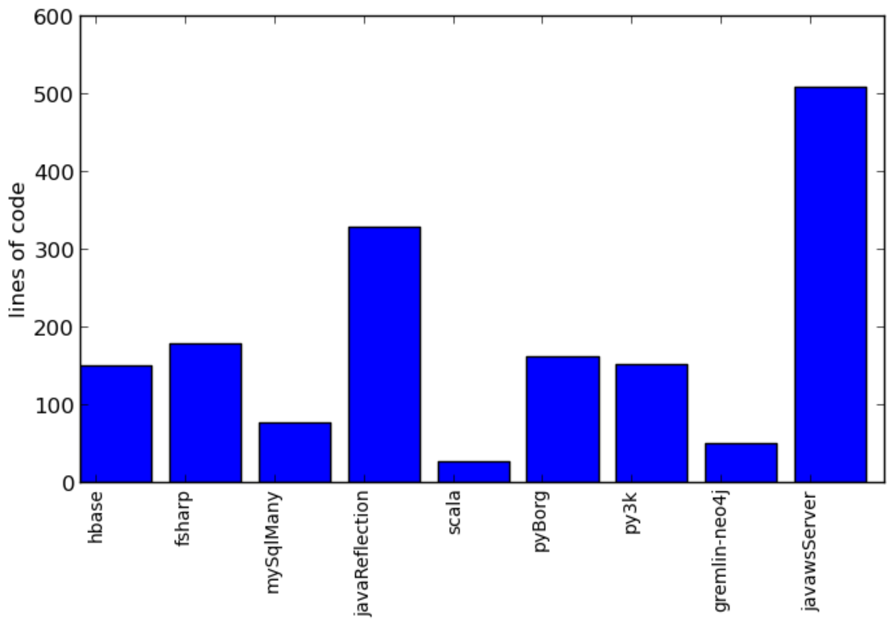


Figure 7.13: LOC for contributions (lower) of the same feature set (upper).

This can be achieved by walking over the repository tree as provided by the exploration service while gathering all contributions. The 101wiki Triplestore can provide links between contributions and features, therefore one needs to use the link provided by the exploration service for every contribution to access these triples and to extract the feature set of a contribution. Then, the files can be extracted for every contribution. Metrics for every file will be available as a derived resource and can once again be accessed through a link. The metrics can then be summed up for every contribution and contributions of the same feature set can be compared. An simplified source code is shown in figure 7.14.

```
configs = {} # Associate feature configurations with contributions
metrics = {} # Associate contributions with LOC metric

# Iterate over all contributions
contribs = loadPage(
   'http://101companies.org/resources/contributions'
)
for contrib in contribs['members']:
   data = loadPage(contrib['resource'])

   # Collect features for contribution
   features = retrieveFeatures(data['triplestore'])
   key = tuple(features)
   if not key in configs:
      configs[key] = {'features': features, 'contribs': []}

   # Map feature configuration to contribution
   configs[key]['contribs'].append(contrib['name'])

   # Aggregate LOC for all files of the contribution
   files = collectFiles(contrib['resource'])
   loc = 0
   for file in files:
      mdata = retrieveMetrics(file['resource'])
      if not mdata ={}:
        if not 'relevance' in mdata
          or mdata['relevance'] ='system':
             loc += int(mdata['loc'])
   metrics[contribution['name']] = loc
```

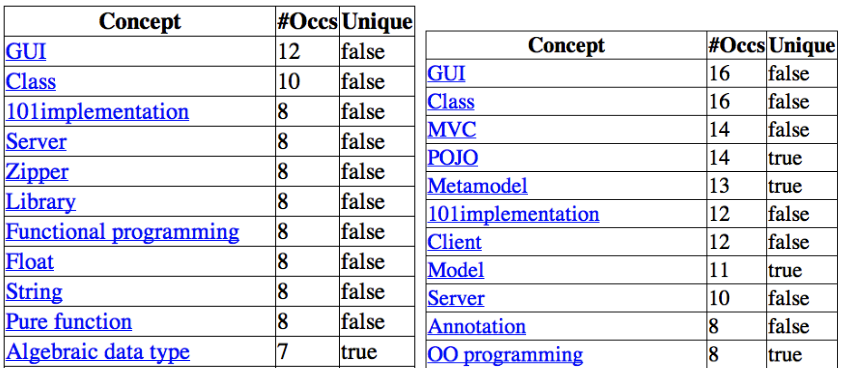
Figure 7.14

7.2.3 Concept analysis

The last scenario presented here is to associate concepts, as they are referenced on the documentation of contributions, to program paradigms. Then, the number of occurrences can be counted. It can be determined whether a concept is exclusive to a programming paradigm[3]. Figure 7.15 shows examples for the results of this analysis. It can for example be seen that the concept "Algebraic data type" is exclusively used in functional programming paradigms, while the POJO concept

[3]At least according to the data stored in the chrestomathy.

is referenced exclusively in an object oriented paradigm.



Concept	#Occs	Unique
GUI	12	false
Class	10	false
101implementation	8	false
Server	8	false
Zipper	8	false
Library	8	false
Functional programming	8	false
Float	8	false
String	8	false
Pure function	8	false
Algebraic data type	7	true

Concept	#Occs	Unique
GUI	16	false
Class	16	false
MVC	14	false
POJO	14	true
Metamodel	13	true
101implementation	12	false
Client	12	false
Model	11	true
Server	10	false
Annotation	8	false
OO programming	8	true

Figure 7.15: Concepts associated with functional (left) and and object oriented paradigms (right).

The results are computed by querying the wiki graph. In the code shown in figure 7.16, the Gremlin query language is used. The query starts at the node for the paradigms and walks from there via the languages to the contributions. There, it checks all links with the type "mentions". As mentions is not exclusively used for concepts, other uses of "mentions" have to be filtered out.

```
static final String resources =
  'http://101companies.org/resources/'
static final String properties =
  'http://101companies.org/property/'

public findConcepts(paradigm) {
  def concept =
    getResource(resources + 'namespaces/Concept')

  def concepts = graph.v(paradigm).
    inE(properties + 'instanceOf').outV. //Languages
    inE(properties + 'uses').outV. //Contributions
    outE(properties + 'mentions').inV. //Mentions
    toList().findAll{ //Concept mentions only
      it.outE(properties + 'instanceOf').inV.
      filter{it =concept}.toList().size() > 0
    }
  return concepts
}
```

Figure 7.16: Concept analysis in Groovy.

Chapter 8

Conclusion

8.1 Summary

The basic idea of this thesis was to apply Linked Data principles on the 101companies project, meaning that the data found in the chrestomathy is fully linked and explorable in a resource-oriented fashion. The basic data schemes and implementation have been described (chapters 5 and 6) and the results have been evaluated (chapter 7). It was shown that by fully linking the data, that the problems created by the diverse code artifacts can be tackled and additional software engineering research can be enabled enabled.

8.2 Future work

There are several improvements and problems that still need to be addressed. The schemas shown in this thesis are currently mainly used for validation. However, the goal is to use them for schema mapping. Several schemas are still missing, especially schemas describing the contents of derived resources. But if combined with the already existing schemas, they could for example be used to actually produce the data that the exploration service creates instead of just validating it.

Further ideas intend to improve the available metadata, especially for fragments as most of the data available for them, such as the language, is inherited from the file. An example are features and concepts. In the current system, they can be assigned based on tokenization results, but as tokenization only considers complete files at this point, associating concepts and features with fragments is not possible. Furthermore, concepts that are divided on several files, like a MVC Pattern, are problematic. It may be possible to detect the individual parts of such a pattern, however identifying the set of files as one concept would be ideal. This also raises the question how one would expose such a pattern.

Lastly, one must also recognize that, while the 101companies project is integrated into the global data space created through the Linked Data principles, it currently does not take advantage of it. [BHBL09] mentions that Linked Data applications can benefit from operating on top of an unbound global data space. Generally, there is great potential in being able to easily enhance ones own data with data provided by other sources and it needs to be evaluated what sources are potentially interesting for improving the software chrestomathy. Possibilities range from enhancing the chrestomathy data with DBPedia articles to using external services such as the SeCold[1] fact extraction [KFRC11].

[1]http://www.secold.org/

Bibliography

[Ber12] Olivier Berger. Linked data descriptions of debian source packages using adms.sw. In *Proc. of Semantic Web Enabled Software Engineering*, 2012.

[BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[BL07] Tim Berners-Lee. Linked data. `http://www.w3.org/DesignIssues/LinkedData.html`, July 2007. [Online; accessed 01.07.2013].

[BM03] Dave Beckett and Brian McBride. Rdf/xml syntax specification. `http://www.w3.org/TR/2003/WD-rdf-syntax-grammar-20031010/`, October 2003. [Online; accessed 03.07.2013].

[Dek13] Makx Dekkers. Asset description metadata schema (adms). `https://dvcs.w3.org/hg/gld/raw-file/default/adms/index.html`, June 2013. [Online; accessed 08.07.2013].

[FLL+12] Jean-Marie Favre, Ralf Lammel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. Linking documentation and source code in a software chrestomathy. In *Proc. of WCRE 2012*, pages 335–344. IEEE, 2012.

[FLSV12] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. *101companies*: a community project on software technologies and software languages. In *Proc. of TOOLS 2012*, volume 7304 of *LNCS*, pages 59–74. Springer, 2012.

[FLV12] Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proc. of MODELS 2012*, volume 7590 of *LNCS*, pages 151–167. Springer, 2012.

[Goe11] Stijn Goedertier. Asset description metadata schema for software. https://joinup.ec.europa.eu/asset/adms_foss/description, December 2011. [Online; accessed 08.07.2013].

[GTS10] Lars Grammel, Christoph Treude, and Margaret-Anne Storey. Mashup environments in software engineering. In *Proceedings of the 1st Workshop on Web 2.0 for Software Engineering*, Web2SE '10, pages 24–25, New York, NY, USA, 2010. ACM.

[GZC13] Francis Galiegue, Kris Zyp, and Gary Court. Json schema: core definitions and terminology json-schema-core. http://json-schema.org/latest/json-schema-core.html, January 2013. [Online; accessed 03.07.2013].

[HB11] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

[How08] James Howison. Cross-repository data linking with RDF and OWL: Towards common ontologies for representing FLOSS data. In *WoPDaSD (Workshop on Public Data at International Conference on Open Source Software)*, 2008.

[ICH12] Aftab Iqbal, Richard Cyganiak, and Michael Hausenblas1. Integrating floss repositories on the web. 2012.

[IH12] Aftab Iqbal and Michael Hausenblas. Integrating developer-related information across open source repositories. In Chengcui Zhang, James Joshi, Elisa Bertino, and Bhavani M. Thuraisingham, editors, *IRI*, pages 69–76. IEEE, 2012.

[IUHT09] Aftab Iqbal, Oana Ureche, Michael Hausenblas, and Giovanni Tummarello. Ld2sd: Linked data driven software development. In *SEKE*, pages 240–245. Knowledge Systems Institute Graduate School, 2009.

[Jac03] Elin K. Jacob. Ontologies and the semantic web. *Bulletin of the American Society for Information Science and Technology*, 29(4):19–22, 2003.

[KC04] Graham Klyne and Jeremy J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, February 2004. [Online; accessed 01.07.2013].

[KFH$^+$12] Iman Keivanloo, Christopher Forbes, Aseel Hmood, Mostafa Erfani, Christopher Neal, George Peristerakis, and Juergen Rilling. A linked data platform for mining software repositories. In Michele Lanza, Massimiliano Di Penta, and Tao Xi, editors, *MSR*, pages 32–35. IEEE, 2012.

[KFRC11] Iman Keivanloo, Christopher Forbes, Juergen Rilling, and Philippe Charland. Towards sharing source code facts using linked data. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, pages 25–28, New York, NY, USA, 2011. ACM.

[KLL$^+$13] Kevin Klein, Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. A Linked Data approach to surfacing a software chrestomathy. 20 pages. Submitted for publication. Available online since 21 June 2013., 2013.

[MG08] Shah J. Miah and John Gammack. A mashup architecture for web end-user application designs. In *Second IEEE International Conference on Digital Ecosystems and Technologies*, 2008.

[Moa97] R. Moats. Urn syntax. http://www.ietf.org/rfc/rfc2141.txt, May 1997. [Online; accessed 03.07.2013].

[Pas05] Norman Paskin. Digital object identifiers for scientific data. *Data Science Journal*, 4:12–20, 2005.

[Per11] Vassilios Peristeras. Open government metadata. http://joinup.ec.europa.eu/elibrary/document/towards-open-government-metadata, September 2011. [Online; accessed 01.07.2013].

[unk13] unkown. Chrestomathy. http://en.wikipedia.org/wiki/Chrestomathy, June 2013. [Online; accessed 10.07.2013].