Revising Wikipedia's computer language domain based on bad smells

Masterarbeit

zur Erlangung des Grades
eines Master of Science
im Studiengang Informatik

vorgelegt am 31. März 2015
von

Marcel Heinz

Erstgutachter : Ralf Lämmel
Zweitgutachter : Martin Leinberger

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen - benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. ja ☐ nein ☐

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. ja ☐ nein ☐

Koblenz, den 31. März 2015

Abstract

An ontology serves as a collection of information pieces representing either general knowledge or a specific domain. Various ways exist to create an ontology. Other than manual labor, information extraction programs deal with the problem of retrieving information from available sources considering a specified set of criteria.

Wikipedia is currently the biggest available encyclopedia online. Therefore it is the target of numerous information extraction approaches. The first part of this thesis is concerned with information extraction from Wikipedia. It shows what may be extracted from pages and how it can be mapped to an underlying model.

The second part is concerned with the analysis of an extracted ontology. Common ways known in the field of software engineering are utilized to analyze an ontology that was extracted from Wikipedia. A bad smell approach with a preceding metric analysis is proposed. The bad smells are inspired by a set of related work from various areas such as semantic wikis, source code refactoring and general ontologies.

Based on the results from the second part the third part contains a proposal on how to improve the extracted ontology. The proposal is inspired by related work on general ontologies and source code refactoring. For ontologies one may distinguish between three primary revising activities namely refine, prune and refactor. In this thesis a three staged pruning algorithm is adapted from an existing approach designed for general ontologies. Its goal is to remove irrelevant or flawed information in the extracted ontology. Finally, a small set of refactorings is described that are needed to improve the structure of the extracted ontology. The refinement activity is out of scope for this thesis. Exemplary approaches to refinement are summarized in the context of related work.

Zusammenfassung:

In einer Ontologie können Informationen zu allgemeinem Wissen oder einer spezifischen Domäne strukturiert erfasst werden. Es existieren verschiedene Wege, eine solche Ontologie zu erstellen. Neben der manuellen Erstellung existieren Verfahren, die automatisch Informationen unter Berücksichtigung von aufgestellen Kriterien aus einer Wissensquelle extrahieren.

Wikipedia ist die zur Zeit größte online verfügbare Enzyklopädie. Viele verschiedene Arbeiten haben das Ziel, die vorhandenen Informationen aus Wikipedia zu extrahieren. Der erste Teil dieser Arbeit beschäftigt sich mit der Extraktion von Informationen aus Wikipedia. Die extrahierten Informationen werden in einer Ontologie gesammelt. Dabei wird ein Überblick dazu gegeben, welche strukturierten Informationen auf den verschiedenen Seitentypen vorhanden sind. Diese werden letztendlich auf ein entworfenes Modell abgebildet.

Der zweite Teil dieser Arbeit handelt von der Analyse der extrahierten Ontologie. Dazu werden bekannte Verfahren aus der Softwaretechnik zur Analyse einer speziell aus Wikipedia extrahierten Ontologie verwendet. Nach einer Analyse basierend auf Metriken folgen Vorschläge zu Bad Smells, welche auf Qualitätsmängel in der Ontologie hindeuten können. Diese Bad Smells sind dabei inspiriert durch verwandte Arbeiten aus den Bereichen von Quellcode-Refactorings, Semantic Wikis und allgemeinen Ontologien.

Nach der systematischen Analyse erfolgen Vorschläge zur systematischen Verbesserung der erstellten Ontologie. Diese basieren auf vorhandene Arbeiten aus den Bereichen der allgemeinen Ontologien und Quellcode-Refactorings. In Bezug auf allgemeine Ontologien lassen sich drei verschiedene Aktivitäten zur Verbesserung unterscheiden, nämlich refine, prune und refactor. Ein aufgestellter Pruning-Algorithmus filtert irrelevante Informationen aus der extrahierten Ontologie. Der in dieser Arbeit beschriebene Pruning-Algorithmus entspricht einer angepassten Version eines bereits vorhandenen Ansatzes für allgemeine Ontologien. Zuletzt werden Refactorings beschrieben, mit deren Hilfe die Struktur der Ontologie verbessert werden kann. Die Refinement-Aktivität wird in dieser Arbeit nicht explizit behandelt und wird nur in Bezug zu verwandten Arbeiten diskutiert.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Ontologies serve as collections of information. Different interpretations exist about what the term ontology really means [1]. In this thesis we use the term ontology as a synonym for a specific knowledge base. An ontology contains information, which corresponds to explicitly stated relations. These relations are available in a structured format, such as RDF, which can be accessed through certain APIs. The available knowledge in an ontology corresponds to the explicitly stated relations and further derivable implications. A relation may state that an element has the name 'Programming languages developed in 2010'. Its name implies that it is a container, where programming languages are placed, that have been developed in the year 2010.

Examples of ontologies in the information technology domain are megamodels capturing the linguistic architecture of software products or underlying technologies [2]. A software product's dependencies, input, and output are gathered. Such an ontology may be used to teach software technologies to undergraduates since it makes the available information about a technology explicit through relations. For example a megamodel for 'O/X mapping with .NET' gathers all important information about the technology in a compact form.

Inserting information into an ontology (sometimes also denoted as feeding an ontology) can be realized in various ways. The first way corresponds to manual information insertion. This way needs a lot of time and contributing people. The quality of the resulting ontology is dependent on expert knowledge. An example of manual information collection is Freebase [3], which provides a platform that can be extended by its users and is accessible by provided APIs. The second way is concerned with semi-automatic information extraction. Semi-automatic information extraction uses processes, which try to identify information of interest in some available form. Text2Onto [4] tries to identify information of interest in plain text. It is not a complete automatic extraction tool, since its graphical user interface still takes human input into concern, so that a user may decide which of the proposed information pieces are added to the ontology. The

third way is a completely automatized information extraction. Completely automatized information extraction has been used, for instance in the field of web image classification [5]. The cited work extracts information from Wikipedia in order to provide improved results for image queries on the web in the animal domain. This work will later be further discussed in the related work section 2.1.

This thesis' first concern is the automatized information extraction to retrieve an ontology. Thus a target for the information extraction has to be chosen. Encyclopedias provide huge collections of information. Therefore they are a suited target for this. Wikipedia itself is currently the biggest available encyclopedia online [5, 6]. It provides several data pieces of interest. Each included concept is described in an article. Every article has a title. Its plain text can be structured by named sections. An infobox can be added to an article to summarize important information in a key value manner. The article graph consists of articles, which link to other related articles, for example in their plain text. Each article is assigned multiple categories in the category graph. These categories represent topic specific containers. Thus a taxonomic structure is provided, where categories can also be contained in multiple categories. All of the just mentioned compounds in Wikipedia may be targets for information extraction. A more detailed presentation about Wikipedia's articles and categories is given later in Chapter 2.

Wikipedia is a target of interest for many research approaches. The DBpedia-project [7, 8, 9] is an example of a large tool providing access to extracted information from Wikipedia through an ontology. Their homepage [1] and an API even provide access to a live version of DBpedia that is synchronized with the update stream capturing every change that happens at any Wikipedia page. A user can pose SPARQL queries to access information provided by DBpedia's ontology. Another example is posed by the YAGO-ontology [10], which includes the use of a lexical Database for English vocabulary called WordNet [11]. YAGO and DBpedia are samples for knowledge bases, which already retrieved their content from Wikipedia. These tools and others are presented in more detail in Section 2.1.

Some APIs also provide a more direct access to Wikipedia pages. WikipediaMiner [12] supports processing an XML dump from Wikipedia. It features a large breadth of functionality. The tool's homepage offers several demos [2] such as functions to search for articles in Wikipedia to retrieve an overview of several attributes, compare articles concerning their relatedness and annotate a text, in which concepts may be identified that have a corresponding article.

Another possibility to access Wikipedia is by using the Java Wikipedia API (Bliki engine).[3] This API provides helper classes for the Wikipedia API to download wiki texts and offers some parser classes to support an extraction

[1] `http://wiki.dbpedia.org/DBpediaLive`

[2] `http://wikipedia-miner.cms.waikato.ac.nz/demos/`

[3] `https://bitbucket.org/axelclk/info.bliki.wiki/wiki/Home`

process. The Wikipedia API is accessible by a parameterized link [4]. Without any parameters, the starting link presents an overview of the available parameters. In Section 2.2 we will discuss our chosen way to access Wikipedia.

This thesis aims at the retrieval of an ontology in the domain of computer languages. Thus we begin our extraction at the root category 'Computer language'. Our main target is the category graph, which provides a rich taxonomy in this domain. In Section 2.2 we also describe which specific elements we include in an ontology in detail.

The main goal of this thesis is a quality concerned analysis of the retrieved ontology. While the quality of an ontology is discussed in general by several existing approaches [13], we try to apply a known idea from the field of software engineering as a quality analysis. The notion of 'bad smells' was introduced by Fowler et al. [14]. In his work he describes ways to improve the design of existing code through an application of refactorings. His book contains a catalog containing bad smells, which try to describe situations in which code can be improved. Duplicated code represents a bad smell. If a code snippet exists in two methods of the same class, it hints at the possibility of applying a refactoring, e.g. **Extract Method**. This refactoring takes the duplicated code into concern and puts it into a separated method in the same class. After the duplicated code was put there, the duplicated code can be replaced by a method call to the new method. In this thesis we try to adapt the ideas of bad smells and refactorings, which already exist for code and are common in Development IDEs, to improve an ontology.

After identifying quality issues we differ between two kinds of improvements. On the one hand refactorings exist to improve the ontology's structure. A refactoring as described by Fowler always preserves the semantics. Therefore the contained knowledge in the ontology is preserved as well.

In an ontology not only structural aspects are improvable. The semantics of an ontology can also be improved. Baumeister et al. [15, 16] identify several bad smells concerning the semantics of an ontology like the existence of cycles. The removal of a cycle is not semantics preserving. Therefore we use pruning as improvements to the ontology's knowledge.

In order to motivate certain bad smells and refactorings, we consider an analysis based on metrics. Several research approaches exist, which try to introduce and apply metrics to ontologies. On the one hand we also consider available known metrics in software engineering [17, 18] and try to transfer them to ontologies. An example for transferable metrics is **Depth of Inheritance**, which is concerned with the length of the longest path from a category through subcategories to an article. An average value is calculated as well. On the other hand we take existing metrics for ontologies [19, 20] into consideration and try to

[4] `http://en.wikipedia.org/w/api.php?`

adapt them to our ontology. For example the **Edge Node Ratio (ENR)** gives the ratio between the number of edges and the number of nodes. We adapt this metric to compute the ratio in categories between the number of subcategories and the number of supercategories.

All metric and smell analyses can be executed with a prototypical demo tool available at a GitHub repository [5]. We provide an extracted triplestore for the root category of the computer languages domain. Each bad smell is implemented as a SPARQL query in an own file, which can be tried out in our prototypical graphical user interface. Prunings and refactorings are also available in the demo tool, which are guided by the graphical user interface.

The following questions correspond to a guide through this thesis.

1. How can we access Wikipedia's articles and category-graph in a suitable manner for further research?
2. What information has to be considered to retrieve an ontology in the domain of computer languages from Wikipedia? How can we extract this information?
3. The first main research question is: 'Can we analyze the quality of a retrieved ontology by using well established approaches from the field of software engineering?'. This question can be split into the following guiding parts.
   (a) Can we adapt known metrics to analyze our ontology?
   (b) Can we identify common structural issues in the retrieved ontology?
   (c) Can we identify common semantic issues in the retrieved ontology?
4. The second main research question is: 'After a firm analysis can we improve the retrieved ontology in a systematic way?'. Thus the following guiding questions to our chosen approach can be derived.
   (a) Can we formulate refactorings to improve the ontology's structure?
   (b) Can we formulate prunings to improve the ontology's semantics?

The thesis is structured as follows. Chapter 2 tries to answer the first and the second question. It starts by giving a more detailed overview of Wikipedia's content and extractable information pieces. In Section 2.1 we look at related work in the field of information extraction with a focus on information extraction from Wikipedia. Section 2.2 then compares possibilities to access Wikipedia and discusses our own decisions and implementations in this context. Chapter 3 tries to answer the third question. First it motivates qualitative ontology analysis by describing existing mechanisms in Wikipedia for quality assurance. Then

[5] `https://github.com/MarcelH91/ReviseMiniCLOntology`

in Subsection 3.1 several related approaches are summarized, which formulate metrics and bad smells for ontologies. With the metrics from related work as a template, the metrics we explain in Section 3.2 were formulated. Afterwards Section 3.3 presents bad smells in a systematic manner. After an analysis we start discussing how to improve an ontology in Chapter 4. In Section 4.1 we give an overview of existing approaches which improve an ontology. The stated existing approaches inspired us in formulating a pruning algorithm in Section 4.2 and refactorings in Section 4.3. At the end Chapter 5 summarizes overall results.

Chapter 2

Information extraction from Wikipedia

Wikipedia has been a target of several research approaches in the field of information extraction, which target its various types of pages. One type of page is an article. An article describes one entity and consists of several structured information pieces as well as plain text. While an article's plain text also contains a lot of information, it is not considered in this thesis. Instead we want to focus on available structured information. Structured information can be found in specific markup structures. The MediaWiki software uses Wiki markup [1] as a markup language to support formatting a page. In the following paragraphs we name all the possible structured targets for an information extraction from Wikipedia pages starting with the category graph. All of the observations, which serve as examples, are based on the version available in March 2015.

The category graph contains categories and articles as nodes and primary taxonomic relations as edges. A category can contain multiple subcategories and articles. Thus it represents a labeled topic specific container. Other relations may also be expressed by such a containment relation. Therefore we will try to explain the most important ones in the course of this section.

Category pages, such as the page for the category 'Computer languages' [2], have to be created by an editor first. After they were created an article or other categories can be assigned to this category by adding a specific markup in their source text. Figure 2.1 poses an example showing the links to category pages, which are annotated in the SQL article's source text. These formatted links to the category pages can be found at the bottom of the SQL article's page.

[1] `http://en.wikipedia.org/wiki/Help:Wiki_markup`
[2] `http://en.wikipedia.org/wiki/Category:Computer_languages`

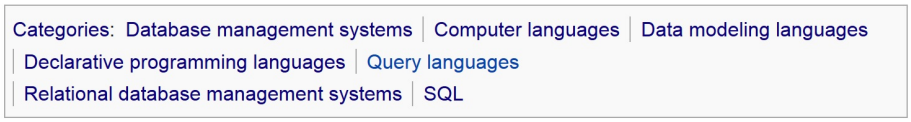

Figure 2.1: Picture of SQL article's categories.

The relation between an article and a category is created by writing the category's name with a specific Wiki markup at the end of the article's source text. Each annotation starts with two opening squared brackets and the keyword 'Category:'. Then follows the category's name and an optional pipe symbol. Everything contained in the source code from after the optional pipe symbol to before two closing squared brackets is ignored in the formatted version. The two closing squared brackets mark the end of one category annotation. Listing 2.1 shows an example for these lines in the original Wiki markup from the SQL article.

```
1 [[Category:Articles with example SQL code]]
2 [[Category:Computer languages]]
3 [[Category:Data modeling languages]]
4 [[Category:Declarative programming languages]]
5 [[Category:Query languages]]
6 [[Category:Relational database management systems]]
7 [[Category:SQL| ]]
8 [[Category:Requests for audio pronunciation (English)]]
```

Listing 2.1: Wiki markup for the SQL article's categories.

One may observe that the list of categories in listing 2.1 is different from those in figure 2.1. There are several reasons for this difference, which we want to use to explain several types of Wikipedia categories in more detail.

The primary idea of the category graph is to provide a hierarchy. This idea is also stated at Wikipedia's page about categorization at `http://en.wikipedia.org/wiki/Wikipedia:Categorization`. Thus the category graph seems a feasible target for the extraction of taxonomic information, such as in [21]. Help pages, such as `http://en.wikipedia.org/wiki/Help:Category`, often provide guides to editors on how to use specific concepts on Wikipedia pages.

The markup in listing 2.1 contains the categories 'Articles with example SQL code' and 'Requests for audio pronunciation (English)'. These categories are hidden categories. They do not appear in the formatted version displayed in a browser. A category is declared as a hidden category by adding a special annotation to the category's page. In the case of 'Articles with example SQL code' the tag '{{hidden category}}' was used to mark this category as hidden. Another possibility is present in 'Requests for audio pronunciation (English)',

where '__HIDDENCAT__' is used instead. With this markup, the category is automatically added as a subcategory to a special maintenance category containing all hidden categories [3].

One further aspect, which makes an information extraction from the markup more difficult, is the concept of **transclusion** described at `http://en.wikipedia.org/wiki/Wikipedia:Transclusion`. Wikipedia supports a partial inclusion of template pages into general pages. Template pages are used to collect information, for instance boilerplate text, infoboxes and navigation boxes [4]. They can be included in another page by using the markup '{{templatename}}', where 'templatename' may be replaced for example by 'Databases' referencing to the template named 'Databases' [5]. By using special tags, such as '<noinclude>', '<onlyinclude>' and '<includeonly>' in the source text of a template, an editor is able to specify what should be included or excluded. Figure 2.1 contains the category 'Database management systems', which is completely absent in listing 2.1. A reference '{{Databases}}' appears in a previous line. This transcluded template page contains the corresponding category markup for 'Database management systems' as '[[Category:Database management systems]]'. Because this category annotation exists in the transcluded page, the SQL article is also contained in the category 'Database management systems'.

The article named 'SQL' is contained in a category with the same name. This leads to the conclusion that the category called 'SQL' is an eponymous category [6]. An eponymous category covers the same topic as an article. The noted Wikipedia page states that it typically contains related categories mentioned in the corresponding article. Thus instead of providing taxonomic relation it includes entities with strong relatedness.

Another type of category is a non-diffusing subcategory [7]. This type cannot be observed in the SQL example. This type of category is suited for the following situation. There is a subcategory which corresponds to a subset of a category expressing special characteristics of interest in the contained pages. All pages contained in this non-diffusing subcategory may also be contained in its supercategory. Thus the subcategory does not continue the taxonomic hierarchy. Non-diffusing subcategories have to be annotated with 'Non-diffusing subcategory' or the supercategory is annotated with 'All included'. While this annotation seems to be unused in our chosen Wikipedia pages, one has to keep in mind that this annotation may be missing for a diffusing subcategory.

Every category may have a main article, which further defines the topic related to all contained pages. For example 'Computer language' is the main

[3] `http://en.wikipedia.org/wiki/Category:Hidden_categories`
[4] `http://en.wikipedia.org/wiki/Wikipedia:WikiProject_Computing/Templates`
[5] `http://en.wikipedia.org/wiki/Template:Databases`
[6] `http://en.wikipedia.org/wiki/Wikipedia:Categorization#Eponymous_categories`
[7] `http://en.wikipedia.org/wiki/Wikipedia:Categorization#Non-diffusing_subcategories`

article of the category 'Computer languages'. A main article is added to the category page by adding tags for instance '{{Cat more|Computer language}}' in the source text. Figure 2.2 displays the formatted version presented on the category page.

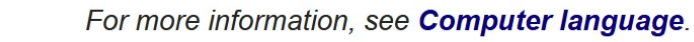

Figure 2.2: Automatised formatting of a category's reference to its main article

Other possibilities for annotations can be found in the XML category as 'Cat main|XML' or in the PostgreSQL category as 'Cat main'. The latter does not name the referenced article since the name of the category and the name of the main article are completely the same. The category SQL poses another example since no additional annotation is written, which marks the article SQL as its main article. Instead the category page contains a text line linking to the article presented in listing 2.2. The link is realized with an annotation with double squared brackets containing the article name '[[SQL]]'.

```
1 '''Structured Query Language''' ('''[[SQL]]''') is one of the most
    popular [[computer language]]s used to create, modify and query
     [[database]]s.
```

Listing 2.2: Additional lines on the page of the SQL category.

Article pages provide several interesting information pieces, which may be considered for an extraction. Every article has a name, which serves as its unique identifier. The plain text starts off with an introductory text. This text may contain the first valuable information. For example the help page for Wiki markup starts by stating two synonyms for Wiki markup namely 'wikitext' and 'wikicode'. A synonym relation is extractable.

Every article is structured further by sections, that each have a name as well. The contained plain text may have links to other pages. Links to articles are common in all types of pages. The article graph primarily considers articles and their links to other articles. For example the article about SPARQL [8] contains the following partial sentence in its original Wiki markup: 'SPARQL allows for a query to consist of [[triplestore|triple patterns]]'. The contained link is represented in '[[triplestore|triple patterns]]', where the part before the pipe symbol namely 'triple patterns' conforms to the text displayed in the formatted page and the part after the pipe symbol namely 'triplestore' conforms to the referenced article's name [9]. Again such a link is surrounded by double squared brackets.

[8] http://en.wikipedia.org/wiki/SPARQL
[9] http://en.wikipedia.org/wiki/Triplestore

The article graph has already been the target of many research approaches. WikipediaMiner [12] offers functionality to compute the relatedness between two articles based on their referenced articles. This functionality is also available in a demo version on their web page.

Important information concerning the discussed entity on an article page is summarized in an infobox. The infobox is always placed on the right side. It contains information in the form of key value pairs, which one may easily extract. Figure 2.3 displays the infobox from the SPARQL article. A corresponding topic name can always be found at the top. Then follow several key value pairs. In order to see better what is formatted here listing 2.3 presents the original Wiki markup for the infobox.

Figure 2.3: Picture of SPARQL article's infobox

```
{{Infobox programming language
| name                   = SPARQL
| paradigm               = [[Query language]]
| year                   = {{Start date and age|2008}}
| designer               =
| developer              = [[W3C]]
| latest_release_version = 1.1
| latest_release_date    = {{Start date and age|2013|03|21}}
| typing                 =
| implementations        = [[Jena (framework)|Jena]],<ref name="sw-
    programming">{{cite book
 | last1 = Hebeler | first1 = John
 | last2 = Fisher | first2 = Matthew
 | last3 = Blace | first3 = Ryan
 | last4 = Perez-Lopez | first4 = Andrew
 | title = Semantic Web Programming
 | location = Indianapolis, Indiana
 | publisher = [[John Wiley & Sons]]
 | pages = 406
 | date = 2009
 | isbn = 978-0-470-41801-7 }}</ref> [[Virtuoso Universal Server|
    OpenLink Virtuoso]]<ref name="sw-programming" />
| dialects               =
| website                = {{url|http://www.w3.org/TR/sparql11-
    query/}}
}}
```

Listing 2.3: Wiki markup for the SPARQL article's infobox.

An infobox's formatted version and its markup depend on a corresponding template. A set of infobox templates is available already [10]. The first line in listing 2.3 names the used template for this infobox namely 'Infobox programming language'. This infobox's corresponding template can be inspected at the conforming Wiki page [11].

Several observations can be made in the exemplary Wiki markup for this infobox. Wiki markup offers some data types, such as dates in the value of 'Stable release' namely '{{Start date and age|2013|03|21}}'. This mark up is formatted to a text, where the month is written in letters and the time is computed and displayed, which passed by until the current date. DBpedia [7] transforms the extracted Wiki markup itself and converts the source text into a human readable form.

Listing 2.3 demonstrates the possibility to add reusable citations annotating an attribute's value. A citation's definition and its reuse are presented in the values of the attribute with the key 'implementations'. In the example the citation is defined starting with the markup '{{cite book' and several key value pairs posing more information about the cited work. The created citation is

[10] http://en.wikipedia.org/wiki/Wikipedia:WikiProject_Computing/Templates#Infoboxes

[11] http://en.wikipedia.org/wiki/Template:Infobox_programming_language

contained in a HTML structure, where the opening tag is '<ref name="sw-programming">' and the closing tag is '</ref>'. After this citation is defined and made referencable through those HTML tags, it is reused for 'OpenLink Virtuoso' by putting the tag '<ref name="sw-programming" />' after the corresponding Wiki markup link. Citations also conform to templates defined at a specific Wikipedia page [12].

One article can have multiple infoboxes. Such articles cover various concerns, where each concern is summarized in an own infobox. For example the article named 'ARM architecture' has four infoboxes. The first infobox is concerned with general information about the family of ARM architectures and the others cover specific architectures. This points to the information that multiple topics may be covered by a single article. Wikipedia's guidelines state when to create a standalone page and when information should be put into a section of an already created page [13]. The understandability of a description for one concern is prioritized over a separation of concern. Therefore the guidelines state that sometimes it is better to put a topic into the context of another existing article.

2.1 Related Work on information extraction

After naming targets for an information extraction from Wikipedia this section describes related work in this area. The first couple of projects provide access to knowledge extracted from Wikipedia through a derived ontology. These approaches have their own extraction mechanisms, which fed the extracted data into a queryable ontology.

DBpedia: DBpedia is a project based on community effort, that has the goal to extract structured information from Wikipedia and make the extracted information accessible on the web [7]. Wikipedia content is converted into a rich multi-domain knowledge base. Two kinds of access to DBpedia exist. The older version is based on a previously retrieved dump from Wikipedia [14]. DBpedia-live presents another option [9]. If the live feature is used the DBpedia knowledge base reflects the current state of Wikipedia. This is realized by accessing Wikipedia's live article update feed. In this version current updates to Wikipedia are processed, saved in specific triplestores to log the processed updates, and integrated into the DBpedia knowledge base by replacing the obsolete version with updates.
DBpedia generally targets several information pieces through its implemented extractors. Each extractor targets one specific element. One targets the infobox of an article. DBpedia's infobox extractor processes Wikipedia's defined data

[12] http://en.wikipedia.org/wiki/Wikipedia:Citation_templates
[13] http://en.wikipedia.org/wiki/Wikipedia:Notability#Whether_to_create_standalone_pages
[14] http://dumps.wikimedia.org/

types and may use a mapping extraction, which maps the value types used in Wikipedia infoboxes to DBpedia's own value types. This infobox extractor also detects synonymous attribute keys and considers them in the resulting knowledge base. Further exemplary targets are the category graph and the article graph. Published work includes descriptions to nineteen different extractors in total and their mapping into DBpedia resources.

YAGO: YAGO presents an ontology automatically derived from Wikipedia and WordNet [10]. The extracted ontology is based on a thorough and expressive data model that slightly extends RDFS. Wikipedia's category graph is combined with taxonomic information from the electronic lexical database for English named WordNet [11] to form an Is-A hierarchy. Further semantic relations are created based on extracted information from Wikipedia's infoboxes. It differs between entities, which correspond to persons, such as 'Elvis', and concepts, such as 'rockSinger'. The information from the category graph provides them with candidates for entities that conform to articles and concepts that conform to categories. These candidates are checked against the more clean taxonomy provided by WordNet. A further type checking process verifies the plausibility of newly extracted data.
YAGO extracts the information by parsing an XML dump of Wikipedia and applying four different types of heuristics to articles. Infobox heuristics consider the key value pairs in infoboxes. An original key value pair may be transformed into one of 170 available target relations in the YAGO model. These relations are based on attributes, which were identified to appear highly frequent. Type heuristics establish a concept for each entity by using the category graph and WordNet synsets and hyponymy (type-of) relations. Word heuristics establish knowledge about the meaning of a word. While WordNet reveals synonyms through its synsets, Wikipedia's redirects are used to extract further information. Person names are also detected and integrated via specific parsers. Category heuristics retrieve further semantical relations by exploiting category names.

Further projects, which try to make information more accessible are Freebase [3] and Cyc [15]. Both present manually crafted ontologies. Cyc also provides a reasoning engine and claims to be the world's largest and most complete general knowledge base. Several published research papers exist, which cover Cyc and its specific platforms, such as OpenCyc and more [22, 23]. One specific approach shows how resources from Cyc can be mapped to Wikipedia's resources [24].

After presenting approaches, which try to make the information from Wikipedia accessible by offering access to an extracted ontology, we present tools that provide mining functionality to access Wikipedia in a more direct manner.

[15] `http://www.cyc.com/`

WikipediaMiner: WikipediaMiner is a platform for sharing data mining techniques [12]. It also works on an XML dump from Wikipedia, which is converted in a run-once extraction process. The toolkit creates a database environment containing summarized versions of Wikipedia's content and structure. A Java API simplifies access to Wikipedia articles, categories or redirects etc., which are represented by Java classes. WikipediaMiner further supports several types of functionality. It may be used to compare articles and terms. Secondly it provides annotation functions for topic detection, disambiguation, link detection and document tagging. Further services capture its capabilities by including possibilities to search, suggest, explore articles, explore categories, compare and annotate. Some services can already be explored by users in a demo version, which was put online at WikipediaMiner's homepage [16].

Java Wikipedia Library (JWPL): JWPL is an alternative to WikipediaMiner [25]. It also provides direct mining capabilities for Wikipedia in a similar manner. JWPL operates on an optimized database that has to be created in a one time effort from an XML dump. Its focus is different from WikipediaMiner since in JWPL all tasks are strongly related to Natural Language Processing (NLP). Java classes represent pages and categories. The term page is used here for Wikipedia articles, disambiguation pages and redirects. A page object provides access to the article text with or without markup, assigned categories, and simplified access to relations from the article graph, such as ingoing links from other pages and outgoing links to other pages. Newer versions of JWPL, which can be viewed in corresponding Java documentations also provide more fine grained access to structural elements, such as sections and their titles.

In the previous paragraphs we described existing tools and approaches targeting Wikipedia. In the next paragraphs we also want to name some further research in this area.

A different way to make use of a Wikipedia article's structure and provided information is proposed by Wang et al. [5]. They extract information from Wikipedia articles to improve web image classification in the animal domain. In essence they demonstrate a possibility to improve the quality of search results for animal related image queries. Therefore they name the following three process steps. A first step considers the category graph for taxonomic information and infoboxes for further scientific classifications. The second step takes advantage of an article's structure by retrieving mentioned synonyms and retrieving further semantic relations from the rest of the text. An article is structured by sections, which have section titles. These section titles are used to propose candidates for relation names. Only terms mentioned in the section that have an own page are considered candidates for the relation's target. The rest of the text is believed to be of trivial importance. Since not all candidates may be correct an association rule mining and quality measure are used to filter bad candidates. Implied

[16] `http://wikipedia-miner.cms.waikato.ac.nz/`

information is considered as well to gain further knowledge. The third and last step is concerned with the visual part based on Support Vector Machines, which are used for trained classifiers according to global and local features.

Wikipedia's rich content structure may also be used for several other purposes, such as the retrieval of an Is-A hierarchy from category labels [21], identification of topics and concepts in plain text [26], constructing ontologies in specific domains [27] possibly with a specific focus on infoboxes [28] or document clustering [29].

2.2 Implementing information extraction

This section presents our decisions concerning how we access Wikipedia and what information is extracted. Therefore we first discuss our choice on using the Java Wikipedia API (Bliki engine) [17]. Afterwards we name all targets to our extraction and give a brief process description of the implemented extraction process.

Most tools described in the previous section work on Wikipedia dumps. In order to use them, a Wikipedia dump and CSV-summaries may have to be retrieved, for example from Wikimedia's homepage [18]. A complete dump has a large size. A proportion is also given on the JWPL-project's homepage [19]. JWPL works on optimized databases of a size of around 158GB (126GB for the data and 32GB for the indexes). The primary advantage named for downloading and working on a dump is the reproducibility of results [25].

Our primary interest lies in formulating abstract bad smells plus corresponding improvements and describing an exemplary implementation. Therefore we decided to use the Bliki engine in our implementation instead of Wikipedia dumps in analogy to [30]. A Wikipedia dump represents Wikipedia's complete state at the time of the extraction. Since we only want to access the domain of computer languages we do not need all pages and articles. Our proposed tool can extract an ontology to work with in about thirty minutes. Then a user can try out any of the exemplary implementations to detect bad smells and improve an ontology representing the current state of Wikipedia's computer language domain.

Our extraction process uses the Wikipedia API, which can be accessed through HTTP requests supported through the Bliki engine to `http://en.wikipedia.org/w/api.php?`. The referenced homepage already provides information of how to pose requests to the Wikipedia API. Bliki's parsers help retrieving the wanted information from the returned raw XML-text. An example is posed by the link from the following URL, which returns the infobox and

[17] `https://bitbucket.org/axelclk/info.bliki.wiki/wiki/Home`
[18] `http://dumps.wikimedia.org/enwiki/20150205/`
[19] `https://code.google.com/p/jwpl/wiki/JWPLDocumentation`

the summary at the beginning of article 'Fortran':
`http://en.wikipedia.org/w/api.php?action=query&prop=revisions&rvprop=content&format=xmlfm&titles=fortran&rvsection=0`.

The extraction process saves the retrieved data in a temporary model. Its schema contains four main classes namely `Category`, `Entity`, `AttributeSet` and `Attribute`. A name is assigned to each instance of these classes at the field `name` contained in the abstract superclass `NamedElement`. A Wikipedia category is mapped to an instance of the class `Category`. It has three collections. One collection includes all subcategory members that correspond to processed `Category`-objects. The contained articles are represented in a collection of `Entity`-objects. If the category has a main article, the corresponding entity is saved as well. For `Category`-objects and `Entity`-objects a Set of Strings `supercategories` also includes the containing supercategories' names. This set is placed in the abstract class `Element`, which is specialized by the classes `Category` and `Entity`. `Entity`-objects have a collection of `AttributeSet`-objects. An article's infobox is mapped to an `AttributeSet`-object. Its assigned value in `name` corresponds to the name of the used infobox template, such as 'programming language' or 'software'. All of the infobox's key-value pairs, where the value is not empty in the source text are saved in the fields `name` and `value` of `Attribute`-objects that are assigned to the infobox's corresponding `Attributeset`-object. The described schema is displayed in figure 2.4.

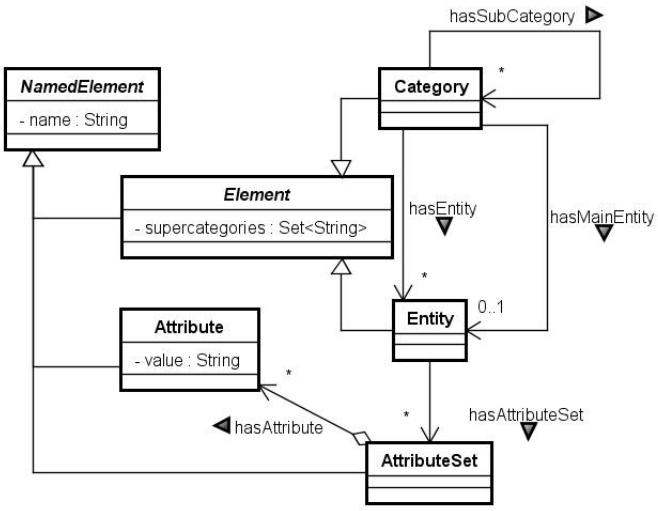

Figure 2.4: The schema for the extraction

After describing the temporary model we present a process description for our extraction process. The extraction process includes multiple threads. A main process is present in the class `CrawlerManager`. It contains multiple col-

lections to guide the overall process and to keep track of the Wikipedia elements that have already been processed.

The initialization phase includes the following steps. At first the main process initializes a list of excluded categories `excludedCategories`. A rationale to why some categories should be excluded is given later. In the second step a thread pool with a fixed number of threads is initialized. To efficiently utilize the CPU, it creates as many threads as the number of available cores. A field called `threadcounter` is used to tell how many of the threads are active. In the initialization phase it is set to zero. As the last step the process assigns the name of the root category to a new `Category`-object and adds it to `categoryQueue`, which collects the `Category`-objects that should be processed next.

If a category is added to `categoryQueue` the managing process takes and removes it from the queue, increases `threadcounter` by one and starts a new `Crawler`-thread that is supposed to process the category given as a parameter. The complete process stops only if no category is available in `categoryQueue` and no `Crawler`-thread is running anymore. The process description concerned with the `CrawlerManager`-process is displayed in figure 2.5.

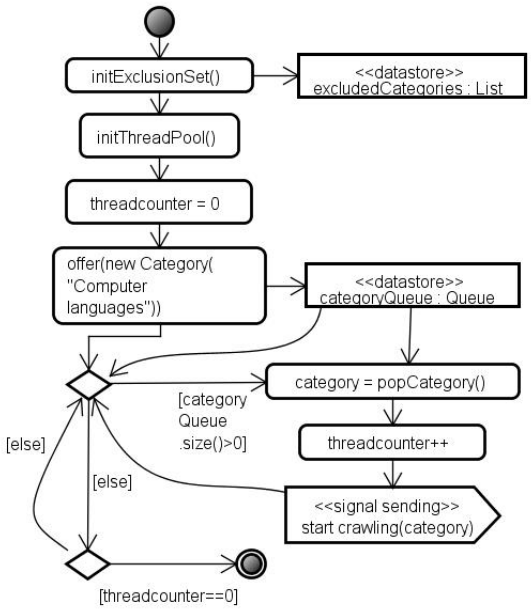

Figure 2.5: The `CrawlerManager` process

A `Crawler`-thread processes one `Category`-object. In order to keep track of what has been processed already the `CrawlerManager`-object contains maps for categories and entities where the names are mapped to the objects created by a `Crawler`-thread.

A thread starts by extracting the name of the category's main article. If

the name was processed already the corresponding entity is retrieved from the `CrawlerManager`-object's map. Otherwise a new entity is created. This `Entity`-object is assigned to the `Category`-object that is being processed.

In the second phase a thread retrieves the names of the supercategories and saves them in the corresponding collection.

Next the third phase is concerned with the subcategories. Each subcategory's name that was not processed and is not in `excludedCategories` is offered to `categoryQueue` in `CrawlerManager` and then added to the processed categories.

The fourth phase is concerned with processing the article members in the category that were not processed already. From each article the infobox is retrieved and converted into an `AttributeSet`-object and its contained `Attribute`-members. Existing markup in values is filtered. A value in an infobox may have multiple elements separated by a comma. For example the infobox of 'SQL' contains '[[Static typing|Static]], [[strong typing|strong]]' as the value for 'typing'. First the markup is removed converting it to the String as it is formatted by a browser 'Static, strong'. Then the comma is recognized and two attributes are created that have the same assigned name. The first one has the value 'Static' and the second one has the value 'strong'. Afterwards the entity's containing categories' names are saved in `supercategories` and the corresponding entity is added to the processed entities.

Before a `Crawler`-thread terminates, it decreases `threadcounter` in the `CrawlerManager`-object.

Figure 2.6 shows the corresponding process description. We used UML's visual notation for an expansion region to show that the contained activities are performed for each input element. The first expansion region has subcategories as input and the second takes articles that were extracted in the previous activity.

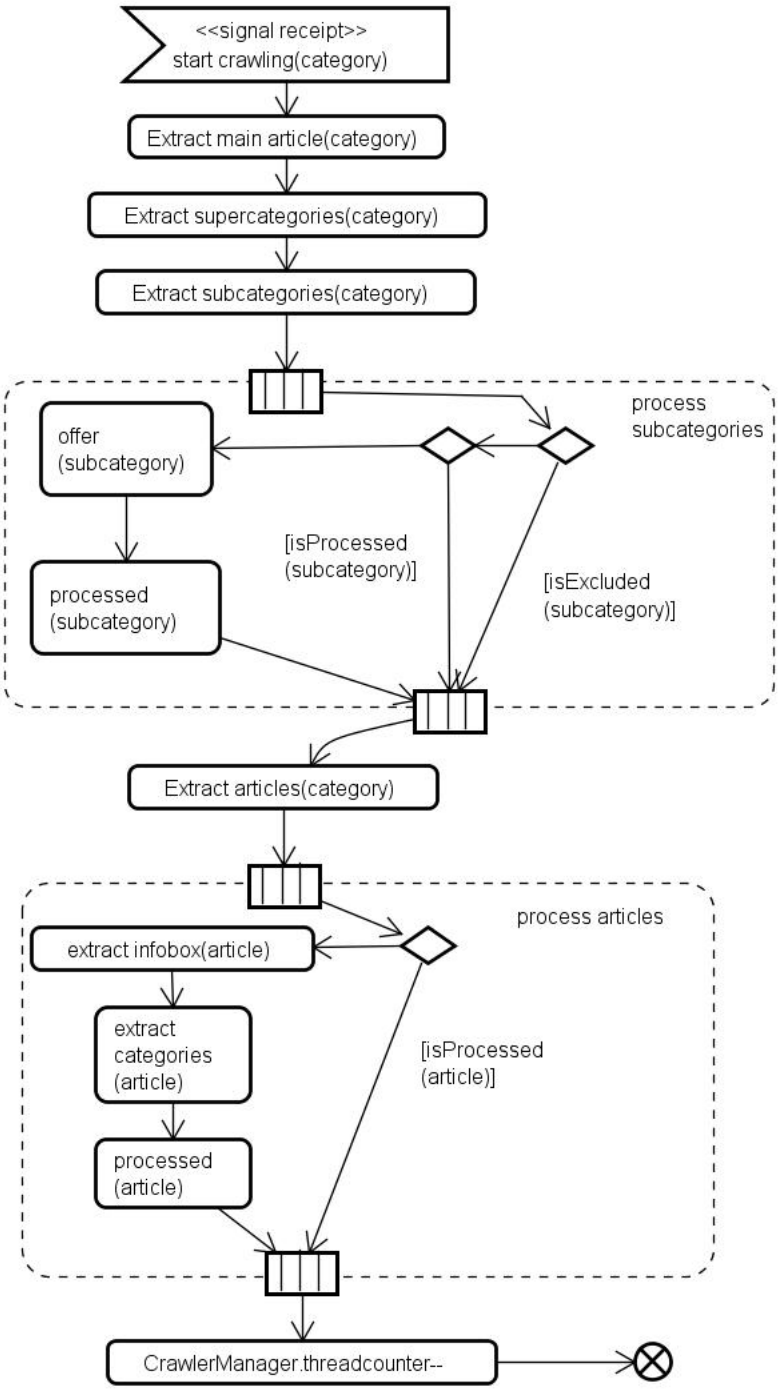

Figure 2.6: The `Crawler` process

A transformation process takes the temporary model as input and returns a data set containing RDF triples. All of our extracted information is saved in a TDB triplestore [20]. The listing 2.4 contains exemplary triples saved in the triplestore. This sample also shows the chosen resource URIs, such as resource URI for the root category. For the sake of readability we use the shortform 'clonto:' instead of the complete prefix for each URI 'http://myCLOnto.de/' and put semicolons surrounding the relation name.

```
clonto:Category#0 ; clonto:name ; "Computer languages"
clonto:Category#0 ; clonto:hasMainEntity ; clonto:Entity#0
clonto:Category#0 ; clonto:hasEntity ; clonto:Entity#1
clonto:Category#0 ; clonto:hasSubCategory ; clonto:Category#1

clonto:Entity#1 ; clonto:name ; "SQL"
clonto:Entity#1 ; clonto:hasAttributeSet ; clonto:AttributeSet#0

clonto:AttributeSet#0 ; clonto:name ; "0"
clonto:AttributeSet#0 ; clonto:topic ; "programming language"
clonto:AttributeSet#0 ; clonto:hasAttribute ; clonto:Attribute#0

clonto:Attribute#0 ; clonto:name ; "year"
clonto:Attribute#0 ; clonto:value ; "1974"
```

Listing 2.4: Exemplary data from the triplestore.

All resources and relations start with the prefix 'http://myCLOnto.de/'. Then follows the key word for either the element's type, such as **Category**, **Entity**, **AttributeSet** or **Attribute**, or a relation's name, such as **hasEntity**, **hasSubCategory**, **hasAttributeSet**, **hasAttribute**, **name**, **topic** or **value**. Resources contain a diamond as a symbol for separation in their name. After the type's name follows the actual elements index to provide unique identification. Since one may want to rename an element the URI should not contain an element's assigned name. The relation names are exactly the same as in figure 2.4. Several relations are not in analogy to this figure. Supercategories are no longer saved as String values. The transformation process creates a distinct resource for each supercategory. Instead of adding has-supercategory relations we chose to add has-subcategory relations for supercategories of categories and has-entity relations denoting an entity containment in a category. An attribute-set is also not completely analogous to the figure. Its value for the name relation no longer contains the topic (infobox template name). Instead its id from the actual resource is given in order to provide at least one property that may serve as a unique identifier. The topic is now saved separately in a topic relation.

We decided to take the category 'Computer language' [21] as the root for an exemplary extraction. This root category was chosen since the thesis extends

[20] http://jena.apache.org/documentation/tdb/
[21] http://en.wikipedia.org/wiki/Category:Computer_languages

previous work [30], which also analyzed the general taxonomic relations in this domain.

In a first experimental run the extraction process did not terminate after two hours. Our observations revealed that there is a way through has-subcategory relations from the category 'Computer languages' to 'Government of Nazi Germany'. By tracking the necessary paths we found has-subcategory relations to topics that in our opinion do not belong to the field of computer languages. One questionable transitive subcategory is 'Statistical data types' that is more concerned with statistical analysis than the domain of computer languages.

By further observations we were able to identify several categories, which took us to other categories that are not related enough to our target domain. These categories were added to `excludedCategories` in the `CrawlerManager`. After several trials the categories 'Statistical data types', 'Internet search engines', 'Instant messaging', 'Internet search', 'Google services', 'Net-centric', 'Service-oriented architecture-related products', 'Programming language topic stubs' and 'User BASIC' were excluded. 'Programming language topic stubs' and 'User BASIC' pose examples of maintenance categories that are not hidden. In our opinion the rest of the named categories do not belong into the domain of computer languages. Further categories can be removed in the analysis of the extracted ontology, which reveals other semantically distant categories.

In order to provide ground for the upcoming analysis we discuss the semantics of our triplestore's elements and relations. An ontology may need different semantics from the ones used at Wikipedia itself. A category serves as a topic container. An optimal containment relation suggests an Is-a relation between a member's name and the category's name. The weakest derivable relation should be a strong relatedness. A has-subcategory relation is transitive. Thus every category that is reachable through has-subcategory relations from the root category is its subcategory. This leads to further semantics for the has-entity relation. An entity contained in a category is a member of the category's supercategories as well. Thus there does not have to be an explicit relation in the triplestore expressing the containment in a supercategory.

In our analysis several alternatives concerning improvements are given. The final decision is left to an ontology engineer. For example an eponymous category poses a special case. An ontology engineer may either decide that it is not a real category and removes it or he leaves them in the triplestore to not loose the information about strongly related categories and entities.

Chapter 3

Analysing extracted information

Before describing our own methods to analyze the extracted ontology from Wikipedia, we want to motivate the topic by describing existing quality assurance in Wikipedia. Wikipedia's information quality is assured through several processes that were analyzed, for instance in [6]. Special editors exist, whose job is to evaluate the quality of an article by modifying, deleting or changing its status. An article may have different kinds of flaws. In the next paragraph we want to give several examples of lacks of quality, that can be detected.

Wikipedia provides a few templates, which are formatted as message boxes at the top of an article. These inform a reader about apparent quality issues. An article is marked as flawed based on several observations. For example, the article of 'Apache Empire-db' was annotated with '{{Unreferenced|date=May 2011}}' as an article that does not cite or reference any sources. The corresponding formatted box is displayed in figure 3.1.

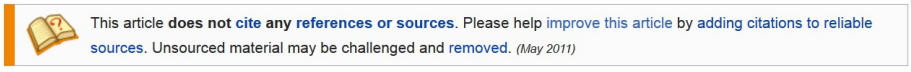

Figure 3.1: Missing references

Quality in Wikipedia is also assured through several written bots [1]. Flaws, such as missing references, could also be detected by bots. A bot could measure the number of references in an article. If it is equal to zero or very low in relation to the text's size it could already be a sign for bad quality in the inspected article. Therefore this kind of measure could also be seen as a metric for an article's quality.

[1] `http://en.wikipedia.org/wiki/Wikipedia:Bots`

Another example is given by the category 'XML'. This category is annotated with the tag '{{CatDiffuse}}', which is formatted to a box telling the articles in this category should be moved to its subcategories. The box is displayed in figure 3.2.

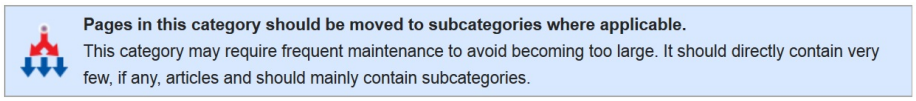

Figure 3.2: Category needs further diffusion

The fact, that a category's articles should be pushed into its subcategories, is also based on some measure. A category should not become too large. Thus categories with a lot of members already point to this lack of quality especially if they possess several subcategories. A metric might pick up this kind of reasoning and hint at categories with too many articles, which can be pushed into subcategories.

An analogy can be drawn to the bad smells proposed by Fowler [14]. Bad smells are observations sometimes based on patterns or metrics that may hint at a bad quality. The bad smell **Large Class** is based on the observation, that a class has a lot of instance variables. A class might provide too much functionality. Several instance variables can be bundled in a newly created class. There is no fixed value for the number of articles in a category, that which definitely tells, that a category needs diffusion. Categories exist where articles cannot be pushed into further subcategories, such as 'Free software programmed in C'.

The last example we want to pose can be found at the article for 'Cl4 (programming language)'. This article was marked with multiple issues containing a mark as an orphan with '{{Orphan|date=December 2013}}'. An articles is marked as an orphan, if there exist no articles that link to it. The corresponding box is shown in figure 3.3.

Figure 3.3: Article without incoming links

Detecting orphan articles could also be realized automatically, since the observation is based on a fixed pattern. When there exist no links to an article, it can be marked as an orphan. This pattern can be formulated as a bad smell,

which may either be used to analyze Wikipedia or to analyze an ontology that is also concerned with the article graph.

3.1 Related Work on quality analysis

Three types of general ontology evaluation can be distinguished [31]. The first one is a **Gold Standard Evaluation**, which compares an ontology with an established ontology, e.g. based on similarity measures as in [32]. Next, the second type **Criteria based Evaluation** is concerned with evaluations based on a defined set of criteria. Exemplary aspects for this type are consistency, completeness, conciseness, expandability and sensitivity as in [33]. At last **Task based Evaluation** evaluates the ontology's competency in completing tasks. Therefore the suitability for applications can be measured. An exemplary evaluation of this type is conducted on Wikipedia and its categories in [31].

Our analysis corresponds to a **Criteria based Evaluation**. We come up with bad smells that hint at lacks of quality for criteria, such as structural design, consistency and completeness. Since our analysis starts with a metric analysis, related work on metrics is discussed next. We start by describing metrical analysis in the field of software engineering. For every presented related work, we try to emphasize the contained ideas that we reuse and adapt to our task later.

Framework comprehension can be evaluated through reuse-related metrics for the implemented design and usage of frameworks [18]. A specific analysis is conducted for the Microsoft .Net Framework. Exemplary metrics here are the measures for **MAX size class tree** and **MAX size interface tree**. They determine the largest class or interface inheritance tree in terms of the node count. These measure the use of polymorphism encouraging reuse in client code.

Many different metrics can be used to measure the quality of the object oriented design of a system. A survey on object oriented metrics is presented by Jamali [17]. It identifies five characteristics of object oriented metrics, such as localization, encapsulation, information hiding, inheritance and object abstraction. Based on these characteristics nine classes of object oriented metrics are stated, which are implemented in several metric suites, such as **Chidamber & Kemerer's Metrics Suite**. These classes correspond to measures for size, complexity, coupling, sufficiency, completeness, cohesion, simplicity, similarity and volatility.

Metrics are also used in the field of quality evaluation of ontologies. Inspired by existing metrics from software engineering García et al wrote a survey on ontology metrics that measure the complexity of an ontology at the graph and the class level [19]. They also make a proposal for a suite of ontology metrics.

These metrics are analytically evaluated and empirically analyzed on public domain ontologies to show their characteristics and usefulness. The following list provides several metrics that are interesting for our goal as well. Here the first four metrics are concerned with the complexity at the ontology level and the rest focuses on specific classes.

Size of vocabulary: SOV measures the amount of defined vocabulary. It is the sum of the number of defined classes and individuals added to the number of defined properties.

Edge node ratio: ENR measures the relation between the mount of nodes and edges in the graph. Therefore the number of nodes is divided by the number of edges.

Tree impurity: TIP gives a degree on how far the taxonomy deviates from having a tree structure. The number of nodes in an ontology's hierarchy is subtracted from the number of edges plus one.

Number of children: NOC measures the size of the set of all immediate child classes of the targeted class.

Depth of inheritance: DOI measures the length of the longest path from a given class to the root class in the taxonomy.

Class in-degree: CID measures the number of edges whose target is the given class.

Class out-degree: COD measures the number of edges whose source is the given class.

Another approach to analyze ontologies via metrics is proposed by Gangemi et al. [34]. After proposing a metamodel for ontologies and a further model for ontology evaluation they identify and discuss three distinct types of measures, in essence structure, functionality and usability. Several exemplary measures for each type are stated and discussed. We are primarily interested in their proposals concerning the structural dimensions. Several metrics are therefore presented in the following list, that are concerned only with taxonomic relations in the whole graph.

Absolute depth: Computes the sum of the length of all paths in a graph.

Average depth: Computes the average depth of all paths. The absolute depth is divided by the number of paths.

Maximal depth: Computes the length of the longest existent path in the graph.

Absolute breadth: Measures the absolute breadth of a graph by computing the sum of paths having a leaf node at a certain depth. Thus it starts by

adding the number of leaf nodes reachable one step from the root. Then this number is added to the number of leaf nodes reachable in two steps from the root and so on.

Average breadth: Computes the average breadth of a graph by dividing the absolute breadth of the graph by the number of depth levels.

Maximal breadth: Computes the maximal number of leafs on one level.

Absolute leaf cardinality: Measures the total number of leafs in the graph.

Ratio of leaf fan-outness: Measures the ratio of leafs by dividing the absolute leaf cardinality by the total number of nodes.

Measuring modularity: A module is a subgraph in the ontology whose elements conform to the subset of the graph's elements. Two modules are disjoint, if there are zero or more Is-a edges connecting the first with the second subgraph and each edge has the same direction. With an appropriate procedure, it is possible create a modularization of a nonmodularized graph.

Further related works exist concerning the evaluation of ontologies via metrics. Some are very specific to OWL-ontologies and are therefore difficult to adapt to our kind of ontology [20]. Metrics exist that analyze the separate layers of an ontology. Specific measures analyzing the schema level and the instance level can be formulated separately [35].

Message boxes in Wikipedia already hint at possible patterns and measures to detect quality issues in an extracted ontology. Before we introduce formulated metrics and bad smells the related work on using bad smells and related concepts is described next.

Fowler et al. describe when and how to improve the design of existing code [14]. A guide to when a code refactoring should be performed is based on the concept of bad smells. In the book, the following statement concerning bad smells is made: 'In our experience no set of metrics rivals informed human intuition.'. This already points out that the answer to when to apply a refactoring may be subjective and dependent on the situation. For all bad smells one has to remember that there is no fixed pattern or fixed value for a metric, which tells a programmer that some part of the code definitely has to be structurally improved. The programmer still has to make a decision, based on his own intuition, whether a fix will really raise the code's quality.

Since this section focuses on the analysis part, the refactoring concerned aspects are covered later in Section 4.1. We especially want to name three bad smells from Fowler et al., that gave off ideas on what to analyze in our extracted information. These are presented in the following list, whose elements state each bad smell's name and a short description.

Large Class : A class has many instance variables or many long methods.

Lazy Class : A class does not provide much functionality and may be obliterated in order to reduce maintenance costs.

Speculative Generality : An abstract class that might even have a super class does not provide much functionality and can be obliterated in order to raise the code's quality.

Rosenfeld et al. describe the use of bad smell in a semantic wiki's context that may evolve in a chaotic manner [36]. Their approach helps to detect quality problems and assists users in the process of solving them. A toolbox with refactorings and a bad smell catalog is proposed. This toolbox provides instructions on how to apply refactorings to remove a smell in analogy to [14]. Each smell is described in a systematic manner. An entry has a name, a description, related refactorings and a detection mechanism. The paper also presents an implementation for the detection of the first bad smell as a SPARQL query.

The following list states the names and description of bad smells that we are interested in as well.

Twin Categories: Two categories appear together very frequently. This may either be caused by the fact, that two categories are synonyms, or they're in a has-subcategory relation.

Concept too categorized: A concept is a member of too many categories. This may expose the fact, that this concept covers more than one real concept.

Large Category: A category has too many category members. Therefore some kind of over-categorization is apparent.

A series of related work by Baumeister et al. [16, 37, 38] is concerned with the syntactic verification of ontologies. The term anomalies is used to denote a lack of quality in an inspected knowledge base. Several anomalies are stated based on previous work by Preece and Shinghal [39] and Gómez-Pérez [40].

The anomalies are categorized by the following groups. **Redundancy** captures all anomalies, where duplicate or subsuming definitions exist in the ontology. **Inconsistency** contains anomalies which describe the existence of ambivalent definitions of information in the ontology. **Circularity** denotes the existence of cycles in the knowledge base. This does not just consider taxonomic cycles in subclass relations, but rule chains and circularity between rules and taxonomy as well. **Deficiency** comprises those anomalies that do not cause errors concerned with reasoning, but affects completeness, understandability and maintainability instead. Another category is concerned with **design anomalies**, which may be more subtle. These may not directly point to errors in an ontology, but hint at problematic areas.

All of the anomalies can be detected through evaluation queries. These were implemented with FNQuery, which can be used to analyze OWL based ontologies.

The work series of Baumeister et al. was further extended by Fahad and Qadir [41], who proposed a framework for the evaluation of ontologies. In this framework they proposed several new anomalies that were not covered by Baumeister et al..

The following list states all of the anomalies we try to transfer to a bad smell for our own concern. Since we do not extract enough information to perform logical reasoning on our ontology many anomalies stated in the above named papers cannot be adapted. Therefore we identify a small set of transferable proposals. Each item in the list states the anomaly's name and a description and a citation to where it was stated.

Circular subclasses: There is a path of subclass relations such that by transitivity there exists a class that is a subclass of itself [38].

Identity: Multiple formal definitions of classes, properties or rules exist that can only be discriminated by their names [38].

Redundancy by Subsumption between Rules: There is a relation in the knowledge base whose deletion does not change the apparent knowledge. Thus one relation can be implied from the knowledge expressed by another relation [38].

Chains of Inheritance: There exists a very long path containing only subclass relations [38].

Partition Error : A class is subclass of two classes that have a common superclass [38].

Weaker domain specified by subclass error: A class has a subclass whose domain should actually be greater than the domain of the class itself [41].

Disjoint domain specified by subclass error: There exists two disjoint subclasses of one concept, where the concept and one of the subclasses are from different domains [41].

Sufficient knowledge Omission Error (SKO): All concepts in hierarchies should possess features so that an inference engine can distinguish them properly. Thus this anomaly tells that there is not enough contained knowledge about an existing concept [41].

Several papers exist that discuss a lexical pattern based analysis [42]. The hypothesis is that concept naming in ontologies can be used to analyze their structure. An analysis of graph structure and concept naming is combined. Though this approach is controversially discussed [43], the general idea of using

names to analyze the ontology is used to formulate bad smells as well, but the formulated bad smells are not inspired by specific ideas from the cited work.

Evaluations focused on more general ontologies are also available. For example several works exist that name bad practices denoted as 'pitfalls' in ontology constructions [44, 45]. Another approach searches for anti pattern in ontologies [46]. The term anti pattern is used to name bad practices that may superficially look like a solution.

3.2 Metrics

The metric analysis we present in this section serves as ground for our proposed bad smells. We analyze the ontology first and present observations based on the metric analysis that result in the formulation of several bad smells. A discussion concerning examples for metric analyses and bad smell matches is primarily based on the information available at the corresponding Wikipedia pages [2].

At first we want to describe some metrics that are concerned with the complete ontology graph. The following list contains the graph metrics we chose, where each metric is named and explained with a rationale.

#Nodes (#N): The first metric counts the total number of nodes in the graph. This number can be measured by counting the name relations that assign a name to every node. For the extracted ontology the metric returns a count of 104783 nodes in total that have a name relation.

#Edges (#E): The second metric computes the number of edges in the graph. In the ontology the 104783 nodes are connected by 135338 edges. Only has-subcategory, has-entity, has-main-entity, has-attributeset and has-attribute relations are considered. The relations concerning name, depth and topic have values as their targets. Adding the number of edges concerned with values as well would bias further observations especially concerning **Tree Impurity**.

Edge Node Ratio (ENR): As suggested in [19] we measure the resulting ratio that is calculated by dividing the number of nodes by the number of edges. For the extracted ontology this results in the following ratio: $\frac{\#N}{\#E} \approx 0.774$.

Tree Impurity (TI) : A formula was suggested in [19] that may state the degree of how far the taxonomy in the ontology differs from being a tree. The resulting value tells how many edges would have to be deleted in order to transform the graph into a tree. For our ontology we get $\#E - \#N +$

[2] `http://en.wikipedia.org/`

1 = 30556. Thus the tree impurity metric underlines that the retrieved ontology is tangled.

The question arises how the number of nodes and edges can be broken down to the specific numbers for each type of resource and relation in the ontology. The diagrams in figure 3.4 answer this question and already give hints to what one has to look for in an analysis. They show how the number of nodes and edges are partitioned by displaying the number of categories (C), reachable categories from root (RC), unreachable categories from root (UC), entities (E), has-subcategory relations (SC), has-entity relations (HE), has-attributeset relations (HAS) and has-attribute relations(HA). We did not include has-main-entity relations since only 246 has-main-entity relations exist.

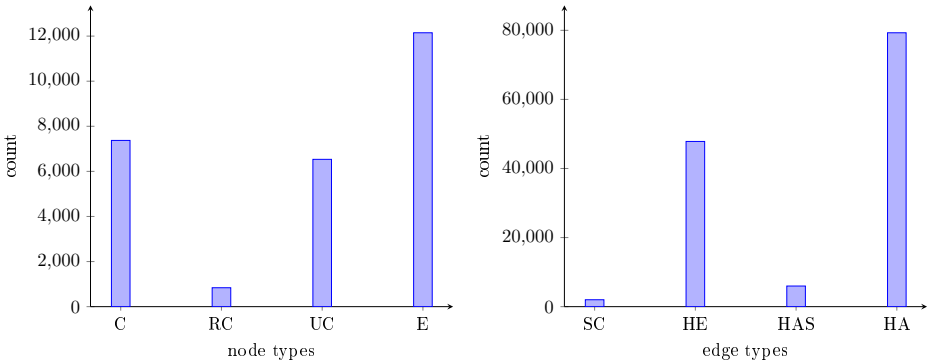

Figure 3.4: Type partitions

Figure 3.4 allows several observations. The left diagram shows how the number of categories can be broken down into reachable from the root category named 'Computer languages' and unreachable categories, that we also retrieve since we extract the supercategories of entities and categories. Out of 7373 extracted categories there are 839 categories that are reachable in the taxonomy starting at the root category. Therefore 12148 entities contained in the ontology are contained in 6534 categories that do not directly belong to the domain of computer languages.

These observations further lead to the question whether there is some kind of overcategorization apparent for many entities. **Overcategorization** was already identified as a bad smell by Rosenfeld et al. [36] labeled as **Concept too categorized**. The observations strengthen the rationale for such a bad smell. Therefore we also considered this bad smell and stated it in a specific way for the extracted ontology from Wikipedia in Section 3.3.

Since the number of reachable categories is low in relation to the total number of extracted entities, another issue is apparent. Further investigations show, that there are categories, which contain too many entities. This issue is captured as the bad smell **Bloated Category** in Section 3.3.

Another reason behind the numbers is considered as well. There might also be a certain amount of entities or even whole categories that do not belong to the computer languages domain itself. Those kind of categories and entities may relate more strongly to categories that do not belong to the target domain, which results in the high number of unrelated categories. In the next section we give an even stronger rationale for bad smells based on the stated suggestions.

Several observations can be made for the partition of the edge types as well. There is a relatively low number of has-attributeset relations. This number shows that 5971 attributesets are available in 12132 entities. The number of has-attribute relations namely 79291 results from the existence of many available attributes. Redundancies, inconsistencies and contradictions in available attributes can only be properly detected and resolved through logical reasoning, which is out of scope for this thesis.

There are still other observations that can be made. The number of has-entity relations exceeds the number of has-subcategory relations by far. The question arises what the average and the maximum of the number of entities per category and categories per entity is. This question can also be based on the observations made when one combines the two diagrams of figure 3.4. The number of has-entity relations is 47830 and the number of entities equals 12148. So each entity is contained in approximately four categories on average. In the following list several further metrics are stated based on the ideas of García et al. [19] and their proposed metrics named **Class out-degree** and **Class in-degree**. We use upper case letters to denote sets, e.g. C for the set of all categories and E for the set of all entities, and lower case letters to denote single elements in the upcoming formulas. All sets and elements are part of the extracted ontology.

Max Entities per Category (MaxEC): This metric tries to capture the maximal number of entities contained in one category. In order to compute this metric the set of entities in a category has to be computed through $hasEntity(c)$. The resulting formula is:

$$\begin{aligned} MaxEC = max_e \Leftrightarrow & \exists c \in C : |hasEntity(c)| = max_e \\ & \wedge \nexists c_2 \in C : |hasEntity(c_2)| > max_e \end{aligned}$$

A computation for the extracted ontology returns 474.

Average Entities per Category (AvgEC): We also want to provide ground for a bad smell that is concerned with categories having too many entities

that we state in the next section. The resulting formula for the average is:

$$AvgEC = \frac{\sum_{c \in C} |hasEntity(c)|}{|C|}$$

A computation for the extracted ontology returns approximately 6.49.

Max Categories per Entity (MaxCE): This metric calculates the maximal number of categories that contain one common entity. Since the ontology only contains an explicit relation from a category to an entity but not backwards, we use the inverse of this relation in our formula:

$$\begin{aligned} MaxCE = max_c \Leftrightarrow & \exists e \in E : |hasEntity^{-1}(e)| = max_c \\ & \wedge \nexists e_2 \in C : |hasEntity^{-1}(e_2)| > max_c \end{aligned}$$

A computation for the extracted ontology returns 56.

Average Categories per Entity (AvgCE): We also consider the average in order to provide further ground for the detection of a bad smell concerned with overcategorization. The formula for the average measure here is:

$$AvgCE = \frac{\sum_{e \in E} |hasEntity^{-1}(e)|}{|E|}$$

A computation for the extracted ontology returns approximately 3.92.

MaxEC shows, that there are categories containing a huge number of entities, but **AvgEC** is low in relation to it. This may suggest that there are specific categories, that have too many entities.
MaxCE demonstrates that there are entities, where the annotation of categories may be used to tag the entity with every somehow related category. All of the computed numbers are used as ground for the upcoming bad smells considering overcategorization of entities and categories containing too many entities.

After covering all proposed metrics that focus on the whole ontology graph, metrics follow, that are only concerned with one category at a time. Since depth measures may provide further information on the existence of chains of inheritance and further tangledness, we also propose metrics for this. An aspect of interest is the maximal path length from one node to another. Instead of calculating the maximal path directly we chose another way. Figure 3.5 shows a tree and a graph with abstract elements in them.

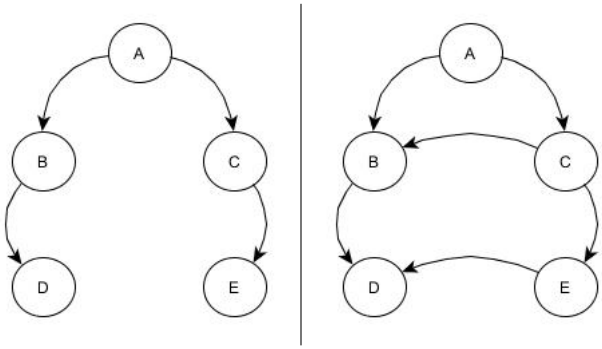

Figure 3.5: An abstract sample tree and an abstract sample graph

A well designed taxonomy may have a hierarchy, that has the same structure as a tree. In a well formed tree there is one path from one node to another at most. Therefore the distance can be calculated by counting the difference in the nodes' levels in the hierarchy.

Wikipedia's taxonomy does not conform to a tree. Instead its taxonomy graph may be structured as the abstract sample graph on the right of figure 3.5. Therefore the distance can no longer be calculated through computing the level difference. In the extracted ontology we assign each category its shortest distance to the root category. When comparing two categories, where there is a path from one category to the other, the difference in depth is of interest. We are further interested in the number of paths in the taxonomy and their lengths.

Therefore we propose a metric that combines measures on how many paths exist from one node to the other and each path's length. In the abstract graph on the right side of figure 3.5 the shortest path from `A` to `D` through `B` has the length two. More alternative paths exist. A sequence of `A,C,B,D` is of length three. The last possible sequence is `A,C,E,D` that also has length three. There are three paths in total from `A` to `D`.

A mediator is a node that appears on a way from one node to the other. The number of mediators including the start node and excluding the target node on all available paths is denoted as **Depth Mediators (DM)**. For example the first path from `A` to `D` has two mediators `A` and `B`. The second and the third path both have three mediators. This results in **DM** from `A` to `D` of eight.

When this kind of measure is used, one always has to observe the shortest distance to the root. This way **DM** can be used as a measure for tangledness on the way between two nodes.

The following list provides several metrics concerned with paths reachable from a category and corresponding depth values. These metrics were inspired by the depth concerned metrics from Gangemi et al. [34] and **Depth of inheritance** from García et al. [19].

Number of total reachable Categories (NTC): The metric calculates the amount of transitive and direct reachable subcategories for a given cat-

egory. A corresponding formula uses the function $hasSubCategory^{+}(c)$, that returns the set of categories reachable through a path of at least one has-subcategory relation:

$$NTC(c) = |hasSubCategory^{+}(c)|$$

Maximal Reached Depth (MaxRD): Here we are interested in how deep we can get from the target category in the taxonomy. Let $depth(c)$ return the depth value that was assigned in the extraction process. Then the metric formula corresponds to:

$$\begin{aligned} MaxRD(c) = d_{max} \Leftrightarrow & hasSubCategory^{+}(c) = C_r \wedge C_r \subset C \wedge c \in C \\ & \wedge \exists c_{max} \in C_r : depth(c_{max}) = d_{max} \\ & \wedge \nexists c_2 \in C_r : depth(c_2) > d_{max} \end{aligned}$$

Maximal Depth Mediators (MaxDM): The metric calculates the maximal number of depth mediators from the target category to a reachable subcategory. Let $dm(c, c_2)$ be the function that returns the set of mediators on the way from a category c to a direct or transitive subcategory c_2. The formula can be stated as follows:

$$\begin{aligned} MaxDM(c) = md_{max} \Leftrightarrow & hasSubCategory^{+}(c) = C_r \wedge C_r \subset C \wedge c \in C \\ & \wedge \exists c_{max} \in C_r : |dm(c, c_{max})| = md_{max} \\ & \wedge \nexists c_2 \in C_r : |dm(c, c_2)| => md_{max} \end{aligned}$$

Average Reached Depth (AvgRD): For each category, we are also interested what the average reached depth is. Therefore the above formula for the maximal depth has to be slightly adapted as follows:

$$\begin{aligned} & AvgRD(c) = \frac{\sum_{c_i \in C_r} depth(c_i)}{|C_r|} \\ & \Leftrightarrow hasSubCategory^{+}(c) = C_r \wedge C_r \subset C \wedge c \in C \end{aligned}$$

Average Depth Mediators (AvgDM): This metric calculates the average number of depth mediators from the target category to a reachable subcategory. The corresponding formula is:

$$\begin{aligned} & AvgDM(c) = \frac{\sum_{c_i \in C_r} |dm(c, c_i)|}{|C_r|} \\ & \Leftrightarrow hasSubCategory^{+}(c) = C_r \wedge C_r \subset C \wedge c \in C \end{aligned}$$

After stating our chosen metrics we also give some examples of metric calculations for chosen categories in table 3.1. In this table we focus on the maximal values for depth mediators and reached depth.

Name	depth	NTC	MaxRD	MaxDM
XML-based standards	3	31	8	5
Articles with example code	1	32	3	2
Markup languages	1	103	8	7
Programming language topics	1	773	9	26

Table 3.1: Depth metrics value examples

Table 3.1 allows several observations. From 'Programming language topics' one may reach 773 categories through has-subcategory relations out of 839 processed categories in the complete ontology. This can be caused by an abstract represented topic for this category, such that a wide range of assignable subcategories and entities can be covered. From 'Programming language topics' one may reach the category 'Data types' through has-subcategory relations, which is a direct subcategory of 'Computer languages' as well. A high maximal number of depth mediators can also be caused by redundant has-subcategory relations and multiple paths. Compared to the category 'Markup languages' the taxonomy differs more from being a tree.

The metric values of 'Markup languages' hint at a well chosen taxonomy concerning its reachable subcategories. All categories in the table except for 'Programming language topics' show metric results that points to a tree like structure, since the difference between the maximum of mediators and the maximum of reached depth is small.

For now only depth was considered. Next we want to describe several metrics that are concerned with breadth based on ideas from Gangemi et al. [34] and García et al.[19], who coined related measures with the term 'Number of children'.

Number of direct Entities (NDE): This metric counts the number of entities that are reachable through one has-entity relation from a target category. Let $hasEntity(c)$ be the formula that returns the set of directly reachable entities through an entity relation:

$$NDE(c) = |hasEntity(c)|$$

Number of direct Subcategories (NDS): Other than the metrics concerned with depth values, this metric computes the number of direct subcategories for a given category. The corresponding formula can be stated as follows:

$$NDS(c) = |hasSubCategory(c)|$$

Number of total reachable Entities (NTE): This metric computes the number of entities contained in the subgraph with the given category as its root category. A formula for the metric relies on the following function:

$$haveEntities(C_s) = \cup_{c \in C_s} hasEntity(c)$$

Name	depth	NDE	NDS	NTE	NTC
XML-based standards	3	261	10	934	31
Markup languages	1	216	22	2019	103
Articles with example code	1	55	30	600	32
Automata (computation)	3	1	1	240	7
C++ libraries	4	116	7	761	53

Table 3.2: Exemplary values for breadth concerned metrics

The function takes a set of categories as a parameter and returns the set of all entities that are part of the set's categories. Thus the resulting formula for the metric itself can be stated by using the function $hasSubCategory^*(c)$, which returns the set of categories containing the given category and all reachable subcategories:

$$NTE(c) = |haveEntities(C_s)| \Leftrightarrow hasSubCategory^*(c) = C_s$$

After stating these metrics, the categories that showed results pointing to a tree like structure concerning their depth values are analyzed concerning these breadth metrics as well. We further display the breadth values for several other categories that we deem interesting for our concern. Table 3.2 shows five exemplary analyzed categories.

Table 3.2 allows several observations. While the first three stated categories showed results concerning their depth values that might point to a tree structure, the breadth values show that a lot of entities can be reached from them. In Section 3.3 we identify and discuss those three categories as matches for the bad smell **Bloated Category**.

'Automata (computation)' is a category that has only one subcategory and only one entity. Further the contained entity is reachable from its subcategory as well. Therefore one may argue that this category has to be analyzed concerning its usefulness. We will identify this category as a match to a bad smell later.

'C++ libraries' covers three depth levels namely depth five, six and seven, but 761 entities are reachable from this category. The category also has 53 reachable subcategories to provide a structure for the amount of entities. Since many libraries exist, such as 'Qt' and others, the amount is reasonable and can be interpreted as not suspicious. This category is an example that metrics provide room for interpretation. A computed value does not tell that some structure is corrupted.

Due to the breadth values, for example for 'Automata (computation)', the bad smell from Fahad and Qadir [41] **Weaker domain specified by subclass error** and the possibility for an unnecessary category we formulated the metric **Subdomain Entity Ratio (SER)**. It is concerned with the number of entities that can be reached from a category and a chosen subcategory.

Category	Subcategory	Diff
Automata (computation)	Automata theory	0
OCaml programming language family	OCaml software	0
Haskell implementations	Free Haskell implementations	0
Object-based programming languages	Object-oriented programming languages	4

Table 3.3: Exemplary values for Subdomain Entity Ratio

Subdomain Entity Ratio (SER):

$$SER(c_1, c_2) = NTE(c_1) - NTE(c_2) \Leftrightarrow c_2 \in hasSubCategory(c_1)$$

Fahad and Qadir [41] tried to find classes where a subclass' domain is greater than its superclass' domain. In Wikipedia's category graph the above stated metric provides means to detect related problems. Table 3.3 has several examples for categories that seem to match this bad smell based on the metric values.

The category 'Automata (computation)' may be obliterated since all transitively reachable entities are reachable from its subcategory 'Automata theory' as well. Its name does not diffuse the topic further. Thus 'Automata theory' may be added to all supercategories of 'Automata (computation)'.

The next example is about the category 'OCaml programming language family'. One may expect a couple of interpreted languages that are related to 'OCaml', but instead software concerned entities are contained in its subcategory 'OCaml software'. 'OCaml software' is its only subcategory and 'OCaml programming language family' has no direct entities. Thus this has-subcategory relation may be obliterated.

A third sample is posed by 'Haskell implementations', which has no contained entities and its only subcategory is 'Free Haskell implementations'. We will later identify this category as a match for the bad smell **Speculative Generality**.

The last example is concerned with the category 'Object-based programming languages' and its subcategory 'Object-oriented programming languages'. This category's page states the following suggestion: 'Note that these programming languages are further classified as being either: Prototype-based or Class-based.' The terms Prototype-based and Class-based have a reference to the corresponding articles 'Prototype-based programming' and 'Class-based programming'. Both articles can only be found in the subcategory 'Object-oriented programming language'. This already hints at the fact that the intended partition was not proceeded correctly, since out of 2785 entities in 'Object-based programming languages' 2781 are also reachable through 'Object-oriented programming languages' including the defining articles for the intended partition. In Section 3.3 we will cover other kinds of partition errors as well.

3.3 Bad Smells

In Section 3.1 we described the existing variety of approaches to finding a lack of quality in an ontology. Now we want to present our own proposals for bad smells that are suited to analyze the extracted ontology.

In the extracted ontology we are not only interested in bad smells concerning its structure. We propose samples for bad smells that consider its semantics as well. For each proposed bad smell we give its name and the source of the existing smells that served as basis. Afterwards we discuss the proposed bad smell based on existing matches. At last we describe a mechanism on how to detect the bad smell with exemplary SPARQL queries working on our triplestore.

3.3.1 Bloated Category

This first bad smell was inspired by the existing object oriented design bad smells **Large class** from Fowler et al. [14] and **Large Category** from Rosenfeld et al. [36]. It denotes the situation, where a container, such as a class or a category, has too many members. In our case we use it to tell, that there may be too many entities in one category.

Wikipedia published several guidelines to editors concerning the use of categories [3]. Those guidelines contain several clear requirements for categories. For example there has to be a sufficient number of subcategories. These should provide means for a well designed taxonomy such that one category can be diffused further into its subtopics.

The bad smell **Bloated Category** is concerned with weakly structured categories, where many entities are contained and no further diffusing subcategories are provided. We try to detect this bad smell with the help of the **NDE** metric. If a category has more than thirty entities, we consider it suspicious. The detection mechanism can be formulated in a formula as follows:

$$BloatedCategory = \{c \in C \mid |hasEntity(c)| > 30\}$$

Since the extract ontology is saved as a Jena TDB triplestore we chose to implement a detection mechanisms in SPARQL. One has to take notice here that such a detection mechanism has to be clearly separated from the actual bad smell. While the bad smell relies on the detection through human intuition including an observation of the whole situation the detection merely give proposals for possible matches to the bad smell. Listing 3.1 shows the SPARQL query for **Bloated Category**.

[3] `http://en.wikipedia.org/wiki/Wikipedia:Category`

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?categoryname ?howMany
WHERE {
        ?category clonto:name ?categoryname .
    {SELECT ?category (COUNT(?entity) AS ?howMany)
     WHERE  {
                ?category clonto:hasEntity ?entity .
        }
    group by ?category}
    FILTER(?howMany > 30)
}
ORDER BY ?howMany
```

Listing 3.1: SPARQL query detecting Bloated Category

After giving a rationale and a proposal for a corresponding detection mechanism, exemplary matches are discussed next. For every bad smell we picked matches. These matches do not always imply the necessity of improvements. A decision on whether an improvement is needed depends on the ontology engineer's choice.

The first match we want to discuss is the category 'User Basic'. Even though it is not marked as a hidden or as a maintenance category, this category can be reached through has-subcategory relations from 'Computer languages', since it is a direct subcategory of 'Basic programming language'. It was detected by the detection mechanism for **Bloated Category**, because it has 1113 contained entities. After its detection we put it on the list of excluded categories for the extraction process, because only user pages and templates are contained in it, that do not provide valuable information to our computer languages ontology. This category may basically be seen as an example of missing annotations leading to its inclusion.

The second match we want to discuss is the category 'Free software programmed in C'. With 474 contained entities, it is the biggest category in the extracted ontology. This category is an immediate subcategory of 'C (programming language)'. A match like this is difficult to improve. At first one has to decide whether software should be integrated in an ontology concerned with computer languages. The decision depends on the chosen degree of detail. Contained articles pose examples of software that is programmed in C. Therefore a relation to the programming language C clearly exists, but if one was to chose a more constrained view on computer languages then categories for software could be filtered out. Secondly it is questionable whether the category's structure could be improved through subcategories since the contained articles are diverse and detailed further diffusion is difficult.

'Scheme (programming language) implementations' is the third exemplary match. It contains 31 entities and three subcategories. Therefore it conforms to an edge case for our chosen number. Its title implies that it collects compilers

and interpreters, which actually implement the programming language Scheme. Instead it actually contains other entities as well. For example the contained entity 'Racket (programming language)' is a programming language, which belongs to the family of Scheme. Another example is 'Scsh', which corresponds to a POSIX API layered on top of Scheme to optimize Scheme's capabilities. These entities can be removed from the category to improve consistency.

The above discussion leads to several possibilities on improvements. If the reason for a category being bloated is, that it is actually a maintenance category, it can be obliterated completely resulting in the removal of the category from the ontology and possibly members that are not reachable after its deletion. Secondly if a category has many entities and no suitable subcategories, one may introduce new subcategories based on commonalities in a subset of contained entities. Third and last if a category already has subcategories that are not used consistently, one may move chosen entities into subcategories.

3.3.2 Overcategorization

In contrast to a category having too many entities, we also want to look at entities that are contained in too many categories. The idea is based on the bad smell **Concept too categorized**, which was formulated by Rosenfeld et al. [36] for a semantic wiki.

Overcategorization denotes the situation that an entity is contained in too many categories. A probable reason for it is that editors used the category assignments more in a way to show what categories are related to a corresponding article even though Wikipedia's guidelines for categories state that categories should not be annotated this way.

The metric analysis discussed the number of categories per entity. A computation for the extracted ontology results in an average of 3.92 and a maximum of 56. Based on further manual observations we choose to suspect the entities to be too categorized that are contained in more than eight categories. The resulting formula can be stated as follows:

$$Overcategorization = \{e \in E \mid |hasEntity^{-1}(e)| > 8\}$$

The corresponding SPARQL query is placed in listing 3.2, which also uses the inverse of the has-entity relation.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?entityname (COUNT(?cat) as ?howManyc)
WHERE {
        ?entity ^clonto:hasEntity ?cat .
        ?entity clonto:name ?entityname .
}
GROUP BY ?entityname
HAVING (?howManyc > 8)
ORDER BY ?howManyc
```

Listing 3.2: SPARQL query detecting Overcategorization

The entity with the highest number of containing categories is 'John von Neumann' having 56 containing categories. He appears in many categories that do not belong to the domain of computer languages. The corresponding entity can be reached through the category 'Cellular automatists'. Here the question can be posed whether to integrate persons in an ontology for Computer languages or not. In our observations we noticed that especially entities representing persons have a high number of categories. As an edge case the entity 'Andreas Raab' representing a german computer scientist can be named. The entity has nine containing categories. If persons should not be included, the entities 'John von Neumann' and 'Andreas Raab', and the category 'Cellular automatists' should be removed from the ontology.

Since 'MATLAB' is a computing environment and a fourth-generation programming language the entity is assigned to many categories either concerned with programming languages, mathematics or specific types of software. One may already find a lot of redundancies here. For example the entity can be found in the categories 'Data analysis software' and 'Data mining and machine learning software', while 'Data mining and machine learning software' is a subcategory of 'Data analysis software'. Before further reductions of annotated categories can be performed, a redundancy analysis has to be made, which we propose in the next subsection.

With eleven assigned categories, 'XML' also becomes suspicious. By taking a look at its containing categories one finds the categories 'Application layer protocols' and 'Presentation layer protocols'. Such protocols exist and use XML, but that does not imply that the entity should be directly a part of these categories. This gives enough reason to remove the has-entity relations from the ontology.

Various improvements are possible. The entity 'John von Neumann' and its category 'Cellular automatists' are matches, where an ontology engineer has ground to argue that both are removed completely from the ontology. 'XML' and 'MATLAB' showed an example, where categories assignments can be judged as misused. Therefore a corresponding improvement involves the deletion of unfit has-entity relations.

3.3.3 Redundant Has-Subcategory Relation

This smell was inspired by Baumeister et al.'s [38] suggestion to search for redundancies between rules. We transfer this idea and try to find redundancies in the category graph. The relation has-subcategory is transitive. Therefore if a category is a subcategory of two categories, who are also connected through a has-Subcategory relation, the relation to the upper category might be redundant.

Since Wikipedia's categories can have a special type one has to decide whether a match really relates to bad quality. As we stated earlier in Section 2 there are two special types of categories. A non-diffusing subcategory only offers the information of an additional property through the category name. Therefore if a category is subcategory of a supercategory and a supercategory's non-diffusing subcategory it may not be a sign of bad quality. An eponymous category serves as a container for every strong related topic. Both special types have to be considered when this bad smell is detected.

The general case that is also mentioned in the guidelines for editors [4] is as follows. If a page belongs to a subcategory of a category then it is not placed directly into the category. An editor is not supposed to add categories to pages, such as articles and categories, as if categories were tags.

A corresponding detection mechanism analyzes two subcategories c_a and c_b in a category c_c. If c_a is also a subcategory of c_b a match for this bad smell is identified. The resulting formula is:

$$\begin{aligned} RedundantSubCategoryR = \{(c_a, c_b, c_c) \mid\ & c_a, c_b, c_c \in C \wedge \\ & c_a, c_b \in hasSubCategory(c_c) \\ & \wedge\, c_a \in hasSubCategory(c_b)\} \end{aligned}$$

Listing 3.3 shows the corresponding SPARQL query.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?subcategoryname ?categoryname ?supercategoryname
WHERE {
    ?supercategory clonto:hasSubCategory ?subcategory .
    ?supercategory clonto:hasSubCategory ?category .
    ?category clonto:hasSubCategory ?subcategory .
        ?subcategory clonto:name ?subcategoryname .
        ?category clonto:name ?categoryname .
        ?supercategory clonto:name ?supercategoryname .
}
```

Listing 3.3: SPARQL query detecting Redundant Has-Subcategory Relation

[4] http://en.wikipedia.org/wiki/Wikipedia:Categorization

The category 'TeX' is a subcategory of 'Markup languages' and its subcategory 'Mathematical markup languages'. 'TeX' is not only used in mathematics. Instead mathematics is just a specific field in which it is usable. Therefore one may argue that 'Mathematical markup languages' is a non-diffusing subcategory since it does not proceed the logical hierarchy in 'Markup languages', but it expresses that markup languages in this category are used for mathematics. This leads to the conclusion that it is an example of a negative match and no improvements have to be done.

A positive match can be found by looking at the category 'Recursion' that is a subcategory of 'Control flow' and its subcategory 'Iteration in programming'. One may argue that the has-subcategory relation from 'Recursion' to 'Control flow' is redundant since recursion is also used to repeat some kind of process.

One may take notice that an improvement here is semantics preserving. This results from the transitivity of the has-subcategory relation. No information is lost due to a removal of a redundant has-subcategory relation.

3.3.4 Redundant Has-Subcategory Cluster

This subsection presents an alternative approach to detecting redundancies. Instead of searching for only one redundant has-subcategory relation we want to look for a category that has at least two subcategories and there is a subcategory from which all other subcategories are reachable as well. This mechanism allows the detection of whole clusters of redundancies. The resulting formula can be stated as follows:

$$\begin{aligned} RedundantSubcategoryC = \{c \in C \mid \exists s_1, s_2 \in hasSubCategory(c) : s_1 \neq s_2 \\ \wedge \forall s_n \in hasSubCategory(c) : s_n \in hasSubCategory^*(s_1)\} \end{aligned}$$

A corresponding SPARQL query is stated in listing 3.4.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?lazycatname ?categoryname
WHERE {
        ?scat clonto:hasSubCategory ?lazycat .
        ?lazycat clonto:hasSubCategory ?category .
        ?lazycat clonto:hasSubCategory ?category2 .
        FILTER NOT EXISTS{
                ?lazycat clonto:hasSubCategory ?cat .
                FILTER NOT EXISTS{
                        ?category clonto:hasSubCategory* ?cat .
                }
        }
        FILTER(?category != ?category2)
        ?scategory clonto:name "Computer languages" .
        ?scategory clonto:hasSubCategory* ?lazycat .
        ?lazycat clonto:name ?lazycatname .
        ?category clonto:name ?categoryname .
}
```

Listing 3.4: SPARQL query detecting Redundant Has-Subcategory Cluster

Several matches can be found. 'C POSIX library' has two subcategories namely 'C standard library' and 'POSIX error codes'. The subcategory 'POSIX error codes' can also be found in 'C standard library'. This match also matches **Redundant Has-Subcategory Relation** and can may therefore be treated accordingly.

In analogy to the first match 'Object-based programming languages' and 'Object-oriented programming languages' fit the described situation as well. Since they also exactly match a redundant has-subcategory relation and an according fix solves this bad smell problem as well the match is of no further interest in this subsection.

A different match is posed by 'Data serialization formats' and its subcategory 'XML'. 'Data serialization formats' only has one other subcategory namely 'JSON'. The category 'JSON' is not directly reachable from 'XML', but through a transitive way that goes through the subcategory of 'XML' namely 'Ajax (programming)'.

3.3.5 Redundant Entity Containment

After considering redundant has-subcategory relations **Redundant Entity Containment** tries to capture redundant has-entity relations. It is based on the same ideas as the bad smell in the previous subsection. Both bad smells may be merged into one single bad smell, but in this thesis we decided to provide a firm separation of concerns, since the exemplary matches present diverse issues causing a match.

The metric analysis in Section 3.2 showed that there are many categories with a lot of contained entities. Exemplary results also show that entities that

can be found in one category, may also be present in a category's subcategory. Such a situation points to a redundant entity containment since the entity is already related to a category when it is present in one of its subcategories.

This bad smell tries to identify an entity that is contained in a category and is part of the category's subcategory. The resulting detection formula can be stated as follows:

$$DoubleEntityContainment = \{(e_a, c_b, c_c) \mid e_a \in E \wedge c_b, c_c \in C \\ \wedge c_a, c_b \in hasSubCategory(c_c) \wedge c_a \in hasSubCategory(c_b)\}$$

The listing 3.5 contains the corresponding SPARQL query that returns the list of possible matches.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?entityname ?subcategoryname ?categoryname
WHERE {
    ?category clonto:hasSubCategory ?subcategory .
    ?category clonto:hasEntity ?entity .
    ?subcategory clonto:hasEntity ?entity .
        ?entity clonto:name ?entityname .
        ?category clonto:name ?categoryname .
        ?subcategory clonto:name ?subcategoryname .
}
```

Listing 3.5: SPARQL query detecting Redundant Entity Containment

One exemplary match is the entity representing 'Unified Modeling language', which is contained in 'Unified Modeling Language' and its supercategory 'Software modeling language'. Its main article is 'Unified Modeling language'. The category 'Unified Modeling Language' can be denoted as an eponymous category. On the one hand an ontology engineer can decide to leave an eponymous category and the article both in the supercategory following Wikipedia's rules. On the other hand an ontology engineer can decide to obliterate the eponymous category or to just remove this redundancy if the category is supposed to remain as a container for all modeling languages that are part of the UML.

'Abstract Syntax' is part of the category 'Compiler construction' and its supercategory 'Programming language theory'. This time one may argue that this match is not a result of a redundant relation, but a taxonomic error instead. Abstract syntax in general is needed in more areas than just compiler construction. It also plays a role at type theory and interpreter implementation. Therefore it should not be present in the category of compiler construction since it is not a concept that can explicitly be found in compiler construction. Instead it is a concept that belongs to the general field of programming language theory. This can be grounded on the observation, that abstract syntax plays a role in the

topics covered by the other subcategories of 'Programming language theory' [5]. Therefore a removal from the subcategory namely 'Compiler construction' can be feasible.

Another match is the entity 'Helium (Haskell)' that is part of 'Free Haskell implementations' and its supercategory 'Free compilers and interpreters'. This is a positive match. Since the category 'Free Haskell implementations' is concerned with compilers and interpreters already the redundant relation between the entity and 'Free compilers and interpreters' can be removed.

In retrospect we see that diverse improvements can be applied to a match for this bad smell. One has to decide whether there is one relation that is flawed because of redundancy. The decision has to be made considering a decision on how to handle eponymous categories.

3.3.6 Speculative Generality

The idea of inspecting **Speculative Generality** was primarily inspired by Fowler et al. [14]. Additionally a discussion and basis for this bad smell was strengthened through the idea by Fahad and Qadir [41] who consider **Sufficient Knowledge Omission Error** in an ontology.

This bad smell describes the situation that a category does not provide enough functionality. A category is suspicious if it only has subcategories, but no entities are assigned to it. **Speculative Generality** in a category may be caused by an editor's expectation, that there will be further assignable subcategories and articles in the future. A category with no entities in it may be just an abstract category, where one would prefer to insert entities into its subcategories or it may be redundant, especially if it only has a small number of subcategories. It lacks contained knowledge to provide enough functionality to exist in an ontology.

The detection mechanism merges both aspects. It returns categories that only have less than eight subcategories and a direct contained entity does not exist. The resulting formula can be stated as follows:

$$SpeculativeGenerality = \{c \in C \mid hasEntity(c) = \emptyset \wedge |hasSubCategory(c)| < 8\}$$

A corresponding SPARQL query can be stated with a subquery as in listing 3.6.

[5] `http://en.wikipedia.org/wiki/Category:Programming_language_theory`

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT DISTINCT ?categoryname ?howmanySub
WHERE {
    ?categoryS clonto:hasSubCategory ?category .
    FILTER NOT EXISTS{?category clonto:hasEntity ?entity}
    {SELECT ?category (COUNT(?subcat) as ?howmanySub)
     WHERE { ?category clonto:hasSubCategory ?subcat . }
     GROUP BY ?category
     HAVING (?howmanySub < 8) }
    ?category clonto:name ?categoryname .
}
ORDER BY ?howmanySub
```

Listing 3.6: SPARQL query detecting Speculative Generality

Categories that already show significant low values in a breadth metric analysis reappear as matches to the detection mechanism. 'OCaml programming language family' has no entities and only one subcategory 'OCaml software'. One might expect variants of the OCaml programming language or certain implementations, but instead software programmed in OCaml is presented as a subcategory. As an improvement one may consider to either remove the matching category and leave 'OCaml software' reachable only through 'Software by programming language'. The other option would be to rename the category to 'OCaml (programming language)' and make the article covering OCaml its main entity. This would transform the category into a eponymous category and a wider range of fitting assignable topics would be provided.

A similar match is presented by the category 'Haskell implementations' that has only one subcategory namely 'Free Haskell implementations'. Since no commercial Haskell implementations seem to be identifiable in Wikipedia this category may be obliterated by collapsing the hierarchy. The subcategory 'Free Haskell implementations' can be added to all super categories of 'Haskell implementations'.

'Uncategorized programming language' does not have any members. This category is a Wikipedia maintenance category and can therefore be removed from the ontology.

The category 'Programming languages by creation date' has only two subcategories. These subcategories further diffuse the time of the creation by centuries namely by 'Programming languages created in the 20th century', which has six subcategories and nine contained entities, and 'Programming languages created in the 21st century', which has two subcategories. Each of the subcategories further diffuse the date ranges. Therefore this category provide enough knowledge to be kept alive.

In retrospect this bad smell may lead to the following improvements. A category that at least has a subcategory may be renamed in order to increase

its range of assignable entities. If no such possibility is given the hierarchy at that point may be collapsed be adding all subcategories of the matching category to its supercategories.

3.3.7 Lazy Category metric based

Speculative Generality is concerned with suspecting categories that may be introduced to provide an abstraction level, but are not of any further use. Most times categories matching speculative generality have only one subcategory that also contains all the entities from the matching category. In this subsection and the next we are concerned with the bad smell **Lazy Category**. It was inspired by the ideas of **Lazy Class** from Fowler et al. [14] combined with redundancy analysis from Baumeister et al. [38], **Sufficient Knowledge Omission Error** from Fahad and Qadir [41] and our own metric analysis on breadth.

A lazy category may not provide enough knowledge to exist in a comprised ontology. Several aspects can be analyzed that may lead to the conclusion that a category fits the bad smell **Lazy Category**. In order to provide a separation of concern we subdivided these aspects into separate subsections.

In this subsection we base the detection mechanism on a metric evaluation. A category that has less than seven members is considered suspicious and should be analyzed. The choice of number is based on the state average entities per category metric. In the extracted ontology an average of about 6.5 is computed. A resulting formula can be stated as follows:

$$LazyCategory = \{c \in C \mid (|hasEntity(c)| + |hasSubCategory(c)|) < 7\}$$

Listing 3.7 presents the corresponding SPARQL query. The query contains two subqueries that compute the entities of a category and its subcategories individually so that no entity or subcategory appears twice for the same category. These results are combined with a union to retrieve results where either the number of entities or the number of subcategories equals zero.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?categoryname (COUNT(?entity) as ?NumberOfEntities) (COUNT(?
    subcategory) as ?NumberOfSubcategories)
    (?NumberOfEntities+?NumberOfSubcategories as ?howmany)
WHERE {
        ?category clonto:depth ?depth .
        ?category clonto:name ?categoryname .
        {SELECT ?category ?entity
         WHERE{ ?category clonto:hasEntity ?entity . } }
        UNION
        {SELECT ?category ?subcategory
         WHERE{ ?category clonto:hasSubCategory ?subcategory . } }
}
GROUP BY ?categoryname
HAVING (?howmany < 7)
ORDER BY ?howmany
```

Listing 3.7: SPARQL query detecting Lazy Category

Only one entity exists in 'Process termination functions' namely 'Exit (system call)', which is also assigned to the category 'Process (computing)'. 'Process (computing)' is the supercategory of the matching category. Therefore 'Process termination functions' can be removed from the ontology since it does not provide much knowledge.

A discussable match is 'Axiomatic semantics'. It contains only one subcategory 'Program logics' that is a match as well. The subcategory only contains articles that are directly needed in the topic of axiomatic semantics namely 'Hoare logic', 'Predicate transformer semantics' and 'Separation logic'. An ontology engineer may decide to remove 'Program logics' and add all contained entities to the supercategories of 'Program logics'. The decision depends on how valuable this category is deemed by the ontology engineer. A logical hierarchy is definetely proceeded, but 'Axiomatic semantics' does not contain any entities. Therefore the other quality improvement option would be to remove 'Axiomatic semantics', but then the relation of 'Axiomatic semantics' to its entities and the knowledge implied by its name would be lost.

A negative match is 'Lua software'. It has two subcategories namely 'Lua-scriptable software', which contains 43 entities and one subcategory, and 'Lua-scripted software', which has six entities and two subcategories. The matching category also has three entities itself. Therefore it provides enough knowledge and a proceeding logical hierarchy to provide reasons for its further existence.

Another discussable match is 'Programming languages created in 1959'. It only has four entities. This kind of category may be kept in order to provide a taxonomy which also relates to the date of creation. On the other hand an ontology engineer may decide that this relation should be expressed only through an attribute and remove this kind of category.

3.3.8 Lazy Category - Entity Containment

In this subsection we try to find categories whose directly contained entities are also reachable through any of its subcategories. Therefore there exist many redundancies whose deletion may lead to the category having zero directly contained entities. This is also similar to **Speculative Generality** only with the difference that we combine its idea with a search for redundancies.

The detection mechanism therefore searches for a category where there exists a path from a subcategory to each directly contained entity. There is at least one entity in the category, which reduces the possibility of false positives based on observations. A formula can be stated as follows:

$$LazyCategoryEC = \{c \in C \mid \forall e \in hasEntity(c) \exists s \in hasSubCategory(c) : e \in hasEntity^{+}(s) \wedge |hasEntity(c)| > 0\}$$

Speculative Generality and **Lazy Category** are related to some degree. Listing 3.8 shows the corresponding SPARQL query. It uses a doubled negation on existence since a for-all quantifier does not exist in Jena's ARQ SPARQL syntax [6].

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT DISTINCT ?lazycatname
WHERE {
    ?lazycat clonto:name ?lazycatname .
    ?categoryS clonto:hasSubCategory ?lazycat .
    FILTER EXISTS{?lazycat clonto:hasEntity ?entity .}
    FILTER NOT EXISTS{
        ?lazycat clonto:hasEntity ?entity2 .
        FILTER NOT EXISTS{
            ?lazycat clonto:hasSubCategory+ ?cat .
            ?cat clonto:hasEntity ?entity2 . } }
}
```

Listing 3.8: SPARQL query detecting Lazy Category - Entity Containment

An analysis of the extracted ontology returns five matches. The first one is the category 'Automata (computation)'. As described earlier this category can be obliterated since it cannot be distinguished clearly enough from its subcategory 'Automata theory'.

The second match is 'Cairo (graphics)', which has only one subcategory 'Software that uses Cairo' with fourteen contained entities. Its main article is also the main article of the subcategory. Since Cairo is a library and not a programming language one may argue that this category can be obliterated even though it is an eponymous category. The connection to the article about Cairo

[6] https://jena.apache.org/documentation/query/

would still remain since it is the main article of the subcategory. Therefore there is no considerable information loss through a removal.

'Clutter (software)' also has only one subcategory 'Software that uses Clutter'. Therefore the same situation is apparent as in 'Cairo (graphics)', which should lead to an analogous treatment.

The last two matches namely 'Persistent programming languages' and 'Data-centric programming languages' present the occurrence of a cycle since the both categories are the subcategory of the other. Therefore we will discuss these matches in Subsection 3.3.9.

In retrospect two improvement possibilities were named to get rid of the **Lazy Category** match. The category's members have to be added to the supercategories of the matching category. Or the category is removed from the ontology without pulling up its members. Thereby members that are not reachable through any other category are removed as well.

3.3.9 Cycle

The next bad smell is 'Cycle'. Here we are concerned with cycles only relating to has-subcategory relations. It is based on the cycle related anomalies proposed by Baumeister et al. [38].

In an extracted ontology from Wikipedia one may try to find cycles in has-subcategory relations by searching for a category that may reach itself through transitivity. Thus the resulting formula is:

$$Cycle = \{c \in C \mid c \in hasSubCategory^{+}(c)\}$$

A corresponding SPARQL query is stated in listing 3.9, which is implemented in analogy to the formula. When one executes this query it returns all categories that appear on the way from a category to itself.

```
1 PREFIX clonto: <http://myCLOnto.de/>
2 SELECT ?categoryname
3 WHERE {
4         ?category clonto:name ?categoryname .
5     ?category clonto:hasSubCategory+ ?category .
6 }
```

Listing 3.9: SPARQL query detecting Cycle

The only positive matches are the categories 'Persistent programming languages' and 'Data-centric programming languages' that have a has-subcategory

relation in both ways. A chosen improvement has to be ground on a few reasons since the taxonomy is affected. At this point we propose the integration of 'Persistent programming languages' into 'Data-centric programming languages' by removing the has-subcategory relation from 'Data-centric programming languages' to 'Persistent programming languages'.

All entities from 'Persistent programming languages' can be moved to 'Data-centric programming languages'. 'Persistent programming languages' can be removed completely from the ontology afterwards. In order to provide further reasons for this kind of improvement we take a look at each contained entity in 'Persistent programming languages' at Wikipedia. The first one is 'ABC (programming language)'. It is defined as an imperative general-purpose programming language and programming environment in its corresponding article. Its only hint at being a programming language that encourages persistence is by mentioning that it supports 'persistent variables'. This term contains a link to the article 'Static variable'. This can be interpreted as a flawed usage of the term. 'Caché ObjectScript' is the second contained entity that supports embedded SQL. Thus it is feasible to add it to 'Data-centric programming languages' as well. The third entity is 'JADE (programming language)', which is also contained in 'Data-centric programming languages' already. 'MUMPS' is the last entity. It is also already contained in 'Data-centric programming languages'. Since all entities in 'Persistent programming languages' either already exist in 'Data-centric programming languages' or can be moved there the category can be dissolved.

3.3.10 Chain of Inheritance

In Wikipedia a category may have members, which are just related to the covered topic and no Is-a or conforms relation can be derived. Therefore long chains of has-subcategory relations may exist leading to topics that do not belong to the target domain. Based on the anomaly **Chain of Inheritance** from Baumeister et al. [38] we also try to detect long chains of has-subcategory relations in the extracted ontology.

As stated beforehand we were able to find a way through has-subcategory relations from the category 'Computer languages' to the category 'Government of Nazi Germany'. These chains pose a problem to extraction processes that do not consider the whole Wikipedia dump. Therefore such an analysis could be made beforehand by executing a corresponding SPARQL query at DBpedia and try to detect long chains. This way categories can be identified that do not relate to the target domain to a sufficient degree and can be excluded in the extraction process as we proposed earlier. Otherwise the whole category and its subcategories that do not belong to the domain can be abandoned later.

Based on observations in a metric analysis we chose to mark categories as suspicious if they are at least seven has-subcategory relations away from the

root category. A corresponding formula can be stated as follows:

$$ChainOfInheritance = \{c \in C \mid depth(c) > 6\}$$

The corresponding SPARQL query in listing 3.10 takes advantage of the situation, that categories, which cannot be reached through has-subcategory relations from the root category, do not have an assigned value for `depth`. Since the depth value is a String it has to be converted to an integer.

```
PREFIX clonto: <http://myCLOnto.de/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?categoryname ?depth
WHERE {
        ?category clonto:depth ?depth .
        ?category clonto:name ?categoryname .
}
HAVING (xsd:integer(?depth)  > 6)
ORDER BY ?depth
```

Listing 3.10: SPARQL query detecting Chain of Inheritance

Several positive matches can be found, where one may quickly decide that the category does not belong to the domain of computer languages. Examples are 'Google Street View', which contains articles related to Google's Street view project in various countries, 'UIQ 3 Phones' capturing a row of mobile phones that use the UIQ technology based on Symbian OS, and 'GNOME Games', which has several entities corresponding to games that are part of free and open-source GNOME desktop environment.

A discussable match is the category 'Video player software that uses GTK+', which corresponds to software using a specific library. Since the category tree from computer languages has many software related categories we already stated that it is a design choice whether such a category should be integrated in a computer languages concerned ontology or not.

'Free theorem provers' with a depth of seven is a category that poses an example of a negative match. This category can be found in the group of 'Formal method tools'. Therefore it should be kept in the ontology since there is a high degree of relatedness to computer languages.

One may notice that measures to improve the quality of the ontology involve the whole abandonment of a category. That includes the deletion of has-subcategory relations to the target category and the removal of all elements and their relations that represent separate subgraphs in the ontology afterwards.

3.3.11 Semantically Distant Category

In the last subsection we suggested that categories, which are too far away from the root category may not be related enough to the target domain. This time we want to strengthen our search for semantically distant elements. Therefore we propose bad smells here and in the next subsection, which try to identify semantically distant categories and entities through an analysis of their containing categories.

This bad smell is inspired by the results of **Chain of Inheritance** discussed in the last subsection and the observations from **Overcategorization**, where the matching entity 'John von Neumann' had many containing categories that do not belong to the target domain. It states that an improvement may be necessary if a contained category has more unrelated supercategories that related ones.

Our detection mechanism is metric based. It counts the number of containing categories that are reachable from the root category and compares it to the number of containing categories that are not reachable from the root category. If the number of unrelated categories is higher we suspect the category to be semantically distant itself. In order to ease the corresponding formula and make it more abstract, we use 'root' as the variable which denotes the root category. Thus it does not only fit an analysis of an extracted computer languages ontology. A corresponding formula can be stated as follows:

$$\begin{aligned} SemanticallyDistantCategory = \{c \in C \mid & \, hasSubCategory^{-1}(c) = C_{super} \\ & \wedge hasSubCategory^{*}(root) = C_{root} \\ & \wedge |C_{super} \setminus C_{root}| > |C_{super} \cap C_{root}|\} \end{aligned}$$

The corresponding SPARQL query in listing 3.11 uses two subqueries and the situation, that unreachable categories do not have an assigned depth value.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?categoryname  (COUNT(?dcat) as ?howManyd)
(COUNT(?rcat) as ?howManyr)
WHERE {
    ?category clonto:name ?categoryname .
    {SELECT DISTINCT ?category ?dcat
     WHERE {
        ?category ^clonto:hasSubCategory ?dcat .
        FILTER NOT EXISTS{ ?dcat clonto:depth ?depth . } }
    }
    UNION
    {SELECT DISTINCT ?category ?rcat
     WHERE {
        ?category ^clonto:hasSubCategory ?rcat .
        ?rcat clonto:depth ?depth . }
    }
}
GROUP BY ?categoryname
HAVING(?howManyd > ?howManyr)
ORDER BY ?howManyd
```

Listing 3.11: SPARQL query detecting Semantically Distant Category

Several positive matches can be found. The category 'GNOME Games' has two distant supercategories namely 'Linux games' and 'Open-source video games' and one reachable supercategory 'GNOME Applications'. Since games may not be included, because their articles do not provide many details about their implementation, this category can be abandoned.

A stronger difference in numbers can be found in the category 'OpenStreetMap' concerned with related entities to the project 'OpenStreetMap', which is a collaborative project to create a free editable map of the world. It has ten distant supercategories and two reachable supercategories. The category itself may be excluded from the ontology since it is not relevant enough to the domain of computer languages. It can only be reached in a forward path for example through 'XML-based standards' to 'Web services' and its containing category 'Web mapping'. 'Web mapping' also fits the bad smell of a distant category and may be abandoned as well.

A negative edge case is presented by the category 'Denotational semantics'. It has four distant and three reachable supercategories. Denotational semantics are definitely strongly related to the domain of computer languages. A redundant has-subcategory relation is also involved since 'Concurrency (computer science)' and its subcategory 'Concurrent computing' are annotated as supercategories. Both are distant categories. After removing another redundancy namely to 'Logic in computer science' whose subcategory 'Programming language semantics' is already a supercategory it is not a match any longer. This is an example, which shows that many bad smell analyses have to be repeated after some improvement measures were executed.

A less improvable match is the category 'Qt (framework)'. As a framework it represents a piece of software. It is contained in many diverse categories concerned with software and libraries, such as 'X-based libraries', 'Embedded Linux', 'Wayland' or 'X Window System'. If the ontology engineer decided to exclude software related entities and categories, this category should be removed as well for the sake of consistency.

The examples showed that one may have to watch out for redundant has-subcategory relations before making a decision. An improvement may involve the abandonment of the category in analogy to the improvement for **Chain of Inheritance**.

3.3.12 Semantically Distant Entity

In order to provide further separation of concern we discussed semantically distant categories in the last subsection and now succeed with semantically distant entities. The bad smell **Semantically Distant Entity** is based on the same ideas.

The detection mechanism works in analogy to the one for **Semantically Distant Category**. This time we are targeting entities and the proportions between reachable and distant containing categories in the same way as above. A corresponding formula for this bad smell can be stated reusing `root` to denote the root category of the extracted ontology:

$$\begin{aligned} SemanticallyDistantEntity = \{e \in E \mid & \, hasEntity^{-1}(c) = C_{super} \\ & \wedge hasSubCategory^{*}(root) = C_{root} \\ & \wedge |C_{super} \setminus C_{root}| > |C_{super} \cap C_{root}|\} \end{aligned}$$

The corresponding SPARQL query can be found in listing 3.12, where has-subcategory relations were replaced by has-entity relations and an entity is selected instead of a category.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?ename  (COUNT(?dcat) as ?howManyd) (COUNT(?rcat) as ?
    howManyr)
WHERE {
    {SELECT DISTINCT ?entity ?dcat
     WHERE {
        ?entity ^clonto:hasEntity ?dcat .
        FILTER NOT EXISTS{ ?dcat clonto:depth ?depth . } }
    }
    UNION
    {SELECT DISTINCT ?entity ?rcat
     WHERE {
        ?entity ^clonto:hasEntity ?rcat .
        ?rcat clonto:depth ?depth . }
    }
    ?entity clonto:name ?ename .
}
GROUP BY ?ename
HAVING(?howManyd > ?howManyr)
ORDER BY ?howManyd
```

Listing 3.12: SPARQL query detecting Semantically Distant Entity

'Angry Birds (video game)' has 22 distant and only two reachable containing categories. The corresponding article does not give much implementation details. Instead the focus is on historical data and game specific analysis. Therefore it may be removed from the ontology since its relatedness is weak.

A similar match is 'Facebook'. Even though Facebook may pose an example of a web application, where many different programming languages are involved, its infobox only contains few implementation specific information, for instance involved programming languages. Under the attribute name 'Written in' the languages C++, PHP and D can be found. Therefore it remains the choice of an ontology engineer to keep it in the ontology or remove it.

A negative match is presented by the entity 'Turing machine'. Since it primarily involves a theoretical construct seven distant supercategories can be found. On the other hand it has four reachable categories, where one is 'Formal languages'. Since we are concerned with Computer languages, formal languages, like Turing machines, should be considered for a computer languages ontology as well.

Another negative match is 'AMOS (programming language)'. It belongs to 'BASIC programming language family' since AMOS BASIC is a dialect of the BASIC programming language implemented on the Amiga computer. Its two other supercategories are 'Video game development software' and 'Amiga development software'. One may argue that a programming language itself is not a piece of software and remove corresponding has-entity relations.

The examples showed that multiple possibilities for improvements exist. Most entities that show a large difference between the number of reachable and distant categories can probably be abandoned completely. Only in special cases one may consider removing a single has-entity relations after a precise analysis of the covered topic and the expressiveness of the entity.

3.3.13 Twin Categories

Rosenfeld et al. [36] propose the bad smell **Twin Categories**. It denotes the situation that two categories contain a large number of common entities. Previous observations showed that especially entities, whose articles need further clarifications, may belong to categories that are insufficient. Therefore we also propose an adaption of **Twin Categories**.

The detection mechanism searches for two categories that have at least nine common entities. The number was chosen randomly in order to experiment with the idea from Rosenfeld et al. [36] on the extracted ontology. In order to prevent capturing more redundancies the detection mechanism excludes pairs, where one category may reach the other through has-subcategory relations. A corresponding formula may be expressed as follows:

$$TwinCategories = \{(c_1, c_2) \mid c_1, c_2 inC \wedge c_1 \notin hasSubCategory^*(c_2) \\ \wedge c_2 \notin hasSubCategory^*(c_1) \wedge |hasEntity(c_1) \cap hasEntity(c_2)| > 8\}$$

A corresponding SPARQL query is placed in listing 3.13. Since a metric is difficult to formulate, that guides the choice of a number for the amount of common entities, one may analyze the extracted ontology by starting to inspect the twin categories with the highest number of common entities. Therefore the SPARQL query also provides an order by this number.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?cat1name ?cat2name (COUNT(?entity) as ?howManye)
WHERE {
        ?cat1 clonto:hasEntity ?entity .
        ?cat2 clonto:hasEntity ?entity .
        FILTER NOT EXISTS{?cat1 clonto:hasSubCategory* ?cat2}
        FILTER NOT EXISTS{?cat2 clonto:hasSubCategory+ ?cat1}
        ?cat1 clonto:name ?cat1name .
        ?cat2 clonto:name ?cat2name .
}
GROUP BY ?cat1name ?cat2name
HAVING (?howManye > 8)
ORDER BY ?howManye
```

Listing 3.13: SPARQL query detecting Twin Categories

The categories 'Scripting languages' and 'Free compilers and interpreters' have 24 entities in common. Articles for scripting languages may cover a language and its corresponding compiler or interpreter together. Therefore such

categories appear together. In an ontology one may try to provide a better separation of concern and separate the interpreter or compiler from the actual language concerned entity. For example the entity 'Perl' is related to the twin categories. An interpreter exists for Perl that is written in C. In order to improve the quality, one may introduce a new entity. It may contain only the relevant information concerning the interpreter implementation and thereby separate this information from the information about the actual language and its provided constructs. This kind of fix could be applied to all cases, where a separation of concern is feasible.

Another sample is displayed by the categories 'Free software programmed in PHP' and 'Free content management systems'. These categories have 54 entities in common. This high number of common entities provides enough ground to introduce a new subcategory that improves the taxonomy called 'Free content management systems in PHP'. This is an example that twin categories may also point out that a new subcategory may improve the ontology's quality.

A negative match can be found at 'Functional languages' and 'Object-oriented programming languages'. 'Scala (programming language)' is a common entity since it follows both categories' represented programming paradigms. No improvements should be applied, when the twin categories just represent two disjoint properties.

In retrospect we saw that this bad smell can be fixed through providing separation of concern by extracting a new entity from an existing one and thus reducing twin category appearances. An alternative exists if the two categories offer the possibility of introducing a new subcategory in case they already offer a large number of common entities that make a new subcategory feasible.

3.3.14 Partition Error

After an analysis concerned with two categories that appear together often and presenting several promising results we also want to propose a related bad smell. Baumeister et al. [38] state the anomaly **Partition Error** which denotes the situation that a concept has two superclasses that have a common superclass. They argue that the partition at this point may be flawed. We adapt the idea for the extracted ontology.

The detection mechanism works as follows. It searches for two categories that have a common supercategory. Both categories either contain a common subcategory or entity. A corresponding formula is presented next:

$$\begin{aligned} PartitionError = \{(c_1, c_2, el) \mid & (el \in E \cup C) \wedge c_1, c_2 \in C \wedge c_1 \neq c_2 \\ & \wedge \exists c_s \in C : c_1, c_2 \in hasSubCategory(c_s) \\ & \wedge (el \in hasEntity(c_1) \cap hasEntity(c_2) \\ & \vee el \in hasSubCategory(c_1) \cap hasSubCategory(c_2))\} \end{aligned}$$

A corresponding SPARQL query using a path expression with an alternation can be found in listing 3.14.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?elementname ?categoryBname ?categoryCname ?categorySname
WHERE {
    ?categoryB clonto:hasEntity|clonto:hasSubCategory ?elementA .
    ?categoryC clonto:hasEntity|clonto:hasSubCategory ?elementA .
    FILTER(?categoryB != ?categoryC)
    ?categoryS clonto:hasSubCategory ?categoryB .
    ?categoryS clonto:hasSubCategory ?categoryC .
        ?elementA clonto:name ?elementname .
        ?categoryB clonto:name ?categoryBname .
        ?categoryC clonto:name ?categoryCname .
        ?categoryS clonto:name ?categorySname .
}
```

Listing 3.14: SPARQL query detecting Partition Error

An exemplary match can be found in the entity 'Erlang (programming language)'. It is contained in 'Concurrent programming languages' and 'Software by programming language', which have the common supercategory 'Programming languages'. Erlang is a programming language, but its article also contains information about its compiler. Therefore it is also assigned to a software related category. In an ontology for computer languages one may want to provide a separation of concern. Thus an improvement can be performed as follows. An entity concerned with the compiler namely 'Erlang (compiler)' can be extracted from the available entity. Attributes from the infobox were assigned to the entity in the extracted ontology that are only related to the compiler namely 'latest release version = 17.4' and 'latest release data = {{Start date and age|2014|12|11}}' formatted in the infobox with the name 'Stable release' and the value '17.4 / December 11, 2014; 3 months ago'. This information is only related to the compiler and can be moved to the extracted entity.

Another match is presented by 'ANTLR', which is part of 'Parser generators' and 'Free compilers and interpreters' having the common supercategory 'Compilers'. ANTLR is a parser generator, but it is not a compiler or interpreter itself. Compilers and interpreters may use ANTLR to generate a parser for a language. Therefore an ontology engineer has to decide whether to keep the has-entity relations. A possibility would be to just remove the has-entity relation from 'Free compilers and interpreters'. The relatedness in the taxonomy to compilers would not be lost since the category 'Parser generators' is still a subcategory of 'Compilers'.

The next match is also presented by 'Erlang (programming language)' since it is contained in the categories 'Declarative programming language' and 'Functional languages' having the common supercategory 'Programming languages'. Since the containing categories only diffuse programming languages in terms of a

paradigm the assignment itself is feasible, but a redundancy can be found. The category 'Programming languages' has 'Functional languages' as one of its direct subcategories, even though 'Functional languages' is reachable through 'Declarative programming languages', which is a direct subcategory of 'Programming languages' as well.

In retrospect the examples showed various improvable aspects by analyzing matches to this kind of bad smell. Even though many matches may exist, where this bad smell does not imply a necessary improvement, it can be used to analyze the usage of categories that have a common supercategory. The examples showed that redundancies can be found, which can be removed. The first example here also showed that there may be detectable possibilities for extracting a new entity and thereby improving the ontology's taxonomy.

3.3.15 Missing Category By Partial Name

After trying to identify structural issues we also want to inspect name related inconsistencies. In the course of our analysis we found several missing relations. For example the entity 'Control flow graph' cannot be reached through the category 'Control flow', even though the category's name is part of the entity's name. Since the idea of using name pattern exists [42], names in the extracted ontology are also put to use. Here we use the idea to identify missing relations.

The bad smell **Missing Category By Partial Name** refers to the situation, where there is an entity that cannot be reached from a category, though the category's name is part of the entity's name. A corresponding formula uses $name(el)$ to retrieve the name of a category, entity, attributeset or attribute. It further uses $contains(s1, s2)$ to state that the String s1 includes the String s2.

$$\begin{aligned} MissingCategory = \{(e,c) \mid\ & e \in E \wedge c \in C \wedge\ \nexists c_{sub} \in C : \\ & c_{sub} \in hasSubCategory^*(c) : \\ & e \in hasEntity(c_{sub}) \wedge contains(name(e), name(c))\} \end{aligned}$$

A corresponding SPARQL query is stated in listing 3.15. SPARQL provides the function `contains()` that can be put inside a filter to express that its second String parameter is contained in the first String parameter.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT DISTINCT ?cname ?ename
WHERE {
    ?category clonto:depth ?depth .
    ?category clonto:name ?cname .
    FILTER(regex(STR(?category),"Category"))
    FILTER(regex(STR(?entity),"Entity"))
    ?entity clonto:name ?ename .
    FILTER contains(?ename, ?cname)
    FILTER NOT EXISTS{
        ?category clonto:hasSubCategory*/clonto:hasEntity ?entity }
}
ORDER BY ?category
```

Listing 3.15: SPARQL query detecting Missing Category By Partial Name

A positive match can be found at the entity 'JSON-WSP', which is a web-service protocol that uses JSON for service description, requests and responses. Though it uses JSON it is not contained in the category 'JSON'. An ontology engineer may decide to add a has-entity relation from the category 'JSON' to the entity 'JSON-WSP' since other protocols, e.g. 'JSON-RPC', are part of the category as well. As already discussed one may also choose not to add it to the category since it relates to a usage relation, but then for the sake of consistency one has to remove the protocols that use JSON from the category 'JSON'. Therefore the problem leads back to the design choice whether to express usage relations through the taxonomy or not

A negative match can be found at the category 'C (programming language)' and the entity 'ABC (programming language)'. ABC is not related to C. Therefore no relation should be added at this point.

Missing Category By Partial Name represents a bad smell, which tells that there should be an existing way from the category to an entity. Therefore the improvement is always the same. The decision has to be based on the knowledge whether the entity representing the article is related to the topic covered by the matching category. If a relatedness is existent either a has-entity relation has to be added to the category or a fitting subcategory.

3.3.16 Inconsistent Attributeset Topic - Category Name

In this subsection we proceed with the goal to analyze consistency in an extracted ontology. Our next targets are attributesets. An attributeset is assigned the name of the infobox template, e.g. 'programming language' or 'software'. It is always assigned to exactly one entity. The bad smell **Inconsistent Attributeset Topic - Category Name (IAN-CN)** denotes the situation when the entity is not contained in a category that is somehow related to the infobox

template's name. It is inspired by the idea of using name pattern [42] and our own observations.

In order to find such inconsistencies the detection mechanism searches for an entity and its attributeset. A match is found if the entity is not part of any category, where the category's name contains the attributeset's topic. The corresponding formula uses AS to denote the set of all attributesets, $hasAS(e)$ as the function the returns the attributesets assigned to the parameter entity and $topic(as)$ returns an attributeset's topic.

$$IAN-CN = \{(e, as) \mid e \in E \wedge as \in hasAS(e) \wedge \nexists c \in C : hasSubCategory^*(c) = C_{sub} \wedge e \in haveEntities(C_{sub}) \wedge contains(name(c), topic(as))\}$$

A corresponding SPARQL query is placed in listing 3.16.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT ?entityname ?atsettopic
WHERE {
    ?entity clonto:hasAttributeSet/clonto:topic ?atsettopic.
    FILTER NOT EXISTS{
        ?categoryt clonto:hasEntity ?entity .
        ?category clonto:hasSubCategory* ?categoryt .
        ?category clonto:name ?name .
        FILTER(regex(?name, ?atsettopic, "i")) }
    ?entity clonto:name ?entityname .
}
```

Listing 3.16: SPARQL query detecting Inconsistent Attributeset Name - Category Name

A match can be found at the entity 'SystemVerilog'. Its attributeset uses the template named 'programming language'. SystemVerilog is not a programming language. Instead it represents a hardware description language. This problem may be cause by the lack of suitable infobox templates. The provided list of infobox templates [7] only contains an infobox template called 'programming language' for the field of 'Software development'. While this may be feasible for a Wikipedia article, since this name is not formatted in any way, a renaming should take place in an ontology. Either an ontology engineer starts by inventing an attributeset name for hardware description languages, like 'hardware description language', or one may try to increase the range of fitting entities by renaming it to 'computer languages'. We will discuss these improvements, various options and resulting possibilities further in Chapter 4.

Another exemplary match is the entity 'RetrievalWare'. It corresponds to an enterprise search engine implemented in C, C++ and Java. Its attributeset

[7] http://en.wikipedia.org/wiki/Wikipedia:WikiProject_Computing/Templates#Infoboxes

name equals 'software'. The entity has only one containing category namely 'Data search engines'. Here the match represents a clear sign that there is a lack of annotated categories. Since it is written in C, one would expect to find a category, such as 'C software'. In the extracted ontology an ontology engineer may therefore decide to add has-entity relations from the related categories concerned with software in the named programming languages.

A major problem is posed by short forms. The attributeset's name at the entity 'House (operating system)' is 'os', which corresponds to the short form for operating system. Since corresponding categories, such as 'Free software operating systems', use the full term it becomes a match to the detection mechanism.

Various improvement measures can be triggered by this smell. Sometimes it may hint at an undercategorized entity. Another case was presented by 'SystemVerilog', where the attributeset name did not possess a suited level of abstraction for an ontology and a rename for the topic may be executed by an ontology engineer.

3.3.17 Inconsistent Attributeset Topic - In Category

A variant to the last bad smell is presented in this subsection. Fahad and Qadir [41] stated the anomaly **Disjoint domain specified by subclass error**, which points to the situation, in which two disjoint subclasses of one concept exist, where the concept and one of the subclasses are of different domains. For example the concept may be 'Food' and disjoint subclasses with an appearing anomaly are 'Burger' and 'Cola'. The anomaly is triggered since 'Cola' is a drink and not food.

Inconsistent Attributeset Name - In Category tries to adapt this idea to the category graph. Whenever there are two entities of disjoint fields as part of a common category these entities are suspicious. Therefore the detection mechanism uses the given attributeset topic. When two entities are part of a common category, but have different attributeset topics, they become a match to the mechanism. A formula can be stated as follows, where the test for String equality for the name is not case sensitive:

$$\begin{aligned} IANIN = \{(e_1, e_2) \mid\ & e_1, e_2 \in E \\ & \wedge as_1 = hasAttributeSet(e_1) \wedge as_2 = hasAttributeSet(e_2) \\ & \wedge topic(as_1) \neq topic(as_2) \wedge \exists c \in C : e_1, e_2 \in hasSubCategory(c)\} \end{aligned}$$

A corresponding SPARQL query is placed in listing 3.17 , where a filter is used to assure the inequality of the attributeset names. Before a comparison the names are transformed to be completely in lower case.

```
PREFIX clonto: <http://myCLOnto.de/>
SELECT DISTINCT ?categoryname ?entityAname ?asettopic1 ?entityBname
     ?asettopic2
WHERE {
    ?entityA clonto:hasAttributeSet/clonto:topic ?asettopic1 .
    ?entityB clonto:hasAttributeSet/clonto:topic ?asettopic2 .
    FILTER(LCASE(?asettopic1)!=LCASE(?asettopic2) && ?entityA != ?
    entityB)
    ?category clonto:hasEntity ?entityA .
    ?category clonto:hasEntity ?entityB .
    ?category clonto:name ?categoryname .
    ?entityA clonto:name ?entityAname .
    ?entityB clonto:name ?entityBname .
}
ORDER BY ?category
```

Listing 3.17: SPARQL query detecting Inconsistent Attributeset Name - In Category

Several exemplary matches can be found in the category 'Compilers'. One would actually expect language implementations as compilers, but instead other domains are represented as well. The entity 'Edison Design Group' has an attributeset with an assigned name 'company'. In an ontology a company should not be in this category and therefore the has-entity relation should be deleted as well. Next the entity 'Lisp in Small Pieces' with the attributeset's name 'book' is a book and not an implemented compiler. Thus the has-entity relation here should be deleted as well.

More discussable matches are presented by eponymous categories, such as 'Ada (programming language)'. It contains the entity 'Ada lovelace' that represents the person. Therefore its attributeset is also named person. The Wikipedia guidelines state that an eponymous category may contain all entities that are mentioned in the corresponding article. Since this is not a normal category it remains the decision of the ontology engineer how to proceed with these matches. It may be changed by removing this entity from the category by deleting the has-entity relation. The same goes for the contained entity 'AdaCore' with the attributeset's name 'company', but then the question has to be stated what entities are supposed to stay in this kind of category or whether this category should better be abandoned completely for a very comprise ontology, where the relation to the named entities are not expressed through the hierarchy in the category graph.

A negative match can be found in the category 'Data modeling languages'. In this category exists a variety of attribute set names. The entity 'XML' has two attributesets named 'file format' and 'technology standard'. Another exemplary entity is 'SPARQL', whose attributeset's name is 'programming language'. The problem here is that no clear suitable infobox template exists in Wikipedia to

be used for data modeling languages. Therefore one possible measure would be to inspect all entities and try to introduce a consistent attributeset topic such that the entities stay in this category that represent data modeling languages, e.g. XML.

In retrospect one finds several possible improvements. Again we posed an example, where entities may have to be removed from a category. Others showed that renaming the topic improves the ontology as well. It all depends on what entities should remain. Sometimes the main entity of the category give a clue, but the problem is as shown in 'Data modeling languages' that there may be several insufficient topics.

3.3.18 Multi Topic

As we saw before one may try to extract an entity from another entity by considering its attributeset. In this subsection we want to propose another Wikipedia specific idea for a bad smell. Even though not many articles in Wikipedia have infoboxes, there are several articles that have multiple infoboxes. This may be caused by the fact, that more than one topic is covered in the article and thus multiple topics may be summarized in separate infoboxes. This idea is based on the guidelines for Wikipedia editors on creating standalone pages [8].

Multi Topic denotes the issue, that one entity has multiple attributesets. It is likely that one may extract an entity from a matching entity by inspecting its attributesets. An extracted ontology may be improved by providing a better separation of concern. A corresponding formula may be stated as follows:

$$MultiTopic = \{e \in E \mid |hasAttributeSet(e)| > 1\}$$

A corresponding SPARQL query is displayed in listing 3.18.

```
1 PREFIX clonto: <http://myCLOnto.de/>
2 SELECT ?ename (COUNT(?aset) as ?howMany)
3 WHERE {
4         ?entity clonto:hasAttributeSet ?aset .
5         ?entity clonto:name ?ename .
6 }
7 GROUP BY ?ename
8 HAVING (?howMany > 1)
9 ORDER BY ?howMany
```

Listing 3.18: SPARQL query detecting Multi Topic

A first exemplary match for the detection mechanism is 'MATLAB'. The entity has two attributesets, where the first one is named 'software' and the

[8] http://en.wikipedia.org/wiki/Wikipedia:Notability#Whether_to_create_standalone_pages

second one is named 'programming language'. Since MATLAB is a software in the sense of a mathematical environment and it provides a fourth generation programming language these concerns may be separated. An ontology may be improved through a separation of concern here. Therefore an ontology engineer may decide to extract an entity 'MATLAB (programming language)' and move the second attributeset from 'MATLAB' to the extracted entity. The original entity would not hold all information about MATLAB anymore. Therefore it may be renamed to 'MATLAB (software)'.

Another match is posed by 'XML'. The entity has two attributesets namely 'file format' and 'technology standard'. As a markup language it actually is a file format and a technology standard at the same time. Therefore this match may be a match that shows an insufficient stock of templates. There is no template that is specific for markup languages, which leads to editors putting multiple multiple infoboxes in one article so that every important information can be found in the infobox. In an ontology one may merge the two attributesets. Since both infoboxes have no attribute name in common no problems should arise for this example.

'Kontact' represents a personal information manager and groupware software. It consists of several separate components. In the corresponding article each component is explained in an own section and most sections have an attached infobox. This results in the article having seven infoboxes in total. For an extracted ontology that focuses on computer languages an extraction of entities for each resulting attributeset may be a little overboard. This would probably be more interesting if one were to derive a software focused ontology instead. At last it remains the ontology engineer's choice whether or not to take measures here.

The examples above showed several different causes for an entity having more than one entity. To sum up the improvement possibilities one may either unite attributesets into one or extract entities and move attributesets to the newly created entities. Specifically the last example showed that it remains the ontology engineer's choice whether something can be done to improve the quality or not. Thus the final message is again that nothing beats human intuition.

Chapter 4

Improving the extracted information

The previous chapter presented the possibility to analyze an extracted ontology from Wikipedia via bad smells. In Section 3.3 we discussed several examples that matched particular bad smells. We also named possible measures to improve the ontology. In this chapter we want to discuss these measures in depth.

Several types of measures can be distinguished. Some proposed improvements affect the structure of the ontology, but they do not affect the provided knowledge. These measures may be compared to refactorings that are discussed by Fowler et al. [14] to improve the design of existing source code. A source code refactoring does not change any behavior of the program. Thus we denote changes to an ontology that do not affect its knowledge as refactorings as well.

Other measures affect the contained knowledge of an ontology. They can be compared to filtering irrelevant knowledge in the ontology. The term pruning denotes this type of change. Conesa and Olivé show several possibilities to prune the Cyc ontology [23].

When talking about software it is easier to distinguish between semantic preserving transformations and semantic changing transformations. The removal of a class does not imply that behavior is lost. Instead the structure may sometimes be improved. Categories in Wikipedia are different. If any category is removed knowledge may be lost since the relation to the main article of the category does not exist anymore. Another problem is that a category's name may contain implicit knowledge. This implicit information is lost through removal as well.

A third type of measure can be distinguished from the others known as refining. This type also affects the existent knowledge, but instead of trying to improve the ontology by cleaning it up in a systematic way, it is concerned with

adding information to the ontology that cannot be derived from the existing knowledge in it. SOFIE is a system that allows automated ontology extension through natural language processing [47].

This chapter proceeds as follows. In Section 4.1 we give an overview on related work. Again we try to emphasize, which ideas were adapted to improve an extracted ontology from Wikipedia. Afterwards we show how to systematically improve an ontology concerning structural and semantical issues that were already mentioned in the previous chapter. Therefore we propose a pruning algorithm in Section 4.2 and a set of refactorings in Section 4.3. For each entry we tell when it should be applied, describe each transformation step and then apply it to one chosen example in the extracted ontology from the computer languages domain.

4.1 Related Work on improvement

In their book Fowler et al. offer guidelines to refactoring source code [14]. They explicitly state that a refactoring is concerned with quality improvements after the code has already been written. The goal of a source code refactoring is to improve maintainability and structure without changing the external behavior.

Besides the bad smells, which we covered earlier in Section 3.1, the book contains a refactoring catalog. Groups collect several related refactorings. For each refactoring a motivation, mechanics and sometimes a discussed example is given. The refactoring **Collapse Hierarchy** is motivated by noting the situation that a subclass may exist that is not adding any value. Therefore the subclass and its superclass may be merged. The corresponding mechanics are described as follows. At first the engineer is supposed to chose whether the subclass or the superclass is to be removed. Then all behavior and data has to be removed from the chosen class and added to the other class by reusing basic refactorings, like **Pull up field** and **Pull up method** or **Push down field** and **Push down method**. These basic refactorings move the fields and methods from one class to the other adapt existing references to them. After each move the project should be compiled and tested. At last the empty class has to be removed from a project. A final compile and test finishes the improvement.

The work by Gröner et al. on the semantic recognition of ontology refactoring is closely related to the proposed refactorings on source code [48]. They try to analyze two ontologies. These two ontologies may express the same knowledge, but their structure is different. In order to recognize such appearances they propose a way to search for ontology refactorings with the use of specific pattern. The refactorings are an adaption of source code refactorings. Each pattern describing such a refactoring is mentioned with a name, a problem description that characterizes a modeling structure of an ontology and indicates applicability, a remodeling steps description and finally an example that demonstrates technical details. Description Logic Reasoning is used for the ontology

comparison. In order to essentially allow a reasoning process the ontologies are combined at first. Therefore the classes that appear in both ontologies with the same name have to be renamed and assigned to a common superclass that inhabits the original name. After this combination several algorithms are used for the analysis.

Baumeister et al. present a framework for automated refactorings in diagnostic knowledge bases [15]. The framework's purpose is to support an engineer during the process of restructuring a knowledge base. Therefore they sketch several design anomalies first. The term anomalies is used to denote symptoms in a knowledge base for probable errors. **Lazy Knowledge Object** notes that an object is never or infrequently used in an environment. In order to simplify the knowledge base's design a refactoring called **Remove Diagnosis** or **Remove Question** may be used to remove the lazy object. Baumeister et al. point out that such a refactoring may cause conflicts within the related knowledge. Therefore an engineer has to decide how to deal with conflicted rules. Deletions can cause deficiencies, such as ambivalence or redundancies. The framework should therefore support the user in his decisions and it should allow a removal of deficient cases automatically.

Three activities in ontology improvement are identified in the work of Conesa and Olivé [23]. They distinguish the terms refactor, prune and refine in relation to ontologies. Refactoring is specified as improving only the structure of a knowledge base. Therefore the input ontology and the output ontology of a refactoring still contain the same knowledge. Pruning is specified as follows. The output ontology is a subset of the input ontology. It still includes the same conceptual schema. Concepts, which are in the input ontology, but not in the output ontology pose an empty extension in the information system and are thus deemed irrelevant. Refining is specified as a process, which adds concepts and relations. First one may check whether identified relevant knowledge exists in an ontology. If it does not exist already, the ontology is extended.

Conesa and Olivé focus on pruning concepts in a refined ontology. Therefore they propose a pruning algorithm that consists of three major steps. At first all **irrelevant concepts** and constraints are pruned from the ontology. In a second step the **unnecessary parents** that do not have any children corresponding to needed concepts are pruned as well. At last **generalization paths** are pruned, which are deemed unnecessary.

Refinement is further emphasized in Kylin [28]. Kylin is a machine learning system using Wikipedia infoboxes as training data. Its goal is to extract information from natural language text. The full potential can be reached by putting it in a cleanly structured ontology. The Kylin Ontology Generator (KOG) can then be used to refine Wikipedia's infobox-class ontology. This is realized by transferring it to a machine learning problem using Supported Vector Machines (SVM) and Markov Logic Networks. With this technology, trained machine learning algorithms on infobox data yield extractors, which can even

generate infoboxes for articles. KOG builds such an ontology by combining Wiki-infoboxes with WordNet using statistical-relational learning.

4.2 Pruning

This thesis is concerned with systematic qualitative improvements for an extracted ontology from Wikipedia. The refinement activity is out of scope since we do not want to extend the ontology with further knowledge at this point. Conesa and Olivé [23] propose to first refine, then prune and refactor an ontology at least. Therefore we also start with a proposed algorithm on pruning in this section.

Matches to detection mechanisms in Section 3.3 show there are elements or relations that present insufficient knowledge. An ontology engineer can use pruning to get rid of such resources and relations. The result is a subset of the input ontology that only contains categories and entities that are deemed relevant.

Our demotool that is provided at a GitHub repository also supports pruning. One may select one of the prunings that we are going to discuss here, display the context that it should be used in and execute it. The implemented transformations correspond to SPARQL Updates on a triplestore via Jena. In our demotool we use **Parameterized SPARQL Strings**[1] to insert user input into a SPARQL Update String. If a value has to be assigned to a variable in the Update String for a pruning or refactoring, the user is asked to insert the name into a dialogue. This human input replaces a variable, such as `name` in the **Parameterized SPARQL String**.

We propose an adaption of the pruning algorithm by Conesa and Olivé [23]. Therefore we specify three stages. The first stage covers the removal of irrelevant elements. In the second stage the elements that have become unreachable from the root category through the changes in the first stage are removed. As the third and last stage the correctness of the taxonomy is inspected. For each stage we name the motivating bad smells, the pruning mechanics and the underlying transformations implemented as SPARQL updates.

4.2.1 Prune irrelevant elements

Bad smells such as **Chain of Inheritance**, **Semantically Distant Category** and **Semantically Distant Entity** showed that entities and categories are included that are not strongly related to the target domain. If such an element is deemed to be irrelevant to the domain it can be removed.

The first pruning is **Abandon Category**. If a category is deemed irrelevant to the target domain it may be removed from the ontology. A category can be

[1] `http://jena.apache.org/documentation/query/parameterized-sparql-strings.html`

removed by deleting all relations targeting the it. Afterwards the category and several members may not be reachable anymore from the root category. Those members are completely removed from the ontology in the next stage.

Listing 4.1 shows the corresponding transformation as a SPARQL update. The variable `name` has to be replaced by user input. This update String also takes advantage of the URI's structure. A category resource's URI contains the keyword 'Category'. Since there may be categories that have the same name as an entity one has to specify that this SPARQL update only targets categories. This is realized with a filter that transforms the URI of the category into a String and searches for the corresponding keyword 'Category'.

```
1 PREFIX clonto: <http://myCLOnto.de/>
2 DELETE { ?n ?r ?cat . }
3 WHERE{
4   ?n ?r ?cat .
5   FILTER(regex(STR(?cat),"Category"))
6   ?cat <http://myCLOnto.de/name> ?name . }
```

Listing 4.1: SPARQL update deleting all incoming relations of a category.

The category 'Perl people' serves as an example. It is contained in the supercategories 'Perl', 'Free software programmers' and 'Free software people'. Ten entities and a subcategory with thirteen further entities are part of it. Since only one category namely 'Perl' is reachable from the root category and two supercategories are unreachable the category is a match to **Semantically Distant Category**.

An ontology engineer may decide not to integrate persons into the ontology. In that case the example category can be abandoned by removing all incoming relations. After such a removal an independent subgraph exists in the ontology. This is demonstrated in figure 4.1. In the figure the category 'Programming language families' is also shown to demonstrate that the category 'Perl' is the only category that is reachable from the root. It does not show all involved elements. Instead it should only give off the idea behind this pruning. In the picture categories are visualized via rectangles and entities via circles. Each element only contains the entity's name. The arrows display any kind of relation, since we do not care at this point what kind of relation exists between elements.

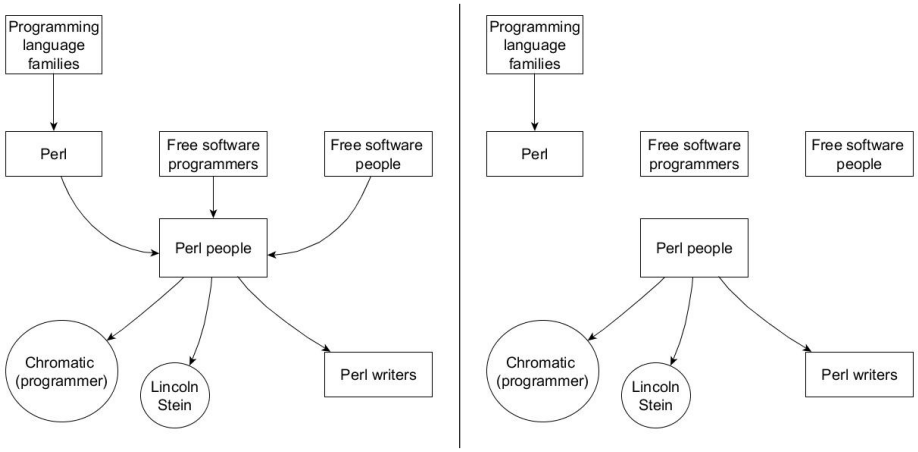

Figure 4.1: Hierarchy overview before and after **Abandon Category**

The second pruning is **Abandon Entity**. If an entity is not relevant to the domain it may be removed from the ontology. In order to prune it completely one has to remove all relations, where the entity is the target of the relation. After the relations are removed it is not reachable anymore through any forward path from the root category. The entity is removed in the cleaning process in the next stage.

Listing 4.2 shows the corresponding transformation as a SPARQL update. It works in analogy to the SPARQL String from listing 4.1. Since only entities are targeted here the placed filter searches for the keyword 'Entity' in the URI.

```
1 PREFIX clonto: <http://myCLOnto.de/>
2 DELETE { ?n ?r ?entity . }
3 WHERE{
4   ?n ?r ?entity .
5   FILTER(regex(STR(?entity),"Entity"))
6   ?entity clonto:name ?name . }
```

Listing 4.2: SPARQL update deleting all incoming relations of an entity.

Next we give an example of the entity 'Browserless Web'. The corresponding article is a stub and does not give off any important information for a computer languages domain. It is a member of three categories namely 'Web Services', 'Buzzwords' and 'Web technology'. 'Web services' is the only reachable category from the chosen root category. Since the number of distant categories is greater than the number of reachable categories the entity is a match to the bad smell **Semantically Distant Entity**.

An ontology engineer may thus decide to remove it from the ontology by removing all has-entity relations to it. As a result it becomes unreachable and will fall pray to the next prune step. The same goes for the containing categories that are not connected to the rest of the graph. Figure 4.2 displays the situation that is created through the pruning. It shows how 'Web technology' is still subcategory of another reachable category, but the other two categories do not have any assigned supercategory and no contained entity. In the figure the same visual syntax is used as in figure 4.1.

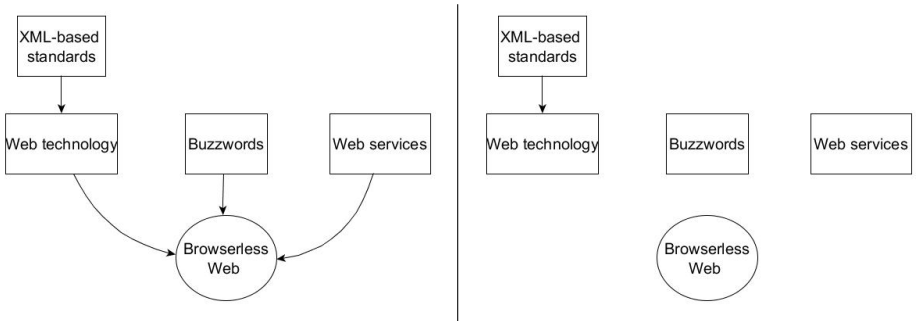

Figure 4.2: Abandon an entity

The bad smells **Bloated Category** and **Overcategorization** showed that there may be single irrelevant relations. Thus a pruning process also has to take these into consideration. Several small prunings can be identified to solve the apparent issues concerning positive matches to these bad smells.

The third pruning we want to propose is **Remove Has-Subcategory Relation**. If an irrelevant has-subcategory relation is found through analyses with bad smells such as **Overcategorization** or **Bloated Category**, an ontology engineer may decide to prune it from the ontology. A category that was only reachable through this has-subcategory relation also forms an unreachable subgraph and is removed completely in the next pruning stage.

A corresponding SPARQL update is placed in listing 4.3. One may notice, that this SPARQL update also has to be written as a Parameterized SPARQL String. Here two parameters exist. An ontology engineer has to replace the name of the containing supercategory `oldsupercatname` as well as the name of its subcategory `subcatname`.

```
PREFIX clonto: <http://myCLOnto.de/>
DELETE { ?supercat clonto:hasSubCategory ?subcat . }
WHERE{
  ?supercat clonto:hasSubCategory ?subcat .
  ?subcat clonto:name ?subcatname .
  ?supercat clonto:name ?oldsupercatname . }
```

Listing 4.3: SPARQL update deleting one has-subcategory relation.

Bad Smells such as **Bloated Category** or **Overcategorization** showed, that entities exist, which are irrelevant to certain categories in an ontology. The fourth pruning in this stage is **Remove Has-Entity Relation**. It removes the has-entity relation that is irrelevant in the ontology. Through this pruning an entity can become an orphan if it is not contained in any further categories. As a result it is deleted completely from the ontology in the next stage.

The implemented SPARQL update can be found in listing 4.4. It corresponds to a SPARQL Update String, where `catname` has to be replaced by the containing category's name and `entityname` by the name of the entity, which should be removed.

```
PREFIX clonto: <http://myCLOnto.de/>
DELETE { ?cat clonto:hasEntity ?entity . }
WHERE{
  ?cat clonto:hasEntity ?entity .
  ?cat clonto:name ?catname .
  ?entity clonto:name ?entityname . }
```

Listing 4.4: SPARQL update deleting one has-entity relation.

The bad smell **Bloated Category** can primarily be used to identify a category that has too many entities in it. For example the category 'Markup languages' has 214 entities. One of them is 'LaTeX2HTML'. LaTeX2HTML is actually a converter to transform LaTeX code into HTML. Thus it does not belong to the group of markup languages. The ontology engineer may thus select the pruning **Remove Has-Entity Relation**, specify 'Markup languages' as the containing category and 'LaTeX2HTML' as the entity that should be removed.

Bad smells such as **Speculative Generality** and **Lazy Category** showed that the hierarchy may be improved by comprising pieces of its structure. **Lazy Category** and **Speculative Generality** showed that there may be unnecessary categories with too less functionality such that they can be dissolved in an ontology. In comparison to the complete removal of categories as in the first major pruning stage this step keeps the category's members in the hierarchy such that only the category itself is removed in any case.

The pruning **Collapse Hierarchy** can be used to dissolve categories that may not be needed in a path from a category to the subcategory's members. At first a category has to be chosen that should be removed. Then all members of this category are added to the supercategories and removed from the chosen category. At last all ingoing relations to the category are removed, which results in the complete removal of the category.

The transformation to dissolve a category can be implemented as a SPARQL update. Listing 4.5 shows a corresponding transformation that adds subcategories and entities from the category that should be dissolved to its supercategories. Afterwards the category can be pruned through **Abandon Category**.

```
1 PREFIX clonto: <http://myCLOnto.de/>
2 INSERT { ?supercat clonto:hasSubCategory ?subcat .
3          ?supercat clonto:hasEntity ?entity . }
4 WHERE{ ?oldcat clonto:name ?oldcatname .
5        ?supercat clonto:hasSubCategory ?oldcat .
6        ?oldcat clonto:hasSubCategory ?subcat .
7        ?oldcat clonto:hasEntity ?entity . }
```

Listing 4.5: SPARQL update pulling members up.

'Automata (computation)' was already mentioned in Section 3.3 as a match to **Speculative Generality**. The ontology engineer may choose to dissolve this category. Its members are the subcategory 'Automata theory' and the entity 'Well-structured transition system'. Both are added to the only supercategory 'Models of computation'. Then all relations targeting 'Automata (computation)' are removed from the ontology.

4.2.2 Prune unreachable subgraphs

This pruning is concerned with an automatized cleanup of the ontology. All elements that are unreachable from the root category should be removed from the ontology.

Orphaned subgraphs exist out of several reasons. In the first stage several unreachable subgraphs may be created. This was demonstrated by displaying exemplary prunings for chosen elements in Figure 4.1 and in Figure 4.2.

Unreachable subgraphs also exist as supercategories. In the extraction process supercategories that cannot be reached were also extracted from articles and categories. Since these unreachable categories are probably not of any concern to the ontology they may be cleaned up in one fellow swoop together with the subgraphs created in the first pruning stage.

We propose a systematic cleanup of all orphaned elements in four steps. First the underlying cleanup algorithm starts at the categories that do not have

a supercategory. Thus all abandoned categories and all distant categories fall prey to this. All relations are removed that have the matching categories as their sources. In order to protect the root category itself one has to state that a category that should be removed should not have its name. This first step can be implemented with a SPARQL updated as shown in listing 4.6. Our proposed SPARQL implementation also makes use of the types implied by the URI to improve performance.

```
PREFIX clonto: <http://myCLOnto.de/>
DELETE { ?cat ?r ?n . }
WHERE{
  FILTER(regex(STR(?cat), "Category"))
  FILTER NOT EXISTS { ?cat ^clonto:hasSubCategory ?supercat }
  FILTER NOT EXISTS { ?cat clonto:name "Computer languages" }
  ?cat ?r ?n .
}
```

Listing 4.6: SPARQL update cleaning up categories

In a second step the cleanup algorithm proceeds with the entities that are not contained in any category anymore and removes all relations that have an orphaned entity as their source. The proposed SPARQL update is placed in listing 4.7.

```
PREFIX clonto: <http://myCLOnto.de/>
DELETE { ?entity ?r ?n . }
WHERE{
  FILTER(regex(STR(?entity), "Entity"))
  FILTER NOT EXISTS { ?entity ^clonto:hasEntity ?supercat }
  ?entity ?r ?n .
}
```

Listing 4.7: SPARQL update cleaning up entities

The third step targets the attributesets that became orphans due to the second step. Again all relations are removed that go out from these attributesets and they disappear from the ontology as a result. A corresponding SPARQL update can be found in listing 4.8.

```
PREFIX clonto: <http://myCLOnto.de/>
DELETE { ?atset ?r ?n . }
WHERE{
  FILTER(regex(STR(?atset), "AttributeSet"))
  FILTER NOT EXISTS { ?entity clonto:hasAttributeSet ?atset }
  ?atset ?r ?n .
}
```

Listing 4.8: SPARQL update cleaning up attributesets

At last the attributes that became orphans due to the third step have to removed as well. In analogy to the other steps the attributes' outgoing relations are pruned from the ontology. The SPARQL update for this last step concerning the cleanup can be found in listing 4.9. Here the resource's URI can no longer be used to recognize the type since the String 'Attribute' is also contained in 'AttributeSet'. Since attributes are the only elements that have a value relation it is included in the 'WHERE' expression to recognize the attribute type instead.

```
PREFIX clonto: <http://myCLOnto.de/>
DELETE { ?attribute ?r ?n . }
WHERE{
  ?attribute clonto:value ?v .
  FILTER NOT EXISTS { ?atset clonto:hasAttribute ?attribute }
  ?attribute ?r ?n .
}
```

Listing 4.9: SPARQL update cleaning up attributes

This cleanup procedure possibly has to be repeated multiple times until no relation is removed anymore. After the cleanup terminates all elements that became orphans due to the previous stage have disappeared from the ontology. The supercategories that we especially needed for the bad smells **Semantically Distant Entity**, **Semantically Distant Categorization** and **Overcategorization** are pruned as well.

4.2.3 Correct taxonomy

After the pruning algorithm removed all irrelevant relations and elements, the hierarchy becomes the explicit target. Olivé and Conesa [23] also suggest to prune the hierarchy in a third step, but there is one notable difference. They specified that only one single generalization path may exist between two concepts. The category graph in Wikipedia is different. A category may reach transitive members through multiple paths as explained in Section 3.2. Therefore we propose a slightly different activity in this third step.

In the first stage we removed irrelevant elements and relations. We also targeted the hierarchy through the pruning **Collapse Hierarchy**. The second stage cleaned up afterwards to remove unreachable subgraphs. Instead of further removing redundant elements we propose to examine the hierarchy concerning its correctness. Various semantic flaws may be identifiable in the hierarchy. Many have already been stated in covered as anomalies by Baumeister et al. [38].

An extracted ontology from Wikipedia as we specified it in Section 2.2 should correspond to an acyclic graph. The bad smell **Cycle** covers the situation, in which a cycle appears through multiple has-subcategory relations. This bad smell can be removed by using **Remove Has-Subcategory Relation**. A corresponding example in the ontology extracted from the domain of computer

languages was already discussed in Section 3.3. The removal of a cycle corresponds to a pruning since important information is lost. One has-subcategory relation does not exist afterwards.

In more complex ontologies other errors may be considered at this stage as well. One may resolve contradictions and further inconsistencies if enough information is available with the help of logical reasoners. This is out of the scope of this thesis especially since not all articles in Wikipedia have an infobox from which information can be extracted.

4.3 Refactoring

We use the term refactoring to denote a series of transformation that is knowledge preserving. Refactorings improve structural issues. The structure can be improved by making explicit knowledge implicit through the removal of redundant relations. Making implicit knowledge explicit is sometimes feasible as well. An entity's name can imply that it belongs to a group of entities, but the relation to the corresponding category does not yet exist. The knowledge about the part-of relation is stated through the name, but not yet through an explicit relation.

The refactorings in this section were mostly inspired by the work from Fowler et al. [14], who present a software refactoring catalog. They are further motivated through the previous analysis in Chapter 3. Each knowledge preserving improvement from Section 3.3 can be mapped to a refactoring.

We propose a small catalog, which consists of ten identified refactorings. Each refactoring is motivated through a rationale and its execution semantics are explained. An exemplary application is described for most refactorings. Any of the following improvements only corresponds to a refactoring if it is applied in a suitable context. Moving an entity from one category to another without any reason and context is obviously not semantics preserving and thus not a refactoring.

4.3.1 Remove Redundant Has-Entity

The exemplary matches for the bad smell **Redundant Entity Containment** show that there are redundant has-entity relations. If an entity belongs to a subcategory of a category, it belongs to the supercategory as well. This is implicit knowledge. An additional relation from the supercategory to the entity makes this knowledge explicit. In contrast to the pruning **Remove Has-Entity** removing redundant has-entity relations through the refactoring **Remove Redundant Has-Entity** preserves the knowledge in the ontology.

Remove Redundant Has-Entity needs a specified category and entity, where the has-entity relation should be removed. Afterwards a transformation

in analogy to **Remove Has-Entity** deletes the relation from the ontology. The following list shows the summarized mechanics:

1. Choose category
2. Choose entity
3. Remove has-entity relation

A SPARQL update, which removes a has-entity relation, is placed in listing 4.4.

An application example is posed by the entity 'Abstract Syntax' that is contained in the category 'Compiler construction' and additionally in its supercategory 'Programming language theory'. One may argue that 'Abstract Syntax' does not belong to 'Compiler construction' explicitly since it may also be needed at the implementation of interpreters. Therefore one may specify 'Compiler construction' as the category and 'Abstract Syntax' as the entity for the refactoring **Remove Redundant Has-Entity** and thereby remove the relation. An explicit relatedness is made implicit since 'Abstract Syntax' is still on the same level with 'Compiler Construction' within the containing category 'Programming language theory'.

4.3.2 Remove Redundant Has-Subcategory

The bad smell **Redundant Has-Subcategory** points out that there may be redundant has-subcategory relations. In Section 2.2 we stated that has-subcategory relations are transitive. Thus if a category is contained in a category and its supercategory as well the latter containment is redundant. Such a redundant relation can be removed by applying **Remove Redundant Has-Subcategory**.

To remove a redundant has-subcategory relation the ontology engineer has to specify a containing category and a subcategory. An existing has-subcategory relation can be removed by a corresponding transformation in analogy to the pruning **Remove Subcategory**. A corresponding transformation implemented as a SPARQL update is presented in listing 4.3. The mechanics can be summarized as follows:

1. Choose category
2. Choose subcategory
3. Remove has-subcategory relation

This refactoring may be applied in the following situation. 'XML parsers' is part of the category 'XML software' and its supercategory 'XML'. Since the part of relation to 'XML software' already implies that it belongs to 'XML' the category 'XML' can be specified as the containing category and 'XML parsers' as the subcategory for the refactoring **Remove Redundant Subcategory**.

4.3.3 Add Missing Category

The bad smell **Missing Category By Partial Name** an entity's name implied knowledge about its affiliation to certain topics. A has-entity relation may be missing to a corresponding category. Therefore we propose the refactoring **Add Missing Category** to insert this relation. This is an example for a refactoring that makes implicit knowledge in the entity's name explicit through adding a corresponding relation.

Add Missing Category needs a specified entity and a category. Then a transformation inserts a has-entity relation from the category to the specified entity. The mechanism is summarized in the following list:

1. Choose entity
2. Choose Category
3. Insert has-entity relation

Listing 4.10 presents a SPARQL update inserting a has-entity relation. Filters ensure the types of the variables `cat` representing the category and `entity` representing the entity. Before the update is executed `entityname` has to be replaced by chosen entity's name and `newcatname` by the chosen category's name.

```
PREFIX clonto: <http://myCLOnto.de/>
INSERT { ?cat clonto:hasEntity ?entity . }
WHERE{
  FILTER(regex(STR(?cat),"Category")
      && regex(STR(?entity),"Entity"))
  ?cat clonto:name ?newcatname .
  ?entity clonto:name ?entityname . }
```

Listing 4.10: SPARQL update inserting a has-entity relation

As an example for the computer languages domain an ontology engineer may choose the entity 'JSON-WSP' that is already mentioned in Section 3.3. Its name implies that it belongs to the group of elements strongly related to JSON. Therefore the chosen category corresponds to 'JSON'. The refactoring **Add Missing Category** can be applied to add the missing has-entity relation.

4.3.4 Change Topic

Since topic names are not unique for attributesets, attributesets have an additional topic relation besides the name relation. In case the topic is not sufficient for the attributeset's entity as discussed concerning the bad smell **Multi Topic**, it may be changed using the refactoring **Change Topic**.

For the transformations one has to specify the attributeset's name that corresponds to its index in the ontology as explained in Section 2.2. Then an ontology engineer names a new topic that should be assigned to the attributeset. Afterwards the old topic is overwritten by the new topic through a transformation. The Mechanics are summarized in the following list:

1. Choose attributeset's name
2. Choose attributeset's new topic
3. Overwrite the current topic with the new topic.

The listing 4.11 shows the SPARQL update which combines deletion and insertion of a relation. In this Parameterized SPARQL String the variable `atsetname` has to be replaced by the name and `newtopic` by the new topic before the update is executed.

```
1 PREFIX clonto: <http://myCLOnto.de/>
2 DELETE { ?atset clonto:topic ?oldtopic . }
3 INSERT { ?atset clonto:topic ?newtopic . }
4 WHERE {
5         ?atset clonto:topic ?oldtopic .
6 }
```

Listing 4.11: SPARQL update overwriting topic

As stated earlier the entity 'SystemVerilog' poses a match to **Inconsistent Attributeset name**. Its attributeset uses the template named 'programming language', though it actually corresponds to a hardware description language. In order to make this more explicit the attributeset may be renamed to 'hardware description language' by using **Change Topic**.

4.3.5 Unite Attributesets

The bad smell **Multi Topic** states that an entity can be considered suspicious if it has more than one attributeset. Thus multiple topics may be covered in the corresponding Wikipedia article or there was a lack of a suitable infobox template for this article. In order to improve the structure both attributesets can be merged in the ontology with the refactoring **Unite Attributesets**.

Unite Attributesets systematically merges two attributesets. At first one has to choose an entity. Then a main attributeset has to be chosen, which is preserved and renamed later. Afterwards the secondary attributeset that is supposed to be removed has to be selected. Next an expressive name has to be given to the main attributeset. In a subsequent step all attributes from the secondary attributeset are added to the main attributeset. At last the secondary attributeset is removed. Knowledge is preserved since the name of the new attributeset covers the knowledge of both previously existing attributesets. The following list summarizes the mechanism:

1. Choose Entity
2. Choose secondary attributeset
3. Choose main attributeset
4. Change main attributeset's topic with **Change Topic**
5. Add all relations from secondary attributeset to main attributeset
6. Remove secondary attributeset

Step three can be realized through the refactoring **Rename element**. The fourth step may be realized through a transformation, which moves all attributes from one attribute set to another. A possible SPARQL update is presented in listing 4.12 . While it is also possible to realize this with a loop and a transformations which move attributes one by one, we decided to implement a transformation that moves all attributes at once. In the Parameterized SPARQL String `atset1name` has to be replaced by the main attributeset's name and `atset2name` by the secondary attributeset's name. The last step conforms to a SPARQL update that removes the has-attributeset relation as demonstrated in listing 4.13. By replacing `atname` as a parameter one may also just move one attribute based on the same Parameterized SPARQL String.

```
PREFIX clonto: <http://myCLOnto.de/>
DELETE{ ?atset2 clonto:hasAttribute ?at . }
INSERT { ?atset1 clonto:hasAttribute ?at . }
WHERE{
  ?atset1 clonto:name ?atset1name .
  ?atset2 clonto:name ?atset2name .
  ?entity clonto:hasAttributeSet ?atset1 .
  ?entity clonto:hasAttributeSet ?atset2 .
  ?atset2 clonto:hasAttribute ?at .
  ?at clonto:name ?atname .}
```

Listing 4.12: SPARQL update moving all attributes from one attributeset to another.

```
PREFIX clonto: <http://myCLOnto.de/>
DELETE { ?entity clonto:hasAttributeSet ?attributeset . }
WHERE { ?attributeset clonto:name ?attributesetname .
        ?entity clonto:hasAttributeSet ?attributeset .
        ?entity clonto:name ?entityname . }
```

Listing 4.13: SPARQL update removing has-attributeset relation.

This refactoring may be applied to the entity 'XML'. It has two attributesets with the topics 'file format' and 'technology standard'. The attributeset

'technology standard' can be selected as the secondary attributeset and the attributeset 'file format' as the main attributeset. **Change Topic** can be applied to the main attributeset's topic in order to rename it to 'markup language'. Afterwards a transformation moves all attributes from 'technology standard' to 'markup language' and removes the secondary attributeset 'technology standard'.

4.3.6 Move Entity

An entity whose assignment to a category is suspicious, is for example a match to the bad smell **Partition Error**. In order to solve the responsible issues one has to perform more complex refactorings based on several distinct transformations. As part of a complex refactoring an entity may have to be moved from one category to another. This includes pushing an entity down to subcategories of a containing category or pulling it up to supercategories. Therefore we propose **Move Entity** is a basic refactoring that is later reused in more complex refactorings such as **Extract Subcategory**.

For **Move Entity** an entity has to be chosen, which should be moved. Next the unfit containing category has to be selected, where the entity should be removed. Then a new category has to be specified, where the entity should be added. Afterwards transformations remove the has-entity relation from the unfit category and insert it at the specified new category. The whole mechanism is summarized in the following list:

1. Choose entity
2. Choose unfit containing category
3. Choose new category
4. Delete entity from unfit category
5. Insert entity in new category

The transformation to remove a has-entity relation as a SPARQL update is placed in listing 4.4. A corresponding SPARQL update to insert a has-entity relation is present in listing 4.10.

4.3.7 Move Category

Since bad smells such as **Partition Error** target has-subcategory relations as well it may be necessary to move categories. The refactoring **Move Category** primarily appears in context of larger refactorings such as **Extract Subcategory**. Therefore it is also a basic reusable refactoring in analogy to **Move Entity**.

Move Category can be realized through a series of transformations. At first one has to specify the category that should be moved. Then the containg

supercategory has to specified where the category should be removed. Next one has to name a new supercatgory. In the end the transformation inserts the category in the new supercategory and removes it from the specified unfit supercategory. The mechanism is summarized in the following list:

1. Choose category
2. Choose unfit containing supercategory
3. Choose new supercategory
4. Insert category in new supercategory
5. Delete category from unfit supercategory

The transformation implemented as a SPARQL update that deletes a category from a supercategory was already displayed in listing 4.3. Listing 4.14 shows the update which inserts the category into a specified new supercategory. `subcatname` has to replaced by the category's name and `newsupercatname` by the supercategory it should be added to.

```
1 PREFIX clonto: <http://myCLOnto.de/>
2 INSERT { ?supercat clonto:hasSubCategory ?subcat . }
3 WHERE{
4   FILTER(regex(STR(?subcat),"Category") &&
5          regex(STR(?supercat),"Category"))
6   ?subcat clonto:name ?subcatname .
7   ?supercat clonto:name ?newsupercatname . }
```

Listing 4.14: SPARQL update inserting a has-subcategory relation

In analogy to **Move Entity** we also do not name any concrete example at this point. Both refactorings are used in an example for the refactoring **Extract subcategory**.

4.3.8 Rename Element

The bad smell **Speculative Generality** identifies categories that are not necessary in the domain. A creation may be cause by the speculation that it will improve the taxonomy at a later point ,when suitable members can be identified. An improvement possibility is to dissolve such categories. In the exemplary matches discussion for this bad smell in Section 3.3 renaming the category is mentioned as an alternative strategy. A rename may provide a category or an entity with a more explicit purpose. In order to rename any element we propose the refactoring **Rename Element**.

Rename Element works as follows. At first an ontology engineer has to specify the element's type since there exist categories, such as 'SQL', that have

the same name as their main entity. Next the engineer has to specify the current element's name and the new name that should be assigned to it. Afterwards the name is added to an element and the old name is removed. The following list sums up the overall mechanism:

1. Choose element's type
2. Choose old name
3. Choose new name
4. Overwrite old name by new name in element

In an implementation with an RDF triple store SPARQL updates can be used to implement the transformations to add or remove a name. All refactorings were implemented as **Parameterized SPARQL Strings**. Listing 4.15 shows the update that overwrites the old name `oldname` by a new name `newname` in the triplestore.

```
PREFIX clonto: <http://myCLOnto.de/>
DELETE { ?element clonto:name ?oldname . }
INSERT { ?element clonto:name ?newname . }
WHERE {
        FILTER(regex(STR(?element),?type)) .
        ?element clonto:name ?oldname .
}
```

Listing 4.15: SPARQL update inserting name

4.3.9 Extract Entity

Bad smell such as **Twin Categories**, **Partition Error** or **Multi Topic** showed that a more firm separation of concern can be provided for entities that may cover multiple topics. Thus a new entity can be generated from an existing one through the refactoring **Extract Entity**. Before we proceed with its mechanics, **Move Attributeset** and **Move Attribute** are introduced as refactorings that are only valid in **Extract Entity**'s context for now.

Move Attributeset

By itself the corresponding mechanics of **Move Attributeset** to not correspond to a refactoring since they are not semantics preserving. An attributeset always holds a set of information pieces. If it is removed from an entity information is lost no matter where it is added. The only exception is in the context of **Extract Entity** for now since an attributeset may not longer fit to the assigned entity and therefore it should be passed to the new entity.

The mechanism works as follows. An ontology engineer only has to specify the attributeset that should be removed from the source entity while the source

entity and the new entity were already specified in the context of **Extract Entity**. Then a transformation removes the attributeset from the source entity and adds it to the new entity.

In our SPARQL implementation only one new transformation has to be mentioned. The removal of an attributeset was already displayed in listing 4.13. Listing 4.16 shows the SPARQL update that adds the attributeset to a selected entity. The variable `entityname` has to be replaced by the entity's name and `atsetname` by the attributeset's name.

```
PREFIX clonto: <http://myCLOnto.de/>
INSERT { ?entity clonto:hasAttributeset ?atset . }
WHERE{
        ?entity clonto:name ?entityname .
        FILTER(regex(STR(?entity),"Entity"))
        ?atset clonto:name ?atsetname .
}
```

Listing 4.16: SPARQL update assigning an attributeset to a specified entity.

Move Attribute

By itself the transformations moving an attribute from one attributeset to another do also not correspond to a refactoring since they're not knowledge preserving. Instead they also can only be found in the context of **Extract Entity**.

When an ontology engineer wants to move an attribute from one attributeset to another he has to specify the source and the target attributeset. Afterwards corresponding transformations remove the attribute from the source attributeset and add it to the target attributeset.

As stated earlier the SPARQL update from listing 4.12 can be reused to move a single attribute as well. The variable `atname` has to be replaced by the targeted attribute's name.

Extract Entity Mechanics

Extract Entity requires the name of a source entity and a name for a new entity. A transformation introduces the new entity with the specified name. Further an engineer has to select categories and add the new entity to them through the refactoring **Add Missing Category**. He may also remove selected supercategories from the source entity that are not needed any longer. An attributeset can be selected and moved from the source entity to the new entity through **Move Attributeset** or a new attributeset can be introduced with a specified topic. Attributes from any existing attributeset in the source entity can be moved to the newly created attributeset by **Move Attribute**. The following list summarizes the mechanism:

1. Choose source entity.
2. Choose new entity name and index.
3. Introduce new entity.
4. Apply at least once **Add Missing Category** to new entity.
5. Remove new identifiable flawed has-entity relations targeting the source entity to preserve correctness.
6. Choose a name and a topic for a new attributeset .
7. Introduce new attributeset for the new entity.
8. Optionally apply **Move attributeset** from source entity to new entity.
9. Optionally apply **Move attribute** to transfer attributes from the source entity's attributeset to the new attributeset.

This refactoring reuses the several previously stated refactorings **Move attributeset** and **Add Missing Category** and the pruning **Remove Has-Entity** to restore semantic correctness. Further an entity and an attributeset can be introduced through a transformation. The corresponding implementation in SPARQL is demonstrated in listing 4.17. Notice that `element` has to be replaced by the URI for a new element and `newname` by a name, that should be assigned to it.

```
1 PREFIX clonto: <http://myCLOnto.de/>
2 INSERT { ?element clonto:name ?newname . }
3 WHERE{ }
```

Listing 4.17: SPARQL update introducing a new element with a given name.

Extract Entity can be applied to the entity 'Perl'. It covers two topics. On the one hand the language Perl is covered. On the other hand it gives off information about the interpreter as well. The attributeset contains the attribute named 'Implementation language' with the value 'C'. Though an interpreter exists for the language implemented in C one may also write a new interpreter in some other language. Therefore the structure may be improved by separation of concern.

Through the refactoring **Extract Entity** one may introduce a new entity called 'Perl (interpreter)'. Next one has to analyze the occurring categories. The categories 'Cross-platform software', 'Free compilers and interpreters', 'Free software programmed in C', 'Software using the Artistic license' and 'Unix programming tools' are specific to Perl's interpreter. 'Perl (interpreter)' may be added to them and 'Perl' may be removed from them. Both entities have to be placed in the category 'Perl' since they are related to the overall topic.

Next the attributeset from the original entity can be analyzed. One may identify three attributes that definitely belong to the interpreter namely 'Implementation language', 'OS' and 'License'. Therefore a new attributeset can be assigned to 'Perl (interpreter)'. It can be named 'software' since an interpreter is a piece of software and has common attributes. All previously mentioned attributes can now be moved from the attributeset of the original entity to the new attributeset through the refactoring **Move Attribute**.

At last one may rename the entity 'Perl' to 'Perl (programming language)' to make entity's purpose explicit. This improvement can be performed through the refactoring **Rename Element**.

4.3.10 Extract Subcategory

The bad smell **Twin Categories** showed that there are may be pairs of categories that appear quite often. Many entities may be in two common categories. From these two categories one may derive a new subcategory that collects multiple entities through the refactoring **Extract Subcategory**.

Extract Subcategory consists of several steps. At first a category has to be created with a specified name. Next the supercategories have to be specified and has-subcategory relations have to be elaborated. At last an engineer has to select the entities that are supposed to be added to the subcategory. Entities and categories from the supercategories can also be added by moving them via **Move Category** and **Move Entity**. Entities and subcategories may also have to be removed from the new category's supercategories in order to avoid redundancies through the refactoring **Remove Redundant Has-Entity** and **Remove Redundant Has-Subcategory**. The mechanics are summarized in the following list:

1. Choose new category name
2. Create category with new name
3. Choose supercategories
4. Add category to supercategories
5. Apply **Move Category** to chosen subcategories of the supercategories
6. Apply **Move Entity** to chosen entities
7. Apply **Remove Redundant Has-Subcategory** to remove redundant subcategories from the supercategories.
8. Apply **Remove Redundant Has-Entity** to remove redundant entities from the supercategories

In our implementation a category can be created through the SPARQL update from listing 4.17. The listing 4.14 presents a possible update that adds a supercategory to a specified category.

This refactoring may be applied to extract a subcategory from common entities in the categories 'Free software programmed in PHP' and 'Free content management systems'. These categories have 54 entities in common. Through **Extract Subcategory** one may introduce a category 'Free CMS implemented in PHP'. This category is added as a subcategory to 'Free software programmed in PHP' and 'Free content management systems'. All common entities may be added to it by applying **Move Entity** for the common entities in 'Free software programmed in PHP' and then applying **Remove Redundant Has-Entity** to remove the common entities from 'Free content management systems'.

Chapter 5

Conclusion

In Chapter 1 we state several guiding questions for this thesis. We summarize the answers to those questions in the following paragraphs.

The first question is: **How can we access Wikipedia's articles and category-graph in a suitable manner for our further research?**. We decided to access the Wikipedia API through requests with the help of the Bliki engine instead of downloading and using dump files. Our two main reasons here are that we only want to access a small subset of Wikipedia. Even by using the API our research results are still reproducible since the Wikipedia API offers access to older versions as well through specific parameters [1]. Our stated examples can also be reproduced by looking at the history of each corresponding Wikipedia page.

The second question is: **What information is necessary to build a small usable ontology in the domain of computer languages from Wikipedia? How can we extract this information from Wikipedia?**. A model is provided to capture the relations we want to extract. The main target is the category graph and thus the relations between categories and entities. Additionally our extraction process also targets the infoboxes that are mapped to attributesets. It also extracts the name of the used template for the infobox to derive a topic name. As shown through a metric analysis in Section 3.2 7373 categories, 12148 and 5971 attributesets entities can be attained already when one sets the the root category to 'Computer languages'.

The third primary question is: **Can we analyze the quality of a retrieved ontology by using well established approaches from the field of software engineering?**. In this thesis we start an analysis of the ontology based on metrics. These metrics are inspired by related work from general ontologies and object oriented design. The results form the basis for the further analysis via bad smells. They allow observations that point to possible issues.

[1] `http://www.mediawiki.org/wiki/API:Revisions`

These observable issues lead to the derivation of certain quality problems which can be explicitly specified through a set of bad smells. By implementing a detection mechanism via SPARQL queries we also showed that an automatized detection is possible, but one has to watch the meaning of a bad smell. Since a bad smell is a more abstract concept it is difficult to detect positive matches automatically. The statement, that automatized bad smell detection is quite complex, was already discussed for code related bad smells in [49]. In this thesis the queries proposing candidates for the bad smell are experimental. They can be evaluated and improved in future work.

The fourth question is **After a firm analysis can we improve the retrieved ontology in a systematic way?**. By following proposals on source code refactorings from Fowler et al. [14] and a proposed pruning algorithm from Conesa and Olivé [23] a systematic way to improve the quality of the extracted ontology can be set up. As suggested by Conesa and Olivé we start with a proposed pruning algorithm that comprises the contained knowledge to a sufficient minimum. Afterwards several refactorings to improve the ontology's structure can be applied that preserve the ontology's knowledge. All prunings and refactorings cover issues from the previous analysis and thus close the revision of the extracted ontology.

Even though the domain of computer languages was our primary target the whole revision process can also be applied to other domains. One has to watch that the extraction process does not access more in Wikipedia than the actual targeted domain, because there are has-subcategory relation chains leading from one large domain to another unrelated domain. Therefore several preparations may have to be made, before setting the root category and performing the extraction process, such as looking for paths to other unrelated domains. This can be realized via queries to DBpedia [2], which already provides a queriable extracted ontology from Wikipedia.

Our analysis can also be used in another way. It can also be used to maintain a domain in Wikipedia directly. The bad smell detection queries can be translated to corresponding queries to DBpedia and used to identify quality issues in current versions of Wikipedia. Bots could be written and sent off to do refactorings and prunings to Wikipedia as well. Using DBpedia one may also be able to propose new more complex bad smells that target semantic issues, like contradictions and further inconsistencies. The idea of integrating a tool for the detection of bad smells in semantic wikis was already proposed in [36].

In the course of this thesis we describe specific implementation examples that we also demonstrate in our tool available on GitHub [3]. Since RDF is a common format for ontologies, we also chose it. Other formats may be chosen as well. The implementation would have to be adapted, but the proposed analysis steps and improvement procedures stay the same.

[2] `http://dbpedia.org/`

[3] `https://github.com/MarcelH91/ReviseMiniCLOntology`

Bibliography

[1] Giaretta, P., Guarino, N.: Ontologies and knowledge bases towards a terminological clarification. Towards Very Large Knowledge Bases: Knowledge Building & Knowledge Sharing 1995 (1995) 25–32

[2] Favre, J., Lämmel, R., Varanovich, A.: Modeling the linguistic architecture of software products. In: Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings. (2012) 151–167

[3] Bollacker, K., Cook, R., Tufts, P.: Freebase: A shared database of structured general human knowledge. In: AAAI. Volume 7. (2007) 1962–1963

[4] Cimiano, P., Völker, J.: Text2onto. In: Natural Language Processing and Information Systems. Springer (2005) 227–238

[5] Wang, H., Jiang, X., Chia, L.T., Tan, A.H.: Wikipedia2onto- building concept ontology automatically, experimenting with web image retrieval. Informatica: An International Journal of Computing and Informatics **34**(3) (2010) 297–306

[6] Stvilia, B., Twidale, M.B., Smith, L.C., Gasser, L.: Information quality work organization in wikipedia. Journal of the American society for information science and technology **59**(6) (2008) 983–1001

[7] Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S.: DBpedia - A crystallization point for the Web of Data. Web Semantics: science, services and agents on the world wide web **7**(3) (2009) 154–165

[8] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: Dbpedia: A nucleus for a web of open data. Springer (2007)

[9] Morsey, M., Lehmann, J., Auer, S., Stadler, C., Hellmann, S.: Dbpedia and the live extraction of structured data from wikipedia. Program: electronic library and information systems **46**(2) (2012) 157–181

[10] Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: A large ontology from wikipedia and wordnet. Web Semantics: Science, Services and Agents on the World Wide Web **6**(3) (2008) 203–217

[11] Miller, G.A.: WordNet: a lexical database for English. Communications of the ACM **38**(11) (1995) 39–41

[12] Milne, D., Witten, I.H.: An open-source toolkit for mining wikipedia. Artificial Intelligence **194** (2013) 222–239

[13] Brank, J., Grobelnik, M., Mladenić, D.: A survey of ontology evaluation techniques. In: In In Proceedings of the Conference on Data Mining and Data Warehouses (SiKDD 2005, Citeseer (2005)

[14] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing code. Addison-Wesley (1999)

[15] Baumeister, J., Puppe, F., Seipel, D.: Refactoring methods for knowledge bases. In: Engineering Knowledge in the Age of the Semantic Web. Springer (2004) 157–171

[16] Baumeister, J., Seipel, D.: Smelly owls-design anomalies in ontologies. In: FLAIRS Conference. (2005) 215–220

[17] Jamali, S.M.: Object oriented metrics. A survey approach Technical report, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran (2006)

[18] Lämmel, R., Linke, R., Pek, E., Varanovich, A.: A framework profile of. net. In: Reverse Engineering (WCRE), 2011 18th Working Conference on, IEEE (2011) 141–150

[19] Zhang, H., Li, Y.F., Tan, H.B.K.: Measuring design complexity of semantic web ontologies. Journal of Systems and Software **83**(5) (2010) 803–814

[20] García, J., Jose'García-Peñalvo, F., Therón, R.: A survey on ontology metrics. In: Knowledge management, information systems, E-learning, and sustainability research. Springer (2010) 22–27

[21] Ponzetto, S.P., Strube, M.: Deriving a large scale taxonomy from wikipedia. In: AAAI. Volume 7. (2007) 1440–1445

[22] Conesa, J., de Palol, X., Olivé, A.: Building conceptual schemas by refining general ontologies. In: Database and Expert Systems Applications, Springer (2003) 693–702

[23] Conesa, J., Olivé, A.: Pruning ontologies in the development of conceptual schemas of information systems. In: Conceptual Modeling–ER 2004. Springer (2004) 122–135

[24] Medelyan, O., Legg, C.: Integrating cyc and wikipedia: Folksonomy meets rigorously defined common-sense. In: Proceedings of the WIKI-AI: Wikipedia and AI Workshop at the AAAI'08 Conference, Chicago, US. (2008)

[25] Zesch, T., Müller, C., Gurevych, I.: Extracting lexical semantic knowledge from wikipedia and wiktionary. In: Proceedings of the 6th International Conference on Language Resources and Evaluation, Marrakech, Morocco (May 2008) electronic proceedings.

[26] Syed, Z.S., Finin, T., Joshi, A.: Wikipedia as an ontology for describing documents. In: ICWSM. (2008)

[27] Cui, G., Lu, Q., Li, W., Chen, Y.R.: Corpus exploitation from wikipedia for ontology construction. In: LREC. (2008)

[28] Wu, F., Weld, D.S.: Automatically refining the wikipedia infobox ontology. In: Proceedings of the 17th international conference on World Wide Web, ACM (2008) 635–644

[29] Hu, X., Zhang, X., Lu, C., Park, E.K., Zhou, X.: Exploiting wikipedia as external knowledge for document clustering. In: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM (2009) 389–396

[30] Lämmel, R., Mosen, D., Varanovich, A.: Method and tool support for classifying software languages with Wikipedia. In: Software Language Engineering. Springer (2013) 249–259

[31] Yu, J., Thom, J.A., Tam, A.: Ontology evaluation using wikipedia categories for browsing. In: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management, ACM (2007) 223–232

[32] Maedche, A., Staab, S.: Measuring similarity between ontologies. In: Knowledge engineering and knowledge management: Ontologies and the semantic web. Springer (2002) 251–263

[33] Gómez-Pérez, A.: Towards a framework to verify knowledge sharing technology. Expert Systems with Applications **11**(4) (1996) 519–529

[34] Gangemi, A., Catenacci, C., Ciaramita, M., Lehmann, J., Gil, R., Bolici, F., Strignano, O.: Ontology evaluation and validation. An integrated formal model for the quality diagnostic task (2005) 30–36

[35] Tartir, S., Arpinar, I.B., Moore, M., Sheth, A.P., Aleman-Meza, B.: Ontoqa: Metric-based ontology quality analysis. (2005)

[36] Rosenfeld, M., Fernández, A., Díaz, A.: Semantic wiki refactoring. a strategy to assist semantic wiki evolution. In: Fifth Workshop on Semantic Wikis Linking Data and People 7th Extended Semantic Web Conference Hersonissos, Crete, Greece, June 2010, Citeseer (2010) 132

[37] Seipel, D., Baumeister, J.: Declarative methods for the evaluation of ontologies. Künstliche Intelligenz **18**(4) (2004) 51–57

[38] Baumeister, J., Seipel, D.: Verification and refactoring of ontologies with rules. In: Managing Knowledge in a World of Networks. Springer (2006) 82–95

[39] Preece, A.D., Shinghal, R.: Foundation and application of knowledge base verification. International journal of intelligent Systems **9**(8) (1994) 683–701

[40] Gómez-Pérez, A.: Evaluation of taxonomic knowledge in ontologies and knowledge bases. (1999)

[41] Fahad, M., Qadir, M.A.: A framework for ontology evaluation. ICCS Supplement **354** (2008) 149–158

[42] Šváb-Zamazal, O., Svátek, V.: Analysing ontological structures through name pattern tracking. In: Knowledge Engineering: Practice and Patterns. Springer (2008) 213–228

[43] Nirenburg, S., Wilks, Y.: What's in a symbol: ontology, representation and language. Journal of Experimental & Theoretical Artificial Intelligence **13**(1) (2001) 9–23

[44] Poveda Villalon, M., Suárez-Figueroa, M.C., Gómez-Pérez, A.: A double classification of common pitfalls in ontologies. (2010)

[45] Poveda, M., Suárez-Figueroa, M.C., Gómez-Pérez, A.: Common pitfalls in ontology development. In: Current Topics in Artificial Intelligence. Springer (2010) 91–100

[46] Sales, T.P., Barcelos, P.P.F., Guizzardi, G.: Identification of semantic anti-patterns in ontology-driven conceptual modeling via visual simulation. In: 4th International Workshop on Ontology-Driven Information Systems (ODISE 2012), Graz, Austria. (2012)

[47] Suchanek, F.M., Sozio, M., Weikum, G.: Sofie: a self-organizing framework for information extraction. In: Proceedings of the 18th international conference on World wide web, ACM (2009) 631–640

[48] Gröner, G., Parreiras, F.S., Staab, S.: Semantic recognition of ontology refactoring. In: The Semantic Web–ISWC 2010. Springer (2010) 273–288

[49] Fontana, F.A., Braione, P., Zanoni, M.: Automatic detection of bad smells in code: An experimental assessment. Journal of Object Technology **11**(2) (2012) 5–1