Pruning the Wikipedia Classification of Computer Languages

Marcel Heinz and Ralf Lämmel

Software Languages Team, http://softlang.wikidot.com/
University of Koblenz-Landau, Germany

Abstract. Wikipedia represents rich ontological knowledge that is also amenable to automated extraction. In particular, Wikipedia's classification graph may be used to provide a taxonomy within a field of interest. However, Wikipedia's classification graph has many issues making pruning necessary. In this paper, we assemble a suite of bad smells to identify and remove flawed classification relationships. The smells take into account Wikipedia's peculiarities, as they are described in guidelines. We organize the smells in a topology to optimize the pruning process. The approach is evaluated for a taxonomy of computer languages—in this field, Wikipedia arguably accounts for the most comprehensive knowledge base that exists.

Keywords: Computer languages. Wikipedia. Taxonomy. Bad smell. Pruning. Topology. Ontology debugging.

1 Introduction

An ontology is a knowledge model, where information is either manually added by domain experts or extracted from available sources, e.g., by using information retrieval approaches. A taxonomy is a common form of an ontology; it focuses on the classification of entities in a field of interest [14].

In this paper, we are concerned with the field of *computer languages or software languages*; see [7] for a discussion of these synonyms. We aim at a taxonomy of computer languages as an important building block of a more comprehensive ontology of software languages, technologies, and concepts. Our effort is well in line with Shilov et al.'s proposal [27] for a comprehensive knowledge portal for computer language classification. We consider Wikipedia the most comprehensive knowledge base for computer language classification—it is rooted by the category 'Computer languages'[1].

Taxonomy extraction from Wikipedia is challenged by a number of factors. Overall, an article's quality depends on authors' expertise and applied quality assurance mechanisms [28]. Further, the use of categories as classifiers leads to mixed expressions of equivalence, is-a and part-of relations [14]. In previous work [19], we developed an interactive tool 'WikiTax' so that a user can manually

[1] https://en.wikipedia.org/wiki/Category:Computer_languages

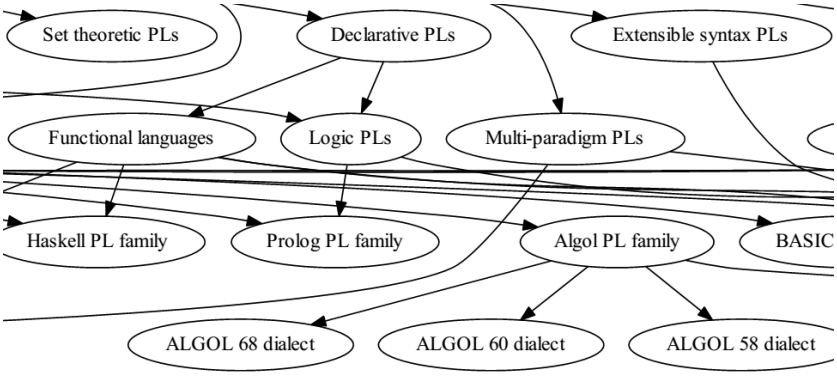

Fig. 1. A fragment of the computer languages taxonomy.

exclude subcategories, subject to informal criteria. Ultimately, this approach does not scale because of the number of issues and the difficulty to manage the informal criteria.

In this paper, we describe a semi-automated approach to obtain the taxonomy by extraction from Wikipedia and subsequent pruning. We aim here primarily at the removal of wrong facts. Pruning is guided by bad smells which in turn are ordered in a topology so that the pruning process is optimized. A bad smell describes a situation where a quality issue is very likely, but not always apparent. Thus, an expert's opinion is still relevant. The concept was coined by Fowler [13] for issues in software's source code and found its way into ontology evaluation [3]. We assemble a suite of bad smells that focus on Wikipedia's peculiarities. Bad smells are mapped to pruning procedures that a domain expert can choose from. Thus, our work is based on the following hypothesis: *There exist many issues in Wikipedia's category graph, when aiming at a taxonomy, but the issues are manageable in a systematic and scalable manner.*

While the proposed pruning approach is evaluated in this paper for computer classification only, we assume the overall approach to be valuable for any field of interest—as long as an existing Wikipedia category can account as the root classifier for the field of interest. For instance, we identified the category 'Computer programming tools'[2] as a foundation for programming technology classification.

Figure 1 plots a small fragment of the computer languages taxonomy. The ovals contain classifiers and the arrows denote subclassifier relationships. There are different kinds of classifiers that are important for a computer language classification in our sense, such as paradigm-based classifiers like 'Declarative PL' (where the string 'programming languages' was replaced by 'PL'), a few language families, and dialect-specific classifiers such as 'ALGOL 60 dialect'.

Contributions of the paper

– We propose a suite of bad smells based on related work on software engineering, ontology engineering, and semantic wikis while also introducing

[2] `https://en.wikipedia.org/wiki/Category:Computer_programming_tools`

smells that specifically take into account Wikipedia's peculiarities, as they are described in guidelines.
- We organize the smells in a topology to optimize the pruning process, as inspired by work on code smells and refactoring [21]. Smells with a larger pruning potential are considered first.
- We evaluate the approach with a comprehensive case study, in which we derive a computer languages taxonomy from Wikipedia. The data and tools of the case study as well as the actual taxonomy are available online[3].

Road-map of the paper Section 2 discusses related work. Section 3 presents the methodology. Section 4 presents the case study. Section 5 concludes the paper.

2 Related work

We discuss related work along dimensions of the field of interest in the case study (i.e., computer language classification) and the more general problem of taxonomy extraction and pruning.

Programming language classification The classification of programming languages (rather than computer languages more generally) has a long tradition in research. In early work, Floyd [12] introduced the notion of paradigms as a pattern-like way of thinking. Wikipedia's category graph covers paradigms, e.g., through the category 'Object-oriented programming languages'[4]. Each paradigm supports a set of concepts that are necessary for the solution to a problem. Van Roy et al. [29] offer a guide to choosing paradigms and a programming language that supports them based on a problem description.

Computer language classification In our previous work [19], we collected references to scholarly work on classifying computer languages—also languages other than programming languages, e.g., model transformation or business process modeling languages. Shilov et al. [27] address the broader problem of a computer language ontology with additional properties of languages—other than just classification along paradigms. They proposed the creation of the ontology in a collaborative manner by also integrating different knowledge bases. Unfortunately, this effort does not appear to be active at this point; no comprehensive knowledge base has come out of it. We consider the taxonomy of this paper as a key contribution to an emerging, comprehensive ontology.

Taxonomy extraction from Wikipedia Wikipedia is the target of data extraction in many ways, e.g., for the purpose of determining the semantics of article links [22,24]. Wang et al. [30] provide an approach to extract an animal taxonomy that is later used in an image search engine from Wikipedia. The approach is dependent on the structure of animal articles. We did not observe any structure of Wikipedia's computer language articles that could be leveraged for pruning.

[3] `http://softlang.uni-koblenz.de/wikionto/`

[4] `https://en.wikipedia.org/wiki/Category:Object-oriented_programming_languages`

Flati et al. [11] describe the retrieval of a bitaxonomy from Wikipedia's category graph and filter the entries using hypernym relations from the articles' text. The filter would remove valid classifying categories as observed for 'Java (programming language)'. More generally, there is no related work that could be used directly to remove wrong classification relationships within a field of interest from Wikipedia's category graph.

Ontology evaluation Evaluation of ontologies is an important part of an ontology development and maintenance process [25]. Most ontology evaluation techniques rely on the existence of another resource that contains exhaustive information. For computer languages, we have been unable to identify any such resource. Thus, we propose a criteria-based approach capable of revealing issues in a taxonomy without relying on other resources.

Ontology qualiy Ontology design patterns [1] describe best practices. Opposite to best practices are bad practices. Poveda-Villalón et al. [23] speak of pitfalls; their tool 'OOPS!' identifies pitfalls in OWL ontologies that include non-taxonomic information. The pitfall 'Including cycles in the hierarchy' aims at identifying cycles in a subtyping hierarchy; we also consider this pitfall in our approach.

In software engineering, bad practices in source code are captured as bad smells [13]. Further work in the context of software refactoring suggests to improve efficiency by analyzing multiple smells ordered in a derived topology [21]. Bad smells are also used in knowledge engineering. Baumeister et al. provides a series of work on bad smells denoted as 'anomalies' in diagnostic knowledge bases [2], in OWL based ontologies [3] and in verification of ontologies with respect to existing rules [4,5]. The proposed bad smell suites were further extended by Fahad and Qadir [6]. Roussey et al. [26] speak of antipatterns instead of bad smells or anomalies. We adopted and complemented existing bad smells to Wikipedia's peculiarities, as described in Section 3.

Ontology pruning Pruning may be based on merging two resources such as WordNet and any online encyclopedia [15] or alignment of a systematically pruned ontology to an expert derived ontology [20]. Our approach is specifically designed to cope with the challenge that there is no obvious second rich resource in the field of interest. Pruning may also be based on 'relevance' such that a too large ontology is narrowed down to the aspects of interest [16]. In contrast, our approach is specifically designed to remove wrong facts.

Taxonomy debugging In [18] Lambrix et al. suggest to repair missing is-a structures with algorithms based on existing knowledge in other external resources. Elsewhere, Lambrix and collaborators [14,17] present an approach to debug and align taxonomies that focuses on repairing missing and wrong classifications with respect to an ontology network. The candidates for missing is-a relations and mappings are validated by a domain expert. Further the idea of recommending repair actions and assigning each recommended repair action a priority is presented. In constrast, our work is focused on pruning and primarily smell-driven.

3 Methodology

We present an approach for pruning a taxonomy based on *bad smells*. Figure 2 presents the involved activities in a control-flow diagram with an initial and final node representing start and end. The first four overlapping activities influence each other and focus on exploring the category graph in Wikipedia. At first, a root category is selected from Wikipedia, such as 'Computer languages'. Next, the user has to carefully state inclusion and exclusion criteria for categories that he finds in the process. Further, the user defines a maximum depth d_{max}, since we noticed that at least for computer languages the subcategories start to not be feasible subclassifiers anymore after a certain level of depth. At last, the user has to try to carefully detect problematic categories for the automatic extraction and should be excluded (C_{ex}). Exploration can be performed directly on top of Wikipedia or by using the interactive 'WikiTax' tool [19].

We further discuss the exploration steps in Section 3.1. Afterwards, the automatized extraction of the initial taxonomy takes place that is later pruned based on a bad smell topology.

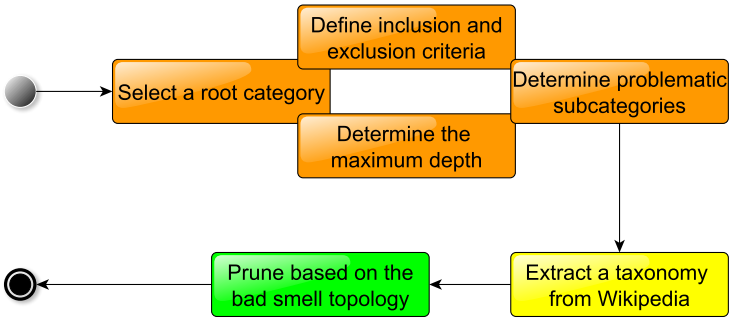

Fig. 2. Methodology towards a cleaned taxonomy.

3.1 Extraction from Wikipedia

Wikipedia's category graph offers a rich taxonomy that can be used to create a taxonomy. When and how to assign an article or a category to a category in Wikipedia is described in its guidelines[5]. Authors express these assignments in the Wiki markup of the concerned page[6]. In the resulting taxonomy Wikipedia's categories can be mapped to possible *classifiers* and the articles represent the classified *instances*. Thus, a subcategory relationship corresponds to *hasSubclassifier* relationship and article containment is viewed as a *classifies* relationship.

If one aims at extracting taxonomic information from Wikipedia, its peculiarities have to be taken into account. In an ontology, each entity represents

[5] `http://en.wikipedia.org/wiki/Wikipedia:Categorization`
[6] `http://en.wikipedia.org/wiki/Help:Wiki_markup`

exactly one object of the real world. This rule is not applied to Wikipedia. An article should cover more than one object, if the text flow and understandability is improved this way.[7] A category might have been assigned based on a smaller section inside the article, but the represented instance should not be classified by such a category that does not relate to the whole.

Wikipedia's guidelines describe a special category kind called 'Eponymous Category'[8]. These categories are proxies for entities; articles that are strongly related to the corresponding entity can be placed in the eponymous category. Since such a category does not correspond to a classifier in a proper way, our analysis pays attention to them.

After the extraction, the taxonomy contains additional information to enable the exploitation of certain patterns. For every extracted category and article we suggest to temporarily extract all direct supercategories as additional superclassifiers even if they are not reachable from the root. This allows further arguments on the semantic relevance at a later point.

3.2 Bad Smell Suite

This section presents our assembled bad smells. They were chosen such that they aim at finding semantic flaws and irrelevant elements that can be pruned. We avoid to formulate bad smells that only focus on structuring or design issues that would lead to necessary refactorings. For each bad smell, we tell its purpose, inspiring related work, a formal specification, and an example.

Bad smell *Eponymous Classifier*: The smell takes care of eponymous categories such as 'Java (programming language)', as discussed in Section 3.1. We search for classifiers, for which exists an instance with the same name. Our formulation ignores the existence of classifiers whose name is the plural form of an existing instance's name or a variation for now. (Notation: $name(e)$ returns the name of a given element e. T is the set of all classifiers in the taxonomy and I corresponds to the set of all instances in the taxonomy.) .

$$EponymousClassifier = \{t \in T \mid \exists i \in I.name(i) = name(t)\}$$

Bad smells *Semantically Distant Classifier* and *Instance*: Wikipedia's guidelines state that the annotation of categories should not be overused[9]. In our exploration, we found many classifiers and instances that were in our opinion neither subclassifiers nor instances of computer languages. We observed that nearly all of them have more unrelated (super-)classifiers than related ones. Inspired by semantic flaws formulated by Fahad and Qadir [6], we define smells targeting classifiers and instances with a measured semantic distance. The most complex cases are classifiers such as 'Computer algebra systems' that still contain relevant instances that need to be rescued before the classifier is pruned. (Notation: *hasSubclassifier* returns the set of direct subclassifiers for a given classifier; *classifies* returns the set of classified instances for a given classifier; *reachable*

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

[8] http://en.wikipedia.org/wiki/Wikipedia:Categorization#Eponymous_categories

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

states for a given element, whether it is an instance or subclassifier of the root in the taxonomy. T_r is the set of reachable classifiers for a given subclassifier or instance and T_u is the set of unreachable ones.)

$$\begin{aligned} \mathit{SemDistClassifier} = \{t \in T \mid \mathit{reachable}(t) \wedge \forall t_r \in T_r : t \in \mathit{hasSubclassifier}(t_r) \\ \wedge \mathit{reachable}(t_r) \wedge \forall t_u \in T_u : t \in \mathit{hasSubclassifier}(t_u) \wedge \neg \mathit{reachable}(t_u) \\ \wedge (|T_r| < |T_u|\} \end{aligned}$$

$$\begin{aligned} \mathit{SemDistInstance} = \{i \in I \mid \forall t_r \in T_r : i \in \mathit{classifies}(t_r) \wedge \mathit{reachable}(t_r) \\ \wedge \forall t_u \in T_u : i \in \mathit{classifies}(t_u) \wedge \neg \mathit{reachable}(t_u) \wedge (|T_r| < |T_u|\} \end{aligned}$$

Bad smells *Double Reachable Classifier* and *Instance*: These smells aim at finding semantically flawed relationships. Such a flaw may be suspected, if a classifier or an instance is reachable from two distinct direct subclassifiers of the root. They are inspired by Baumeister et al's [3] description on 'Partition Errors'. In the case study, with 'Computer languages' (CL) as the root, we observed exceptions for elements that are reachable through both 'Data modeling languages' and 'Markup languages', e.g.,'XML'. (Notation: $\mathit{hasSubclassifier}^*$ returns the set of subclassifier that are reachable from a given classifier including itself; $\mathit{classifies}^*$ works analogously for instances.)

$$\begin{aligned} \mathit{DoubleReachableClassifier} = \{t \in T \mid \exists \mathit{top}1, \mathit{top}2 \in \mathit{hasSubclassifier}(CL) : \\ \mathit{top}1 \neq \mathit{top}2 \wedge t \in \mathit{hasSubclassifier}^*(\mathit{top}1) \cap \mathit{hasSubclassifier}^*(\mathit{top}2) \\ \wedge (\neg \mathit{top}1 = \text{'Data modeling languages'} \vee \neg \mathit{top}2 = \text{'Markup languages'})\} \\ \mathit{DoubleReachableInstance} = \{e \in E \mid \exists \mathit{top}1, \mathit{top}2 \in \mathit{hasSubclassifier}(CL) : \\ \mathit{top}1 \neq \mathit{top}2 \wedge t \in \mathit{classifies}^*(\mathit{top}1) \cap \mathit{classifies}^*(\mathit{top}2) \\ \wedge (\neg \mathit{top}1 = \text{'Data modeling languages'} \vee \neg \mathit{top}2 = \text{'Markup languages'})\} \end{aligned}$$

Bad smell *Cyclic Classifier*: In ontologies, a cycle is a recognized flaw [3,23]. In the case of the extracted taxonomy from Wikipedia, cycles can only appear in subclassification relationships. Thus, all classifiers have to be analyzed if they are subclassifier of themselves. A cycle appears in subcategories of Wikipedia's computer languages category as 'Data-centric programming languages' and 'Persistent programming languages' are subcategories of each other.

$$\mathit{CyclicClassifier} = \{t \in T \mid (t \in \mathit{hasSubclassifier}^*(t)\}$$

Bad smell *Lazy Classifier*: The smell is inspired by Fowler's *Lazy Class* [13] and Baumeister's smell 'Lazy Knowledge Objects' [2]. We propose this smell as means of questioning the relevance of a classifier with less than n subclassifiers and instances in total. In our evaluation we assign seven to n based on our exploration experience.

$$\mathit{LazyClassifier} = \{t \in T \mid (|\mathit{classifies}(t)| + |\mathit{hasSubclassifier}(t)|) < n\}$$

	Lift Cycle	Abandon Type	Abandon Entity	Remove Subtype	Remove Instance
Eponymous Classifier		x			
Semantically Distant Classifier		x			
Semantically Distant Instance		x	x		
Double Reachable Classifier		x		x	
Double Reachable Instance			x	x	x
Cyclic Classifier	x				
Lazy Classifier		x			
Redundant hasSubclassifier				x	
Redundant Classifies					x

Fig. 3. Mapping smells on the left to relevant pruning/refactoring.

Bad smells *Redundant Classifies* and *has-Subclassifier* (relationship): Redundancy is a recognized flaw in ontologies [3]. We say that an instance relationship is redundant, if there is another instance relationship for the same element and subclassifier. Redundancy can be defined analogously for subclassifier relationships. (Notation: $hasSubclassifier^+$ returns the set of subclassifiers that are reachable from a given classifier excluding itself.)

$$\begin{aligned} Redundant\,Classifies = \{(t,e) \mid t \in T \wedge e \in E \wedge e \in classifies(t) \\ \wedge \exists t_s \in hasSubclassifier^+(t) : e \in classifies(t_s)\} \\ Redundant\,hasSubclassifier = \{(t_1,t_2) \mid t_1 \in T \wedge t_2 \in hasSubclassifier^+(t) \\ \wedge \exists t_3 \in hasSubclassifier^+(t_1) : t_2 \in hasSubclassifier(t_3)\} \end{aligned}$$

3.3 Transformations

The procedures leverage primitive automated prunings, but some rely on decisions to be made by the domain expert. Figure 3 gives an overview of our adapted pruning suite. We explain the suite with examples from the case study. For each pruning we describe the transformations that have to be executed. The columns are sorted by decreasing 'priority'. The leftmost applicable procedure should be preferred.

Transformations *Remove Subclassifier* and *Classifies* (relationship): We start with the most basic prunings that focus on removing a single relationship in a taxonomy. A flawed *hasSubclassifier* relationship can be detected by searching matches for *Double Reachable Classifier* and *Redundant Subclassifier* or in the course of fixing a match for *Double Reachable Instance*. For example, the classifier 'Page description markup languages' is a subclassifier of 'Markup languages' and 'Programming languages'. The responsible flawed *hasSubclassifier* relationship to be removed is the one between 'Page description languages' and 'Domain-specific programming languages'. A flawed *classifies* relationship can

be detected by scanning the taxonomy for *Double Reachable Instance* and *Redundant Classifies*. For example, 'XProc' (a XML transformation language for XML Pipelines) is falsely classified as a programming language through classifiers, such as 'Declarative programming languages'. Thus, multiple instance relationships have to be removed until the issue is resolved.
Transformation *Lift Cycle*: A cycle in the subtyping hierarchy can be resolved by searching and removing a flawed relationship in the cycle. E.g., 'Persistent programming languages' (These are programming languages, where objects can be persisted directly without using technology for other technological spaces.) and 'Data-centric programming languages' are subclassifiers of each other. We observed that persistent programming languages are data-centric, but not all data-centric programming languages are persistent programming languages, thereby suggesting a relationship to be removed.

Listing 1.1. A procedure to lift a cycle.

```
liftCycle(t::Classifier) {
        while (t in CyclicClassifier)
                (t2,sub) = identifyFlawedRelation(
                    CyclicClassifier)
                removeSubclass(t2,sub)
}
```

Transformation *Abandon Classifier*: This transformation models the complete removal of a classifier from the taxonomy, as identified by the smells *Distant Classifier*, *Eponymous Classifier*, *Semantically Distant Classifier*, *Lazy Classifier*, and *Double Reachable Classifier*. Irrelevant classifiers may be found by inpsecting the classifiers of a *Semantically Distant Instance* as well.

A classifier can just be removed, if it has no relevant instances or subclassifiers, such as 'Software that uses Qt'. More effort is required, if relevant subclassifier and instances can be identified. 'Ada (programming language)' is an eponymous classifier with various persons as instances and the irrelevant subclassifier 'Free software programmed in Ada', but the subclassifier 'Ada programming language family' should be maintained. One has to decide which subclassifiers and instances are relevant. If all are relevant, this procedure corresponds to collapsing the hierarchy known from software refactorings [13]. For example, the classifier 'SETL programming language family' only has 'SETL' as its instance. We question the value of the language family and collapse the classifier.

Listing 1.2. A procedure to abandon a classifier.

```
abandonClass(t::Classifier) {
  SupT = {ts in T| t in hasSubclassifier(ts)}
  I = {i in E| i in classifies(t)}
  SubT = {ts in T| ts in hasSubclassifier(t)}
  for each ts in SupT
    removeSubClass(ts,t)
  for each ts in SubT
    if relevant(ts)
      addSubClass(SupT, ts)
```

```
    else
      abandonClass(ts)
  for each i in I
    if relevant(i)
      addInstance(SupT,i)
    else
      abandonInstance(i)
}
```

Transformation *Abandon Instance*: The pruning removes an instance, as it possibly matches bad smells such as *Semantically Distant Instance* and *Double Reachable Instance*:

Listing 1.3. A procedure to abandon an instance.

```
abandonInstance(e::Instance) {
  T = {t in T| e in classifies(t)}
  for each t in T
    removeInstance(t,e)
}
```

3.4 Topology of Bad Smells

In initial experiments, we analyzed a retrieved taxonomy with each bad smell separately and encountered a lot of side effects. When we resolved one bad smell match, the involved entities did not appear in other bad smells anymore as well. Therefore, we decided to derive a topology inspired by previous work on source-code refactoring [21], which prescribes the order in bad smells are to be considered by the domain expert.

We describe the effect for each bad smell, when it is addressed before another smell—with respect to the assigned prunings and refactorings. A table for all such effects is displayed at Figure 4; a full documentation can be found in the repository and on the paper's website, as linked earlier.

We found two kinds of side effects between bad smells (bs_1, bs_2) with decreasing priority. 'Complete resolution' has the highest priority, where if a match of bs_1 is resolved, involved instances or classifiers do not appear in a match of bs_2. The second effect 'Ease Detection' occurs if resolving a match of bs_1 introduces results for bs_2 or strengthens an existing result.

From the effects described in Figure 4 we derive a topology as follows. The higher prioritized found effect determines the order. For example, if fixing a match for bs_1 may completely resolve a match for bs_2, bs_1 should be analyzed first, even if a lower prioritized effect takes place if bs_2 was processed before bs_1. If two bad smells have the same effect in both ways, we make a subjective decision based on mapped transformations and experience. The result is presented in Figure 5.

After processing 'Semantically Distant Instance' no additional information on further unreachable superclassifiers are needed and we suggest to clean up the taxonomy by removing all elements that are neither (transitive) subclassifiers

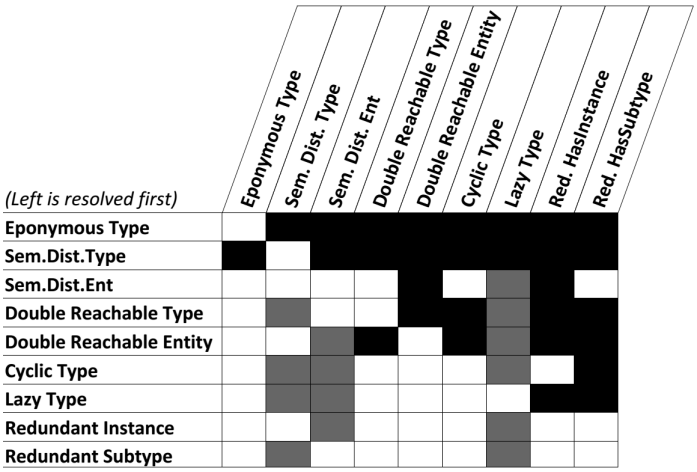

Fig. 4. Effects of analyzing one smell before another. Black represents 'Complete resolution, gray 'Ease Detection'

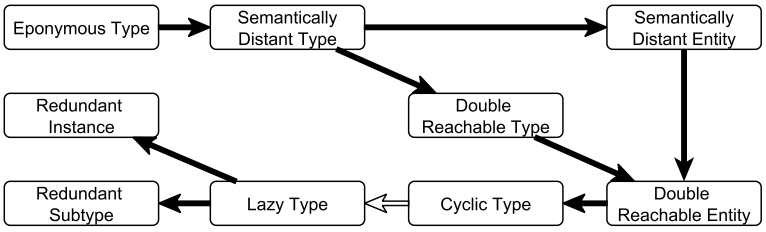

Fig. 5. The smell topology structures the smell-based analysis, where black arrows state that an order was derived from a 'Complete resolution' effect and a white arrow represents an order derived from an 'Ease detection' effect.

nor (transitive) instances of the root. Additional superclassifiers that were used to determine semantic distance and unreachable subgraphs are removed.

4 Case Study

We summarize our findings while applying the approach to the computer languages category. The data and tools of the case study as well as the actual taxonomy are available online[10]. By manually exploring the Wikipedia graph we observed a correlation between the frequency of irrelevant elements and the depth level. We chose a maximum depth of $d_{max} = 5$; deeper elements are irrelevant. An example for a category with a higher depth value is 'Silent Storm engine games' that contains articles on video games using the silent storm engine.

According to the suggested methodology of Figure 2, we present inclusion and exclusion criteria next. We first start by listing general exclusion criteria from [19] and then proceed with new and more specific ones.

[10] `http://softlang.uni-koblenz.de/wikionto/`

Alternative Classifier: We exclude categories that do not classify computer languages by any kind of software concept, purpose or language family. Thus, we are neither interested in classifying a computer language based on its influence as presented in 'Academic programming languages' nor those classifying by creation year as in 'Programming languages by creation date'.
Deviating Classifier: We exclude categories that do not represent a computer languages classifier at all. For example, any eponymous category serves as a container of related topics and is not relevant for classification. Additionally, categories classifying software that can be identified as subcategories such as 'Software by programming language' are excluded this way as well.
List Classifier: We exclude categories that serve as collections of articles covering lists of entities such as 'Lists of programming languages'.
Maintenance Pages: We exclude any category and article that only serves the overview and maintenance of Wikipedia pages, e.g., 'Wikipedia categories named after programming languages' and 'Uncategorized programming languages'.

The inclusion criteria can be summarized as follows. Any category is included that provides a classification based on a software concept or purpose of a computer language. A purpose of a language is for example expressed by 'Hardware description languages' that contains only languages meant to be used for describing a hardware's properties. Since we are interested in the similarity of computer languages, we include the categories that represent language families such as 'C programming language family'.

At the next step, during taxonomy extraction, we noticed subcategories leading far into other domains or adding too much necessary time and effort. Thus, we decided to exclude a few category names from the start that would just add excluded kinds of classifiers and many excluded kinds of subclassifiers and instances such as 'Data types' and 'Programming language topics'.

Figure 6 displays the frequencies for applied prunings from experiments conducted in December 2015. It shows actual differences in the number of matches when a topology was used ('Proc Matches') and when it is not used ('Initial Matches'). The column 'Rem. Matches' presents the number of matches for each bad smell that can be found after we applied our pruning approach. All numbers can vary depending on the executor's expertise. Even though we offered an ordered mapping in Figure 3, subjectivity and improvement with a learning process is unavoidable, but it becomes significantly less as soon as well elaborated inclusion and exclusion criteria are set up before starting.

Various decision issues may still arise in the process such as a system that might implement an own language. For example, 'Troff'[11] is a component of a document processing system. Thus, it is a subsystem and not a language, but it implements a command language which is described in the corresponding article. The same applies to many computer algebra systems[12]. Though, some languages, such as 'MatLab' can be found. The application of *Abandon Type* with a selective rescue is the most feasible.

[11] `https://en.wikipedia.org/wiki/Troff`
[12] `https://en.wikipedia.org/wiki/Category:Computer_algebra_systems`

	Lift Cycle	Abandon Type	Abandon Entity	Remove Subtype	Remove Instance	**Initial Matches**	**Proc Matches**	**Total Transf.**	**Rem Matches**
Eponymous Classifier		83				103	93	83	0
Semantically Distant Classifier		23				106	53	23	?
Semantically Distant Instance		17	426			4017	2365	443	?
Double Reachable Classifier		0		4		91	21	4	3
Double Reachable Instance			8	5	71	1965	546	84	115
Cyclic Classifier	1					2	2	1	0
Lazy Classifier		14				153	80	14	0
Redundant hasSubclassifier				18		110	18	18	0
Redundant Classifies					607	3733	607	607	0

Fig. 6. Frequencies of chosen transformation series are displayed for each smell. 'Total Transf.' sums up the frequencies for each smell. 'Initial Matches' shows the number of matches before the cleaning procedure, 'Proc Matches' displays the number of matches during the cleaning procedure and 'Rem Matches' presents the number of remaining ones per smell.

For each bad smell match, the user has to look for information at the corresponding Wikipedia pages. Some articles on language candidates in Wikipedia do not provide enough ground for a decision on how to classify them. For example, the articles 'WordBASIC' and 'Parser (CGI language)' only provide a minimal description. As a result, other sources have to be inspected as well, whose information quality may be questionable.

The classification is restricted to a tree-like taxonomy. As a result, the dimension problem is apparent at types such as 'Dependently typed languages' as matches for *Double Reachable Classifiers*, where such structure might not fit or is used in the wrong way. E.g., 'Dependently typed languages' is a subtype of 'Specification languages' and 'Programming languages' at the same time, but it has instances that should not be transitive instances of both supertypes.

5 Concluding remarks

We have described work towards a comprehensive ontology of software languages, technologies, and concepts. In this paper, we have focused on the milestone of a taxonomy for languages. Such ontological knowledge is needed, for example, for 'semantic' documentation of software technologies and systems [8,10,9].

The key challenge in deriving a computer languages taxonomy was to identify and leverage a suitable source. There exists no gold standard, no established relatively comprehensive taxonomy. We determined that Wikipedia—in comparison to DBpedia and Wikidata—is the most suitable starting point. DBpedia integrates additional types, e.g., from Yago, which may complicate pruning. (For instance, the entry on the Java programming language[13] contains an additional,

[13] `http://dbpedia.org/snorql/?describe=http%3A//dbpedia.org/resource/Java_%28programming_language%29`

irrelevant type 'communication'. Wikidata does not contain enough information. (For instance, the entry on Java[14] only lists the classifier 'object orientation'.)

Our approach for pruning Wikipedia's classification graph is semi-automated. The search for bad smells is automated, but an expert still has to confirm the issues and make decisions about the exclusion of less or more classification relationships based on different pruning options. In this manner, some degree of subjectivity cannot be avoided. The actual pruning steps are again automated by simple transformations on the evolving category graph.

In future work, we plan to research a multi-dimensional approach, subject to the integration of heterogeneous knowledge resources of language properties [27]. In this manner, we may also incorporate data from Wikipedia articles' sections [24] and integrate data from smaller, existing taxonomies such as those identified in [19]. In terms of evaluation, we hope to involve the 'Software Language Engineering' community.

References

1. Aguado de Cea, G., Gómez-Pérez, A., Montiel-Ponsoda, E., Suárez-Figueroa, M.C.: Natural Language-Based Approach for Helping in the Reuse of Ontology Design Patterns. In: Proc. EKAW 2008. LNCS, vol. 5268, pp. 32–47. Springer (2008)
2. Baumeister, J., Puppe, F., Seipel, D.: Refactoring Methods for Knowledge Bases. In: Proc. EKAW 2004. LNCS, vol. 3257, pp. 157–171. Springer (2004)
3. Baumeister, J., Seipel, D.: Smelly Owls - Design Anomalies in Ontologies. In: Proc. FLAIRS 2005. pp. 215–220. AAAI Press (2005)
4. Baumeister, J., Seipel, D.: Verification and Refactoring of Ontologies with Rules. In: Proc. EKAW 2006. LNCS, vol. 4248, pp. 82–95. Springer (2006)
5. Baumeister, J., Seipel, D.: Anomalies in ontologies with rules. J. Web Sem. 8(1), 55–68 (2010)
6. Fahad, M., Qadir, M.A.: A Framework for Ontology Evaluation. In: Supplementary Proceedings ICCS 2008. CEUR Workshop Proceedings, vol. 354, pp. 149–158. CEUR-WS.org (2008)
7. Favre, J., Gasevic, D., Lämmel, R., Winter, A.: Guest Editors' Introduction to the Special Section on Software Language Engineering. IEEE Trans. Software Eng. 35(6), 737–741 (2009)
8. Favre, J., Lämmel, R., Leinberger, M., Schmorleiz, T., Varanovich, A.: Linking Documentation and Source Code in a Software Chrestomathy. In: Proc. WCRE 2012. pp. 335–344. IEEE (2012)
9. Favre, J.M., Lämmel, R., Schmorleiz, T., Varanovich, A.: 101companies: A Community Project on Software Technologies and Software Languages. In: Proc. TOOLS 2012. LNCS, vol. 7304, pp. 58–74. Springer (2012)
10. Favre, J., Lämmel, R., Varanovich, A.: Modeling the Linguistic Architecture of Software Products. In: Proc. MODELS 2012. LNCS, vol. 7590, pp. 151–167. Springer (2012)
11. Flati, T., Vannella, D., Pasini, T., Navigli, R.: Two Is Bigger (and Better) Than One: the Wikipedia Bitaxonomy Project. In: Proc. ACL 2014, Volume 1: Long Papers. pp. 945–955. The Association for Computer Linguistics (2014)

[14] `https://www.wikidata.org/wiki/Q251`

12. Floyd, R.W.: The Paradigms of Programming. Commun. ACM 22(8), 455–460 (1979)
13. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison-Wesley (1999)
14. Ivanova, V., Lambrix, P.: A Unified Approach for Aligning Taxonomies and Debugging Taxonomies and Their Alignments. In: Proc. ESWC 2013. LNCS, vol. 7882, pp. 1–15. Springer (2013)
15. Jiang, S., Nian, J., Zhao, S., Zhang, Y.: Small Is Powerful! Towards a Refinedly Enriched Ontology by Careful Pruning and Trimming. In: Proc. ADMA 2013, Part I. LNCS, vol. 8346, pp. 300–312. Springer (2013)
16. Kim, J.W., Caralt, J.C., Hilliard, J.K.: Pruning Bio-Ontologies. In: HICSS-40 2007, Abstracts Proceedings. p. 196. IEEE Computer Society (2007)
17. Lambrix, P., Liu, Q.: Debugging the missing is-a structure within taxonomies networked by partial reference alignments. Data Knowl. Eng. 86, 179–205 (2013)
18. Lambrix, P., Liu, Q., Tan, H.: Repairing the Missing is-a Structure of Ontologies. In: Proc. ASWC 2009. LNCS, vol. 5926, pp. 76–90. Springer (2009)
19. Lämmel, R., Mosen, D., Varanovich, A.: Method and Tool Support for Classifying Software Languages with Wikipedia. In: Proc. SLE 2013. LNCS, vol. 8225, pp. 249–259. Springer (2013)
20. Lee, W., Bridewell, W., Das, A.K.: Comparison of Semantic Similarity Measures for Application Specific Ontology Pruning. In: HISB 2011. pp. 97–103. IEEE (2011)
21. Liu, H., Yang, L., Niu, Z., Ma, Z., Shao, W.: Facilitating software refactoring with appropriate resolution order of bad smells. In: Proc. ESEC / SIGSOFT FSE 2009. pp. 265–268. ACM (2009)
22. Nuzzolese, A.G., Gangemi, A., Presutti, V., Ciancarini, P.: Encyclopedic Knowledge Patterns from Wikipedia Links. In: Proc. ISWC 2011. LNCS, vol. 7031, pp. 520–536. Springer (2011)
23. Poveda-Villalón, M., Suárez-Figueroa, M.C., Gómez-Pérez, A.: Validating Ontologies with OOPS! In: Proc. EKAW 2012. LNCS, vol. 7603, pp. 267–281. Springer (2012)
24. Presutti, V., Consoli, S., Nuzzolese, A.G., Recupero, D.R., Gangemi, A., Bannour, I., Zargayouna, H.: Uncovering the Semantics of Wikipedia Pagelinks. In: Proc. EKAW 2014. LNCS, vol. 8876, pp. 413–428. Springer (2014)
25. Raad, J., Cruz, C.: A Survey on Ontology Evaluation Methods. In: Proc. KEOD 2015. pp. 179–186. SciTePress (2015)
26. Roussey, C., Zamazal, O.: Antipattern detection: how to debug an ontology without a reasoner. In: Proceedings of the Second International Workshop on Debugging Ontologies and Ontology Mappings. CEUR Workshop Proceedings, vol. 999, pp. 45–56 (2013)
27. Shilov, N.V., Akinin, A.A., Zubkov, A.V., Idrisov, R.I.: Development of the Computer Language Classification Knowledge Portal. In: PSI 2011, Revised Selected Papers. LNCS, vol. 7162, pp. 340–348. Springer (2012)
28. Stvilia, B., Twidale, M.B., Smith, L.C., Gasser, L.: Information quality work organization in wikipedia. JASIST 59(6), 983–1001 (2008)
29. van Roy, P.: Programming Paradigms for Dummies: What Every Programmer Should Know. In: New Computational Paradigms for Computer Music, IRCAM/Delatour, France. pp. 9–38 (2009)
30. Wang, H., Jiang, X., Chia, L., Tan, A.: Wikipedia2Onto. Building Concept Ontology Automatically, Experimenting with Web Image Retrieval. Informatica (Slovenia) 34(3), 297–306 (2010)