

Extracting Domain Ontologies from Domain Specific APIs

Daniel Ratiu and Martin Feilkas
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, D-85748 Garching
ratiu|feilkas@in.tum.de

Jan Jürjens
Department of Computing
The Open University
Milton Keynes, Great Britain
<http://www.jurjens.de/jan>

Abstract

Domain specific APIs offer their clients ready-to-use implementations of domain concepts. Beside being interfaces between the worlds of humans and computers, domain specific APIs contain a considerable amount of domain knowledge. Due to the big abstraction gap between the real world and today's programming languages, in addition to the knowledge about their domain, these APIs are cluttered with a considerable amount of noise in form of implementation detail. Furthermore, an API offers a particular view on its domain and different APIs regard their domains from different perspectives. In this paper we propose an approach for building domain ontologies by identifying commonalities between domain specific APIs that target the same domain. Besides our ontology extraction algorithm, we present a methodology for eliminating the noise and we sketch possible usage-scenarios of the ontologies for program analysis and understanding. We evaluate our approach through a set of case-studies on extracting domain ontologies from well-known domain specific APIs.

1 Introduction

According to [4] one of the central goals of reverse engineering is to create a representation of a system at a higher level of abstraction than the code itself. A central aspect of the reverse-engineering problem is the location of concepts in the code [3, 9]. In order to do concept location automatically we need two ingredients: an amount of relevant domain knowledge that contains concepts at the proper abstraction level and in a machine processable format, as well as the identification of how this knowledge is reflected at code level.

Most of the knowledge necessary for understanding programs is of technical nature [2, 1] (e.g. knowledge related to graphical user interfaces, networking, XML processing). The difficulty in understanding the programs resides in the

large quantity of knowledge that needs to be managed and the fact that it is non-localized in the maintained programs.

But domain knowledge in an machine processable format and at a proper abstraction level suitable for program analysis is difficult to get. One of the biggest sources of technical knowledge that is represented in a structured form are the public interfaces of domain specific libraries. Every programming language has an implementation of the core technical concepts in its standard libraries. However, a single API contains only one view on its domain and this is usually not sufficient to gain a complete model of the domain. Furthermore, APIs contain a significant amount of bias and noise in form of implementation details mixed with representations of the domain knowledge in their interfaces. In order to overcome these problems, we present a technique for extracting the domain knowledge based on the similarities of several APIs that cover the same domain. Different APIs that are developed by different programmers in different organizations and perhaps even in different languages but still target the same domain, give us a much broader perspective of this domain.

In Figure 1 (left) we illustrate intuitively this situation: the upper part of this figure represents the forward engineering process of building the APIs. Starting from the same domain knowledge, different programmers provide different implementations of domain concepts. In the lower part the approach taken in this paper to extract the domain knowledge is presented: the commonalities of more APIs are captured into a domain ontology.

Since there is a big abstraction gap between the modeled domain and the programming languages, the concepts and aspects of the domain can be reflected in the code in a multitude of ways and from a multitude of perspectives. In the upper part of Figure 1 (right) we present examples of common-sense concepts from the graphical user interfaces and the relations among them (e.g. buttons are graphical components, graphical components have position and size). In the lower part we present how is this knowledge reflected in three of the most well-known widget APIs (Java AWT,

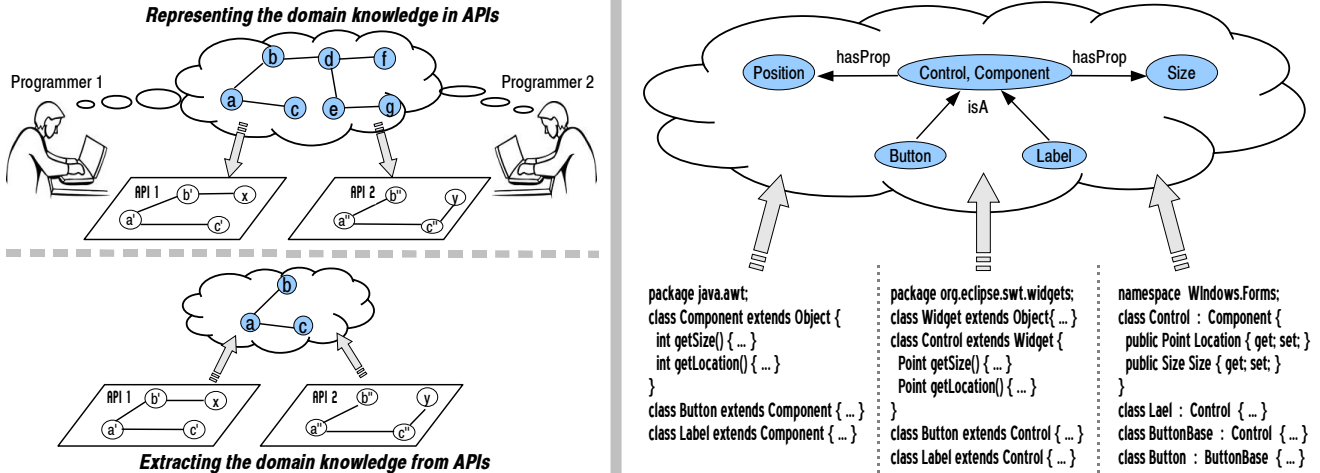


Figure 1. Domain knowledge reflected in APIs

Eclipse SWT and .NET).

The challenge of extracting the domain knowledge automatically is twofold: firstly we need to identify a way to uniformize the possibly different implementations of the same real-world situation, and secondly we need to filter out the noise introduced by particular implementation details.

In this paper we represent the extracted domain knowledge as (light-weighted) ontologies. Even if obtaining good ontologies is challenging, we envision that once they are available, they can improve the current reverse engineering practice in a number of directions:

- **Program understanding.** Mapping parts of programs to concepts from ontologies enables maintainers to regard program parts from the point of view of the concepts that they implement.
- **Assessing the quality of APIs.** Using a domain ontology we can compare different APIs that address the same domain. Thereby, we can identify situations in which an API implements domain concepts in a way that does not match to the domain knowledge represented by the ontology [12, 11].
- **Enriching program analysis with domain knowledge.** Many of the current wide-spreaded code analyses (e.g. clone detection, assessing the design quality) are at a pure syntactical level. However, the proper interpretation of their results requires semantical information about their relation of program parts to the real-world knowledge (e.g. design flaws in classes that represent the core of an application are usually more problematic than design flaws in some UI parts)

Outline In Section 2 we present our framework for representing API that take into consideration both the public program elements and their names. In Section 4.2.1 we give

a short introduction into ontologies with focus on domain specific ontologies as machine processable carriers of domain knowledge. In Section 4 we present our algorithm that uses the similarities of several APIs that address the same domain and extracts domain knowledge in form of a domain ontology. We present also a methodology for post processing the results automatically obtained by our algorithm in order to eliminate the noise. In Section 5 we present two case-studies where we performed on APIs targeting the building of graphical widgets and XML. We demonstrate the usefulness of the obtained ontologies on a concepts location example. In Section 6 we present the related approaches and after this we end our paper by presenting our conclusions and future work plans.

2 Formalizing APIs

In order to identify domain concepts based on similarities of several APIs, we need to represent the APIs in a rigorous manner. Our formalization uses a graph-based representation of the program elements from the public interface of the APIs (Section 2.1) and models explicitly the names of the program elements (Section 2.2).

2.1 Representing APIs as Graphs

We describe the program layer as a labeled directed graph. The nodes of the graph are the program elements accessible to the users of the API and its edges are typed relations defined in the program among these program elements. Formally, the program layer Π of an API is:

$$\Pi = \langle P, \Sigma, e \rangle, \quad e : P \times P \rightarrow \Sigma \cup \{\epsilon\}$$

The set of the nodes (P) represents the program entities accessible through the public interface of the library. The

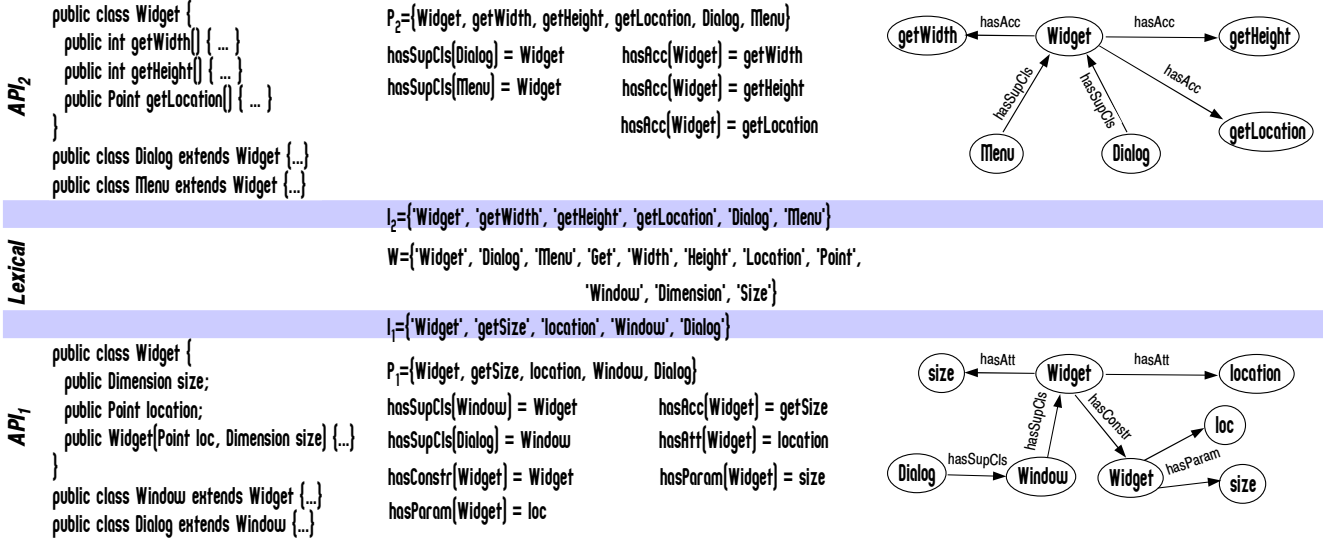


Figure 2. API Layers

kinds of relations between the public program elements is given by a set of labels Σ . Given a pair of nodes, the function e returns the type of the edge between them or ϵ if there is no edge.

The exact set of relation types varies from paradigm to paradigm and even from language to language inside the same paradigm (e.g. Smalltalk has no public attributes). We consider in this paper only the case of Java-like languages and thereby we use the following relation types: $hasSupCls$, $hasType$, $hasAcc$, $hasCtr$, $hasMeth$, $hasAtt$ and $hasPar$. The semantic of the labels is defined as follows: $hasSupCls$ represents the relation between a class and its super classes; $hasType$ is a relation between an attribute and its type; $hasAcc$ is a relation between a class and its accessors; $hasCtr$ is a relation between a class and its constructors; $hasMeth$ is a relation between a class and one of its methods that are neither constructors nor accessors; $hasAtt$ is a relation between a class and its attributes; $hasPar$ is a relation between a method and its parameters.

Notation: In the following of this paper we will describe the relations between the graph nodes through sets of functions named according to the relations' labels. Thus, iff between two program elements p_1 and p_2 is an edge which contains the label $hasType$ then we use the following notations interchangeably: $e(p_1, p_2) = hasType$ and $e(p_2, p_1) = hasType^{-1}$ and $p_2 \in hasType(p_1)$. A path between two nodes in the program graph is given by a sequence of labels between these nodes – e.g. $\langle hasType, hasSupCls \rangle$

Example: In the lower and upper parts of Figure 2 we present examples of two APIs: on the left side is the

source code, in the middle is their instantiation according to our framework and on the right these APIs are represented as graphs. For example, the fact that the class `Widget` has attribute `size` is represented through the relation: $e(Widget, size) = hasAtt$. In API_1 , between the nodes `Widget` and `Dialog` is the following path: $\langle hasSupCls^{-1}, hasSupCls^{-1} \rangle$. If we consider the inverse sense, namely between the node `Dialog` and `Widget` then the path is $\langle hasSupCls, hasSupCls \rangle$. Similarly, between the nodes `Widget` and `loc` the path is: $\langle hasCtr, hasPar \rangle$.

2.2 Lexical layer

In a similar manner to the communication among humans, which is many times realized through words that serve as carriers for the semantic information, we consider the program element names (identifiers) to carry the information about the domain. Formally the lexical layer Λ is:

$$\Lambda = \langle I, W \rangle$$

$$P_2 I : P \rightarrow I, I_2 W : I \rightarrow \mathcal{P}(W)$$

The library's vocabulary (I) is represented by the set of program elements names that are accessible through the public interface of the API. The lexical layer is centered around a set of lexically normalized words (W) that are obtained by the reunion of the words of identifiers. We consider that words carry the basic information and they represent the fundamental lexicalized concepts of the domain. The lexical layer represents the "skin" of the library and is used to communicate among the library developer and its users. The functions program element-to-identifier ($P_2 I$) map the program elements to their names. The functions identifier-

to-words (I_2W) is responsible for obtaining the set of normalized words contained in the identifiers' names. The function I_2W transforms an identifier into a set of normalized words. The splitting of an identifier into words is done by using a set of heuristics (e.g. CamelCase, special delimiters like underscore).

Example: In the middle part of Figure 2 we present an example of the lexical layer corresponding to the two APIs. It is centered around a set of words that is obtained through the splitting of identifiers of the reunion of identifiers from API_1 (I_1) and API_2 (I_2). Examples of the function identifiers-to-words are: $I_2W('getHeight') = \{'Get', 'Height'\}$, $I_2W('drawAndMove') = \{'Draw', 'And', 'Move'\}$, $I_2W('XMLNode') = \{'Xml', 'Node'\}$.

3 Knowledge sharing through ontologies

In this section we give an overview of ontologies in the context of our work (Section 3.1) and present typical ways in which ontological relations are represented in APIs (Section 3.2).

3.1 Ontologies in a nutshell

To support sharing and reuse of knowledge of a particular domain one needs to explicitly represent it in a formal manner. The first step in formally representing a body of knowledge is to decide on a *conceptualization* of the domain. A conceptualization is an abstract, simplified view of a domain which is to be described for a particular purpose. It contains a set of objects together with their properties and relations [6]. An ontology is defined to be an *explicit specification of a conceptualization* [7] and is used for sharing the knowledge about a domain by making explicit the concepts and relations within it. In the present work we use an informal meaning of the term “ontology” - which we regard to comprise only concepts and relations between them, among which the most important is the “isA” hierarchical relation between the superordinate and its subordinates (e.g. Component – isA – Window). Apart from isA we consider in our ontology other three relations: *hasProperty* between an entity and its properties (e.g. Window – hasProperty – Size); *isDoer* between an object and the action that it performs (e.g. Window – isDoer – paint) and *actsOn* between an action and the entities on which it is performed (e.g. Resize – actsOn – Window). We assume that these relations are expressive enough to cover a relative large part of the common real-world situations.

In order to represent an ontology we use a graph language similar to the RDF graphs [8]. Entities within the

ontology are the nodes of the graph and relations between them are represented as labeled arcs.

Current off-the-self ontologies cover only restricted parts of some domains. Usually ontologies are built for a particular purpose and represent the domain concepts from a certain perspective that fits at best for achieving their purpose. Furthermore, to the best of our knowledge, with very few exceptions, there are no ontologies that address the technical domains that are usually implemented in libraries.

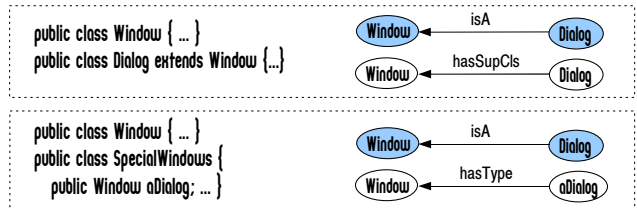
Thus, even if the number of the ontologies is growing rapidly, most of the time there aren't ontologies available that can be used to analyze a program. In [11] we proposed a method for manually building ontologies that are suitable for analyzing APIs. In many cases bigger ontologies (with more than 100 concepts) are needed, but these are difficult to build manually.

Our assumption is that extracting the domain knowledge from APIs is much more efficient than building the ontology beforehand.

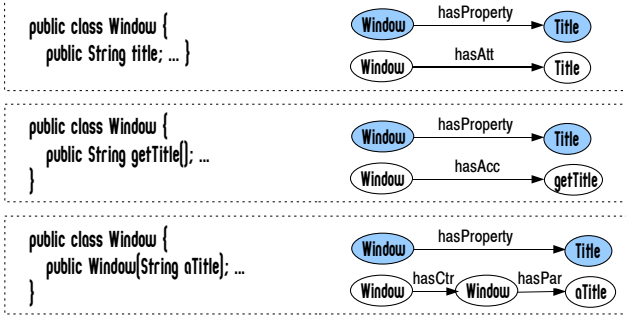
3.2 Reflecting abstract relations in APIs

Our aim is to compare many APIs in order to extract the domain knowledge in form of a domain ontology (concepts and relations among them). In order to identify the similarities between the APIs, we start by studying the typical ways of how the ontological relations that we aim to recover are reflected at the APIs level.

Reflecting the “isA” relation in APIs. The isA relation between an entity and its superordinate is reflected usually at the API level through the type-system generated relations: either through the sub-classing relation or through a relation between a variable and its type. Below are several examples that reflect how the “Dialog isA Window” relation from an imaginary ontology about graphical widgets can be implemented in the code.



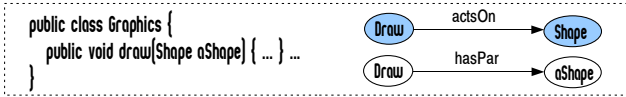
Reflecting the “hasProperty” relation in APIs. The hasProperty relation between an entity and its properties is reflected usually at the API level through the attributes of a class, through accessor methods or through the parameters of the constructors.



Reflecting the “isDoer” relation in APIs. The isDoer relation between an entity and the action it performs is reflected in APIs through hasMethod relation between a class representing the entity and a method representing the action.



Reflecting the “actsOn” relation in APIs. The actsOn relation between an action and the affected entity is reflected in APIs through hasParameter relation between a method representing the action and its parameters representing the entity.



4 Extracting domain knowledge from APIs

In this section we present our algorithm and method for extracting the domain knowledge from more APIs. The graph-matching algorithm (Section 4.3) uses the similarity of program element names (Section 4.1) and the similarity of paths (Section 4.2). In Section 4.4 we present the sequence of steps that needs to be performed to extract the ontology.

4.1 Naming clues

As already mentioned one of the most important sources of information for identifying the concepts in the code is given by the identifiers’ names. We use in our case the identifiers of program elements from the analyzed APIs as hints for matching different program elements that refer to the same concept. In a similar manner with the real world language, we consider that the individual words represent the basic semantic carriers. Identifiers composed of more

words can refer either to one complex concept or to several concepts.

Below we define a similarity between two program elements based on their name. Given two program elements p_1, p_2 , the predicate $simNme(p_1, p_2)$ returns true if the names of these elements are similar. We use the following definition of $simNme$:

$$simNme(p_1, p_2) = true \Leftrightarrow \frac{\|W_1 \cap W_2\|}{\|W_1 \cup W_2\|} \geq 0.5,$$

$$where W_1 = I_2W(P_2I(p_1)), W_2 = I_2W(P_2I(p_2))$$

Intuitively, two program elements have similar names iff more than half of their words are the same.

Example: $simNme(BaseButton, Button) = true$ since from the two words that these identifiers contain (‘Base’ and ‘Button’) one word (‘Button’) appears in both identifiers. $simNme(ColoredButton, ColoredLabel) = false$ since only one of the three words contained by these identifiers is shared between them.

4.2 Mapping paths

The usage of names as clues for the identification of commonalities between more APIs has two disadvantages: Firstly based only on names we can identify the vocabulary of the domain and not a domain ontology; Secondly the names are themselves ambiguous (e.g. synonymy and polysemy) and the matching of two names does not imply the matching of two concepts. In order to overcome these problems we use a graph matching algorithm that extracts concepts based on the similarities of the API graphs. To apply our algorithm, we need to define similarity between program relations. Starting from our observations in Section ??, we define the paths equivalence relation \sim to be:

$$\sim \subset \Sigma^* \times \Sigma^*$$

This relation is defined for the sequences of edges that reflect a similar implementation situation. The relation \sim is transitive and commutative. Starting from the analysis of the implementation of ontological relations in the API we will use the following instantiation of the \sim relation in our algorithm:

- T1. $\langle hasSupCls \rangle \sim \langle attHasType \rangle$
- T2. $\langle hasAtt \rangle \sim \langle hasAcc \rangle \sim \langle hasCtr hasPar \rangle$
- T3. $\langle hasMeth \rangle \sim \langle hasMeth \rangle$
- T4. $\langle hasPar \rangle \sim \langle hasPar \rangle$

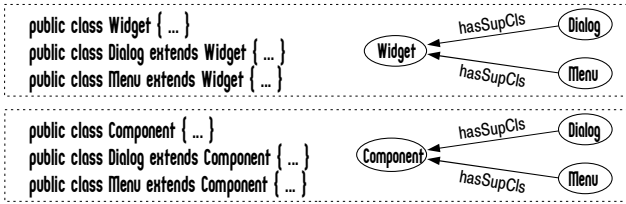
Given two independent implementations of the same domain knowledge, there is a certain amount of inherent variations represented by implementation decisions. Basic variations were discussed in Section 3.2 (e.g. properties can be

reflected as attributes, accessors or constructor parameters; is-a relation can be represented either through sub-classing or through the relation between a variable and its type) and are captured in the equation T2. Below we discuss several other cases of heterogeneity.

4.2.1 Variation points

Different implementations of the same situation can exhibit heterogeneities that reflects the different perspectives under which the domain is represented in the API. In the following we concentrate on two heterogeneities:

Terminological mismatches. Many times the same concept is implemented in different APIs under different names. In order to overcome this problem we use the similarity in the representation of the neighbors of these concepts. For example, if a concept c is implemented in two APIs through the classes p_1 and p_2 with different names ($simNme(p_1, p_2) = false$) and its siblings are implemented as sub-classes of these classes then we can identify the concept c by using the structural similarities between these two implementations. In our example below we identified that *widget* and *component* are used to express the same concept (i.e. GRAPHICAL COMPONENT).

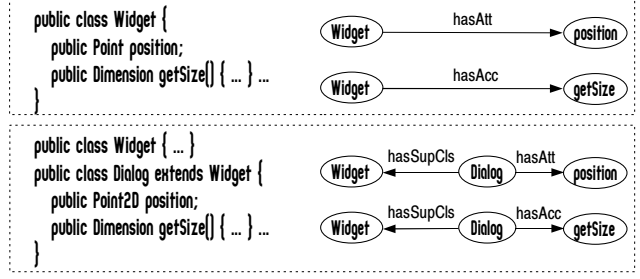


$$T5. \langle hasSupCls \ hasSupCls^{-1} \rangle \sim \langle hasSupCls \ hasSupCls^{-1} \rangle$$

Structural mismatches. In Section we presented the cases where the ontological relations are reflected directly the in APIs. In reality, however, it is often the case that the ontological relations are reflected in a degenerate manner. For example, it is often the case that in one API the properties of a concept are implemented only in one of the sub-classes that refer to the concept. Below we present two code fragments from two APIs where the relations “Widget – hasProperty – Size” and “Widget – hasProperty – Position” are implemented directly (upper part) and are degenerate (lower part). From this observation we deduce the mappings (T6 – T7).

$$T6. \langle hasSupCls^{-1} \ hasAtt \rangle \sim \langle hasAtt \rangle$$

$$T7. \langle hasSupCls^{-1} \ hasAcc \rangle \sim \langle hasAcc \rangle$$



4.3 Ontology extraction algorithm

Having abstracted the APIs as graphs, the identification of similarities between many APIs is based on a graph matching algorithm: it matches the nodes of the two graphs based on the similarity of their names. After this it looks for a compatible path in the graphs between every pair of nodes. Whenever a match between nodes and relations is found, we identify a pair of concepts and a relation between them. The relation between these concepts is one of the ontological relations from Section 4.2.1. Below we present our algorithm in pseudo-code and in Figure 3 we present an example of how this algorithm works. We consider that the two APIs that are compared are $\Pi = (P, \Sigma, e)$ and $\Pi' = (P', \Sigma', e')$.

1. **for-each** $p_1 \in P$ **do**
2. $reflection(p_1) = \{p'_1 \in P' \mid simNme(p'_1, p_1)\}$
3. **for-each** $\langle \sigma_i \dots \sigma_j \rangle \in \Sigma^*, \langle \sigma'_k \dots \sigma'_l \rangle \in \Sigma'^* . \langle \sigma_i \dots \sigma_j \rangle \sim \langle \sigma'_k \dots \sigma'_l \rangle$
4. $neighbors(p_1, \langle \sigma_i \dots \sigma_j \rangle) = \{p_2 \in P \mid p_2 \in \langle \sigma_i \dots \sigma_j \rangle(p_1)\}$
5. **for-each** $p_2 \in neighbors(p_1, \langle \sigma_i \dots \sigma_j \rangle)$
6. $reflection(p_2) = \{p'_2 \in P' \mid simNme(p'_2, p_2)\}$
7. **for-each** $p'_1 \in reflection(p_1), p'_2 \in reflection(p_2)$
8. **check-if** $p'_2 \in \langle \sigma'_k \dots \sigma'_l \rangle(p'_1)$
8. **if-yes** $saveRelation(comNme(p_1, p'_1),$
9. $comNme(p_2, p'_2),$
10. $ontRel(\langle \sigma_i \dots \sigma_j \rangle))$

The function *comNme* takes two program elements as parameters and returns the intersection of the words of their identifiers. We consider that the words obtained through this intersection represent the name of the identified concept. Below is the formal definition of *comNme*:

$$comNme : P \times P' \rightarrow W^*$$

$$comNme(p, p') = I_2W(P_2I(p)) \cap I_2W(P_2I(p'))$$

The function *ontRel* transforms a path in the program graph into its corresponding ontological relation – e.g. $ontRel(\langle hasSupCls^{-1} \ hasAcc \rangle) = hasProperty$

4.4 Knowledge extraction methodology

We proposed an algorithm based on graph matching that is able to automatically find the similarities between different APIs as well as collect and interpret these similarities into a domain ontology. In order to obtain an ontology we need to perform the following sequence of steps:

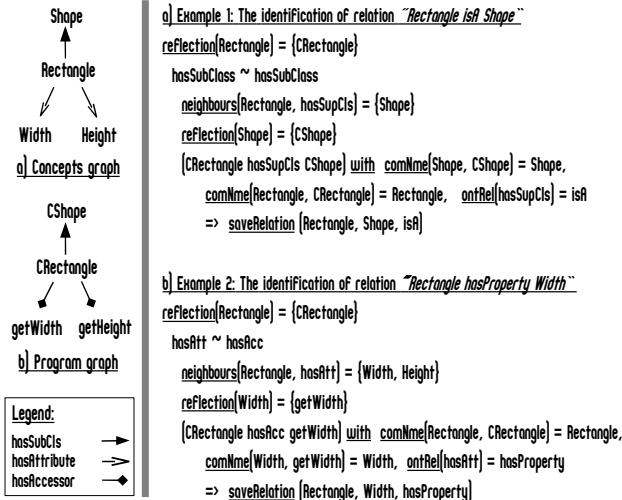


Figure 3. Concept identification examples

Step 1: Establish the scope of analysis. The very first step is to exactly establish the scope of the analysis. Here one should answer to questions like: what is the domain we target? What should the extracted ontology look like?

Step 2: Select the set of APIs. After establishing the analysis scope, we need to select the set of APIs that will be used for extracting the domain knowledge. Ideally, in order to avoid the implementation noise, these APIs should be implemented in different languages.

Step 3: Run the concepts identification algorithm. The running of the algorithm is fully automatic. The algorithm's output is a set of concept candidates and the corresponding relations among them. The algorithm ranks the concepts and the relations according to their frequency.

Step 4: Eliminate the noise. There is a considerable number of relations that are noise from the point of view of the domain knowledge. We eliminate these relations in two manners: Firstly, we use an heuristic that the most important concepts and relations have a higher matching frequency. Thus, we remove the concept – relation – concept that contain concepts with low frequency. Secondly, we manually inspect the remaining triples and remove the ones that do not make sense from the point of view of the domain.

5 Case Study

In our experiments we aim to answer the questions related to the feasibility of our approach (Q1-2) and to briefly demonstrate an example of using the extracted ontology for concept location (Q3):

Q1) Are the overlappings between different domain specific APIs that address the same domain big enough for extracting domain ontologies? This question addresses the

recall of our ontology mining algorithm (Section 5.1).

Q2) What is the amount of noise in the extracted ontology? What is the manual effort for eliminating the noise? This question addresses the precision of the extraction algorithm and the feasibility of eliminating the noise (Section 5.2).

Q3) How appropriate is the extracted ontology for locating concepts in code? This question addresses the relevance of the extracted ontology for a central reverse engineering activity, namely concept location (Section 5.3).

Experimental setup We performed experiments on two sets of widespread APIs: The first set is represented by the APIs that implement the functionality for processing XML documents. In this case we chose the following APIs: `org.w3c.dom` is the implementation of the W3C DOM (Document Object Model) available in the Java standard library; `dom4j`¹ open source library for working with XML; `jdom`² library for accessing, manipulating, and outputting XML data; `xom`³ tree-based API for processing XML and the XML processing API from the .NET platform. The second set of APIs implement the functionality related to GUI toolkits, graphical widgets and basic drawing: the `AWT` and `SWING` APIs from the Java standard library, the Eclipse Standard Widget Toolkit (`SWT`) and the .NET API from the namespace `Windows.Forms`. In order to answer the question Q3 we used the `JHotDraw` framework (version 7.0.9). In Figure 4 we present an overview over the size of the analyzed APIs.

	awt	swing	swt	.net	w3c.dom	dom4j	jdom	com	.net	JHD7.0.9
Cls	354	719	245	772	140	158	65	51	255	371
r	3301	4380	2008	7038	1364	947	449	230	1646	1610
W	1340	1627	856	1362	479	344	235	156	559	718

Figure 4. APIs Overview

5.1 Assessing API Overlappings

Vocabulary Overlapping. The most naive measurement of overlapping represents the vocabulary (terminological) level. In Figure 5 we present the overlapping of the vocabulary of different APIs pairs. For example, in the case of XML libraries, from the reunion of words of `w3c.dom` and `dom4j`, approximately 27% (about 180) are common words; between `jdom` and `dom4j` approximately 40% of their total number of words are shared.

In the case of graphical widgets libraries, we can notice a similar phenomenon: there is a quite wide range of the overlapping ratio from approximately 20% up to almost 40%.

¹www.dom4j.org

²www.jdom.org

³www.xom.nu/

For example, the AWT implements rather low-level drawing concepts and many concepts related to internationalization of GUIs. These concepts do not appear in the other APIs. From these tables we draw the conclusion that the similarity at the vocabulary between different APIs that address the same domain is between 20% and 40%.

Since we consider that the words are carriers of semantic information, the vocabulary overlapping gives hints of the conceptual similarity of these APIs. We considered these results as promising and performed our ontology extraction methodology.

	w3c.dom	dom4j	jdom	nom	.NET		awt	swing	swt	.NET
w3c.dom	-	0.27	0.21	0.18	0.28	awt	-	0.32	0.19	0.31
dom4j	-	-	0.40	0.28	0.35	swing	-	-	0.20	0.38
jdom	-	-	-	0.35	0.28	swt	-	-	-	0.35
nom	-	-	-	-	0.22					

Figure 5. APIs Vocabulary Overlapping

The automatically extracted ontology. In Figure 6 we present the number of extracted concepts and relations. In the case of XML, we obtained 371 concepts and 1145 relations among these concepts. In the case of graphical widgets we obtained 926 concepts and 2918 relations. In Figure 6 we see that the highest number of relations are represented by the relations between concepts and their properties.

	#Concepts	#Relations	#isA	#hasProperty	#isDoer	#actsOn
XML	371	1145	42	477	355	271
GUI	926	2918	203	1625	796	294

Figure 6. Automatically extracted ontology

Terminology mismatches. Many times, the same concept is referred in different APIs under different names. For example, in the GUI APIs the names component, control and widgets are used to refer to the same kind of concept: an abstract element of the hierarchy of graphical components. In Figure 7 we present an example of how we could identify the synonymy between the control and component names.

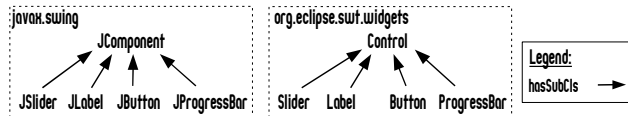


Figure 7. Terminology mismatches examples

Identifying the core concepts and relations. In order to rank the importance of the automatically extracted concepts, we counted how many times they participated in a match. Analogously we did for the relations between these

concepts: we counted how many times a relation was identified.

Below are examples of the concepts and relations from the XML ontology. We present the most frequent 20 concepts, some of their frequencies and a set of concepts that appeared only once (e.g. the concept 'text' was identified in 521 relations).

element (2693), node, name, attribute, document (1268), value, namespace, child, text (521), create, processing instruction, type, datum, xml, uri (318), remove, add, write, x path, local name (242), [...], entity reference node, node name, html, return, omit, create cdata, enumeration, processing instruction node, ha attribute n, any attribute

Below we present the most frequent 10 “concept – relation – concept” triples and their frequencies.

element–actsOn (299) –name	element–hasProp (136) –attribute
element–hasProp (288) –name	element–hasProp (117) –type
attribute–actsOn (191) –name	reader–isDoer (102) –read
element–isDoer (171) –attribute	attribute–actsOn (107) –value
document–isDoer (151) –create	attribute–hasProp (101) –value

Observations: We remark that the concepts with a higher frequency have a very high relevance for the XML domain. At the same time, most of the least frequent concepts have a very low relevance or are compound concepts (i.e. node name, create cdata) that can be expressed in terms of frequent concepts and relations between them (i.e. node hasProperty name; create actsOn cdata). Similarly, the most frequent triples concept–relation–concept represent relations that are in typical for the XML domain.

5.2 Applying the extraction methodology

Starting from these observations, we apply our methodology for eliminating the noise and identifying the central concepts: Firstly we selected most frequent 50% of the concepts; Secondly we eliminated the relations that contained the least frequent 50% of the concepts. Finally we eliminated the 50% least frequent of the remaining relations. We are aware that through these heuristics we loose also useful information. However, we concentrate on the elimination of noise.

Following the application of these heuristics, we obtained in the case of XML ca. 180 concepts and 456 relations. In the case of the GUI ontology we obtained about 450 concepts and 941 relations. The next step is to manually eliminate the noise by inspecting the remaining triples. After the manual inspection we obtained in the case of XML an ontology with 122 concepts and 235 relations. In the case of GUI we got 265 concepts and 580 relations. Below

we exemplify the content of our ontology in the case of the concepts `BUTTON` and `LIST`.

button-hasProp-size	button-hasProp-border
button-hasProp-text	button-isA-Component, Control
button-hasProp-alignment	button-isDoer-add listener
button-hasProp-label	button-isDoer-remove listener
button-hasProp-enable	button-isDoer-click
list-hasProp-size	list-hasProp-count
list-hasProp-selection index	list-isA-Component, Control
list-hasProp-item	list-isDoer-add item
list-hasProp-bound	list-isDoer-add selection
list-hasProp-minimum size	list-isDoer-remove selection
list-hasProp-name	list-isDoer-paint

Effort estimation In Table 1 we present the duration measured in hours of each of these steps. These results represent only the experiments and do not take into account the programming effort. We spent most of the time in selecting the set of APIs and in preparing them for analysis (e.g. we removed the tests). Once the heuristics for eliminating the noise are applied we performed the manual inspection with minimal effort.

Operation	XML	GUI	Auto
Selection of APIs	3	2	No
Preparation of APIs	3	3	No
Algorithm running	1	3	Yes
Noise elimination heuristics	0.5	0.5	Yes
Manual noise elimination	0.5	1	No

Table 1. Estimation of the effort

5.3 Using the ontology for concept location

Having an ontology that represents the knowledge contained in a certain type of libraries in a machine processable format is a gain per se. Obtaining such ontologies for all types of libraries would cover a wide area of the programming knowledge. In the following we give only hints of how such ontologies can be used in reverse engineering. We choose a central reverse engineering problem – namely concept location. We will use a method for locating concepts in code that is based on mapping program entities to ontologies[11, 10]. Whenever a mapping is found, we identify a concept in the code.

In order to perform our experiments we choose the version 7.0.9 of the drawing framework JHotDraw. As knowledge bases we used both the GUI and the XML ontologies. Our concept location algorithm identified concepts from both of these ontologies – JHotDraw uses both the AWT, SWING and the `w3c.dom` APIs. Our algorithm identified

179 concepts in JHotDraw. These concepts were assigned to 1370 program elements.

By inspecting the program elements assigned to XML concepts, we discovered the fact that JHotDraw contains classes that use the `nanoxml`⁴ in addition to the `w3c.dom` library. This represents a sanity check for our approach as we validate that the XML concepts contained in our ontology are general enough and do not depend on a particular XML API. In Figure 8 we present an example of how was the `ELEMENT` concept identified.

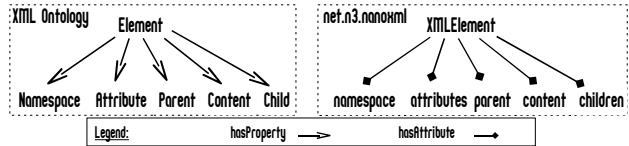


Figure 8. Concept identification example

6 Related Work

Knowledge for program understanding. The central role of knowledge management in the process of maintenance in general and program understanding in particular is widely acknowledged in the literature. In [1] the software maintenance is seen as a knowledge management issue. Among the several dimensions of knowledge (e.g. business knowledge, computer science knowledge), programmers most often make use of technical knowledge during maintenance [2].

[3, 9] presents the role of concepts in program comprehension. These concepts can be either domain concepts or technical oriented concepts. In order to automatize the concepts-centered program understanding, the tools have to be provided with a considerable amount of knowledge that is relevant for understanding a program.

One of the modalities to share and formalize the concepts in practice is through ontologies. In this paper we propose a method for extracting the knowledge from APIs and for expressing it in a formalized manner through domain ontologies. The obtained ontologies can be used as input for other reverse engineering and program analyses activities.

Knowledge representation in programs. This paper is in continuation to our previous work on knowledge representation in programs. In [10, 12, 11] we presented different problems related to the reflection, diffusion and distortion of domain knowledge programs with focus on domain specific APIs. One of the preconditions for the automatic detection of API problems is the availability of domain ontologies.

⁴<http://nanoxml.cyberelf.be/>

Building large domain ontologies that contain hundreds of concepts and relations between them is challenging.

Ontologies in software maintenance. The LASSIE system [5] represents one of the pioneering works in using ontologies in software maintenance. It uses a knowledge base system for intelligently indexing reusable components. The approach is based on mapping between a domain ontology to the code model. Although the code ontology is populated automatically, the domain ontology and its relation to the code model must be maintained manually. Such a system proved to support comprehension tasks but the overhead of manually synchronizing the models reduced the overall benefit.

[14] presents an approach for representing both the source code and the documentation as ontologies and thereby it enables the usage of semantic web technologies in the software maintenance. The semantic of the domain is captured in this case only through the analysis of the program's documentation.

The ontologies that we extract from APIs can be used as complementary sources of knowledge that addresses technical domains typically implemented in APIs.

Extracting ontologies [13] presents a method for extracting an ontology that corresponds to an API by analyzing the javadoc comments. The motivation for this work is the observation that web services reflect the functionality of their underlying implementation. The goal of this paper is to provide a description of web-services.

We advance in the direction of extracting ontologies from programs along two directions: Firstly, we capture the domain knowledge by analyzing multiple APIs and not to provide a representation of a program as an ontology as in [15]. Secondly, we extract ontologies by analyzing the APIs and not by performing natural language processing.

7 Conclusions and Future Work

In this paper we presented a method for extracting domain knowledge by analyzing multiple APIs that address the same domain. Our preliminary experiments suggest that it is feasible to extract a large quantity of information by investing a relative small manual effort. By extracting knowledge from several APIs and by sharing it in a machine processable format, we aim to increase the abstraction level at which the reverse engineering is done.

We are aware that even if the current results are promising, this work represents only the first steps both in extracting domain knowledge from APIs and in using further this knowledge in the maintenance, reverse engineering or program understanding. We can divide the future

work into two directions: Firstly, we plan to qualitatively improve our method in order to extract more expressive ontologies, reduce the noise and increase the number of concepts. Secondly, we plan to advance into a quantitative direction by extracting more ontologies that cover technical domains that are implemented usually by APIs (e.g. networking, data bases, security). By doing so we aim to build a knowledge base that contains knowledge relevant for reverse engineering in a machine processable format.

References

- [1] N. Anquetil, K. M. de Oliveira, K. D. de Sousa, and M. G. B. Dias. Software maintenance seen as a knowledge management issue. *J. of Inf. & Softw. Tech.*, 2007.
- [2] N. Anquetil, K. M. de Oliveira, M. G. B. Dias, M. Ramal, and R. de Moura Meneses. Knowledge for software maintenance. In *SEKE*, pages 61–68, 2003.
- [3] T. J. Biggerstaff, B. G. Mitbender, and D. Webster. The concept assignment problem in program understanding. In *ICSE '93*. IEEE CS, 1993.
- [4] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.
- [5] P. Devanbu, R. Brachman, and P. G. Selfridge. Lassie: a knowledge-based software information system. *Commun. ACM*, 34(5):34–49, 1991.
- [6] M. R. Genesereth and N. J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.
- [7] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum.-Comput. Stud.*, 43(5-6):907–928, 1995.
- [8] P. E. Hayes. Rdf semantics. Technical report, W3C Recommendation, 2004.
- [9] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *IWPC '02*. IEEE CS Press, 2002.
- [10] D. Ratiu and F. Deissenboeck. Programs are knowledge bases. In *ICPC '06*. IEEE CS, 2006.
- [11] D. Ratiu and F. Deissenboeck. From reality to programs and (not quite) back again. In *ICPC*. IEEE CS Press, 2007.
- [12] D. Ratiu and J. Juerjens. The reality of libraries. In *CSMR '07*. IEEE CS, 2007.
- [13] M. Sabou. Extracting ontologies from software documentation: a semi-automatic method and its evaluation. In *ECAI-OLP*, 2004.
- [14] R. Witte, Y. Zhang, and J. Rilling. Empowering software maintainers with semantic web technologies. In *ESWC*, pages 37–52, 2007.
- [15] H. Yang, Z. Cui, and P. O'Brien. Extracting ontologies from legacy systems for understanding and re-engineering. In *COMPSAC '99*. IEEE Comp. Soc., 1999.