Towards A Repository of Common Programming Technologies Knowledge

Daniel Ratiu[1] and Martin Feilkas[1] and Florian Deissenboeck[1] and Jan Juerjens[2] and Radu Marinescu[3]
[1] Technische Universität München, {ratiu,feilkas,deissenb}@in.tum.de
[2] The Open University, http://www.jurjens.de/jan
[3] Politehnica University Timisoara, radum@cs.upt.ro

Abstract

The everyday programming and maintenance activities make use of knowledge about programming-related technologies such as graphical interfaces (GUIs) or XML. Having this knowledge in a machine processable form supports the automation of typical maintenance activities such as concept location and raise the abstraction level at which the current code analyses are performed. In this paper we promote our current project of building a repository of ontologies[1] that contain knowledge about programming technologies. We propose a method for extracting these ontologies by analyzing the commonalities of different APIs that address the same domain. We discuss the motivation for such a repository in form of possible usage directions and present our experience with building and using ontologies that share technical knowledge about GUIs and XML. Based on our experience we discuss the challenges of building and evolving the repository and discuss how can the semantic web technologies contribute to this endeavor.

1 Introduction

Software maintenance is a knowledge intensive activity. Evolving, maintaining and understanding programs require that programmers possess knowledge about their application domain (e.g. banking), programming technologies (e.g. graphical widgets) and the relation between the domain concepts and the technologies used to implement them (e.g. adding a new bank client is done in the 'NewClient' dialog) [2, 1, 5, 17]. The current state of the art automatic reverse engineering analyses do not make use of such knowledge and for example make no difference between analyzing a program sub-system that models the core of the application domain (e.g. banking) and another sub-system that represents the persistency layer of the application. The reason for this is that no knowledge base is available that is suitable for automatic code analysis (i.e. contains concepts at the abstraction level of programs and in a machine processable form). In this paper we promote our project of building a repository of (light-weighted) ontologies that contain technical knowledge about programming technologies. Our repository contains common sense, basic knowledge that is well known to any programmer and that is at a rather superficial level of formalization. For example, in the case of technologies related to graphical widgets (GUIs), such knowledge is that buttons are graphical components, have labels, layout information, can be displayed, can contain other graphical components, can update their views, or that there are other kinds of input-components such as checkboxes and radio buttons. In Figure 1 we present an example about a fragment of a light-weighted ontology that contains concepts about the GUI domain. The concepts and relations between them are in form of triples subject – verb – object.

Button	isA	Component		Button	hasProperty	Text		CheckBox	isA	Button	
Button	isA	Container		Button	hasProperty	Prefered Size		RadioButton	isA	Button	
Button	hasProperty	Alignment		Button	hasProperty	Min Size		ToggleButton	isA	Button	
Button	hasProperty	Background		Button	hasProperty	Style		Show	actsOn	Button	
Button	hasProperty	Enabled		Button	hasProperty	Label		Button	isDoer	Update	

Figure 1. Common knowledge about GUIs

In [14] we present an approach for (semi-)automatically extracting ontologies by analyzing the commonalities of the APIs that address the same domain. Using this approach makes it feasible to extract ontologies that offer a good domain coverage and that can be used to build the programming technologies repository. Once the first version of such an ontology is extracted, it needs to be validated, completed and evolved. Even if this requires a lot of effort, especially for domains that contain hundreds of concepts (e.g. graphical widgets), once such an ontology is available, it contains shared knowledge in a machine processable form that can be used "off-the-shelf" in other analyses. *Our endeavor is driven by the belief that once a repository of programming technologies ontologies is available it will be of interest for a large program analysis community and will enhance the automation of many software engineering activities.*

Outline. In Section 2 we present a set of possible applications of the ontologies repository. In Section 3 we briefly introduce our view over ontologies as means for describing

[1] www4.in.tum.de/~ratiu/knowledge_repository.html

and sharing domain knowledge. In Section 4 we present a method for extracting ontologies about programming technologies by analyzing multiple APIs that address the same domain. In Section 5 we present our experience with building and using two ontologies (one about GUI and the other about XML) for tackling some of the scenarios presented in Section 2. In Section 6 we discuss several open problems and how can semantic web technologies contribute to our project. In Section 7 we present the related work and after this we conclude the paper.

2 Knowledge Repository Usage Scenarios

Even if building a repository of ontologies that address technical domains is challenging, we strongly believe that building it is not an end purpose. We envision that once such a repository is available, it will improve the current reverse engineering practice in a number of directions as listed below. Until now we have different levels of experience with these directions. The list below is ordered according to our experience: the top-most items (1 - 3) are the mostly inspected by us and the bottom-most items (4 - 6) are only speculations based on our general programming experience.

1. Concept location. Concept assignment and concept location are among the most important activities in everyday software maintenance tasks [4, 12]. In our previous work [15] we developed a technique for (semi-)automatic location of domain concepts in the code by mapping programs with domain ontologies. Once the mappings are explicitly realized, they can be subsequently used by other developers. This is especially useful for developers that are new to a project. Furthermore, the explicit links between a program and an ontology (seen as semantic domain) define a conceptual based meaning of the program. In turn this enables different parts of a program to be seen from the point of view of the concepts that they implement. This change of perspective enables a new set of automatic program analyses (e.g. conceptual analysis of identifiers names, logical duplication) and enriches the already existing ones (see 3.).

2. Assessing the quality of APIs. By mapping a domain ontology to existing APIs we analyze the *measure* and *manner* in which the APIs reflect the domain. Thereby, we can identify situations in which an API implements domain concepts in a way that does not match to the domain knowledge [15, 13]. Furthermore, situations in which domain concepts are not implemented in APIs can be identified and thereby we assess the (conceptual) coverage of APIs with respect to their domain. Depending on how the domain concepts are implemented by an API, the API can or cannot be extended towards implementing new domain concepts.

3. Enriching program analysis. Many of the current wide-spreaded static analyses (e.g. clone detection, design quality assessment) are at a pure syntactical level. However, the proper interpretation of their results requires semantical information about the relation of the program to the real-world knowledge. For example, if a class is reported as affected by a design flaw, the relevance/criticality of the flaw could be weighted by the *conceptual centrality* of that class. Similarly, domain knowledge might lead to a better detection of architectural violations (*e.g.* if UI concepts appear in what is supposed to be the persistency layer of an application). Furthermore, domain knowledge can drive (or at least assist) the process of a system's restructuring, by helping decide how the intelligence of a system fragment can be redistributed so that the conceptual integrity of the system gets improved.

4. Indexing and documenting reusable components. Having an ontology that covers a technical domain can enable an advanced search for components (e.g. domain specific libraries) that cover the domain. For example, the programmers can choose a set of relevant concepts from a domain and query which component provides these concepts – e.g., according to the coverage of the AWT (presented in Section 5), a programmer that needs to use tree widgets should choose SWING and not AWT. Once the mapping between a domain ontology and an API is realized, it can be used as a documentation of the API.

5. Teaching and technology transfer. A domain ontology can offer a good starting point for teaching and disseminating programming technologies. Furthermore, it can serve as basis for defining a common vocabulary within a project. Such an ontology could answer the questions like for example: What are the central concepts of a library and how are they related? The central concepts of a technology should be taught before concepts that are only at the margin of the domain. A common ontology of technological knowledge enables a comparison of the design choices that were made in reflecting the concepts by different APIs. This can lead to a catalog of best practices in designing APIs.

6. IDE support for programming. Enriching the current IDEs with programming technologies knowledge opens new possibilities also in the forward engineering. A semantically enhanced IDE can "look over the shoulder" of the programmers and can help in different programming tasks. For example, by knowing the real-world meaning of two variables the IDE might give warnings when one is assigned to the other in the case when the concepts that they represent are not compatible. We regard this as a kind of conceptual type checking.

We are convinced that the possible applications of the ontologies repository are not limited to the ones enumerated above. The success of the repository will be measured in the number of users and usages of its ontologies.

---

3 Knowledge sharing through ontologies

To support sharing and reusing the 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 an area which is to be described for a specified purpose. It contains the set of objects and concepts together with their properties and interrelationships [8]. An ontology is defined to be an *explicit specification of a shared conceptualization* [9] and is used for sharing the knowledge about a domain by making the concepts and relations within it explicit. The term "shared" means that an ontology represents an agreed body of knowledge and the term "specification" implies that this conceptualization is defined in a rigorous manner.

Formalization level. There is a wide spectrum through which ontologies can be seen from the point of view of specification detail [11]. At the lowest level of detail are *controlled vocabularies* which are nothing else than lists of terms. The next level of specification are *glossaries* which are expressed as lists of terms with associated meanings presented as glossary entries in natural language. *Thesauri* provide additional relations between their terms (e.g., synonymy) without assuming any explicit hierarchy between them. Many scientists prefer to have some hierarchy included before a specification can be considered an *ontology*. The most important hierarchical relation in ontologies is the *"is-a" relation*. At the next stage are *strict subclass* hierarchies which allow the exploitation of inheritance (i.e., the transitive application of the "is-a" relation). More expressive specifications include classes attributed with *properties* which, when specified at a general level, can be inherited by the subclasses. A more detailed specification level implies value restrictions for properties; the most expressive ontologies allow the specification of arbitrary logical constraints between their members. Even if at the end of the ontology specificness spectrum are fully axiomatized ontologies, most of the existent ontologies today are weakly specified.

In our work we use an informal meaning of the term "ontology" - which we regard to comprise only concepts arranged in a taxonomy and relations between them (see Figure 3). We do not require any consistency checks, restrictions on properties or logical inference to be defined or that the ontology terms obey to inference rules. This is however, the most common way in which ontologies are currently used in practice. In order to represent an ontology we use a graph language similar to the RDF(S) graphs [10]. Entities within the ontology are the nodes of the graph. Each relation between two nodes of the graph represent a relation between the entities in the ontology.

Obtaining ontologies. To the best of our knowledge, there are no *off-the-self ontologies* that comprise the knowledge about the programming technologies at the abstraction level of the program code and can be used in code analyses today.

In [13] we proposed a method for manually building ontologies that are suitable for analyzing APIs. However, many times such ontologies comprise a large number of concepts and therefore they are difficult to build and validate manually. In [14] we propose a method for extracting domain ontologies through the analysis of commonalities of several domain specific APIs that address the same domain (we will briefly present this method in the next section). This method has the following advantages:

Efficiency: Extracting the domain knowledge from APIs and after this (manually) validating the results is much more efficient than building the ontology by hand.

Completeness: Even if the programmers used different tools and even programming languages, there is a constant in their work: the domain which is to be represented. The different programmers model the same domain from different perspectives and when the number of analyzed APIs is big enough then we can achieve a comprehensive coverage of the domain.

Acceptance: By analyzing the APIs as they are implemented, we obtain ontologies that are very close and at an appropriate abstraction level to the common knowledge that the programmers have about a domain. Since the APIs that we analyze are widely used, this knowledge is accepted by a large community.

4 Extracting Ontologies from APIs

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 that are mixed with representations of the domain knowledge in their interfaces. In order to overcome these problems, we developed a technique for extracting the domain knowledge based on the similarities of several APIs that cover the same domain [14].

In Figure 2 (left) we illustrate this situation intuitively. 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. The lower part

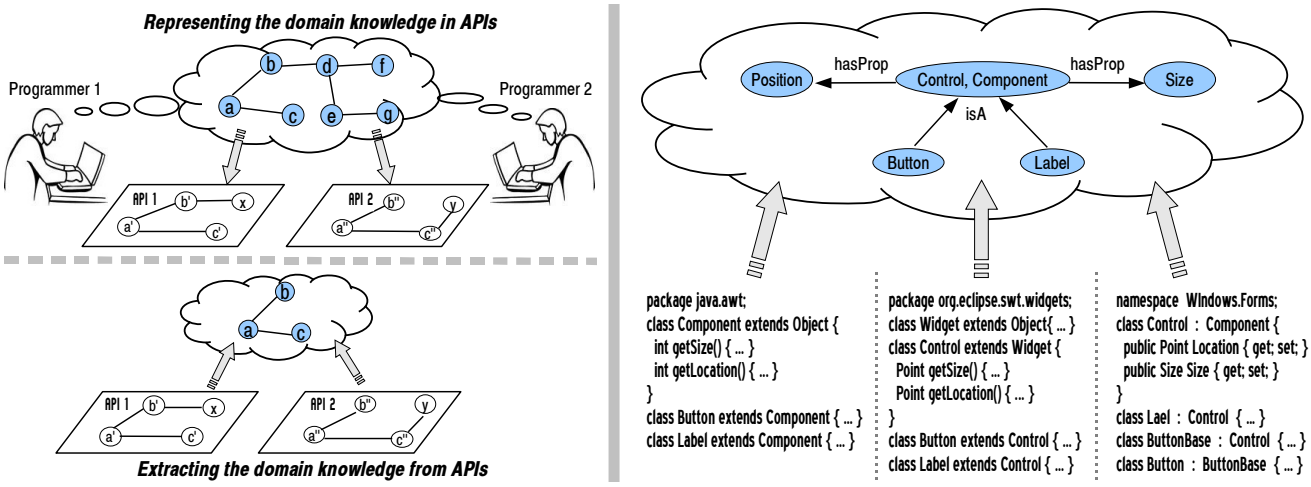

Figure 2. Domain knowledge reflected in APIs

presents the approach taken in this paper to extract the domain knowledge: 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 relations 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 2 (right) we present examples of common 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 this knowledge is reflected in three of the most well-known GUI APIs (Java AWT, Eclipse SWT and .NET).

The challenge of extracting the domain knowledge automatically is threefold: Firstly, we need to identify a way to *uniformize* the possibly different implementations of the same real-world situation (Section 4.1); secondly, we need to identify a proper *abstract representation of APIs* that facilitates their comparison (Section 4.2) and thirdly, we need to *filter out the noise* introduced by particular implementation details (Section 4.4).

4.1 Reflecting abstract relations in APIs

We regard an ontology to comprise concepts and the following relations between them: *isA* between the superordinate and its subordinates (e.g. Component – isA – Window); *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). An example of a fragment of such an ontology is presented in Figure 3.

In order to identify the similarities between the APIs and

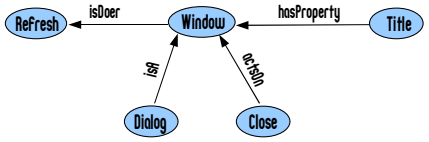

Figure 3. Example of an ontology fragment

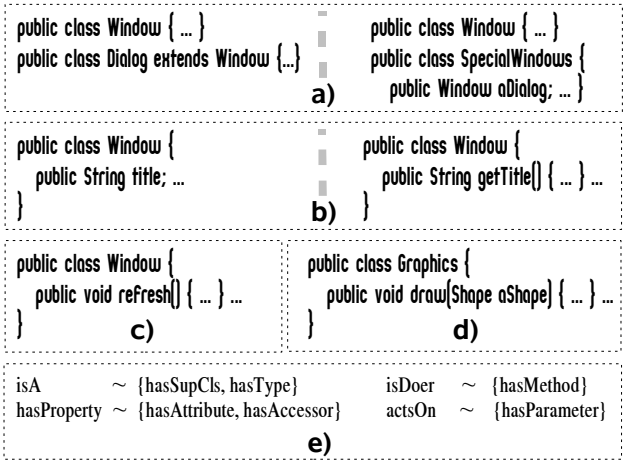

Figure 4. From abstract relations to APIs

thereby to extract the ontology, we need to investigate 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 type-system generated relations: either through the sub-classing relation or through a relation between a variable and its type. In Figure 4a 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 or through accessor methods. In Figure 4b we exemplify how the "Window – hasProperty – Title" is reflected in an API.

Reflecting the "isDoer" relation in APIs. The *isDoer* relation between an entity and the action it performs is reflected in APIs through a relation between a class representing the entity and a method representing the action. In Figure 4c we exemplify how is the "Window – isDoer – Refresh" represented in an API.

Reflecting the "actsOn" relation in APIs. The *actsOn* relation between an action and the affected entity is reflected in APIs through a relation between a method representing the action and its parameters representing the entity. In Figure 4d we exemplify the implementation of the relation "Draw – actsOn – Shape".

The similarities between the ontology and program level relations are presented in Figure 4e.

4.2 Formalizing APIs

We use a graph-based representation of the program elements from the public interface of the APIs and model explicitly the names of the program elements. The full formalization is presented in [13, 14].

Representing APIs as graphs. We describe an API as a labeled directed graph. The nodes of the graph are the program elements accessible to the users of the API (P). Its edges are typed relations defined in the program among these program elements. Given a pair of nodes of the program graph (p_1 and p_2), we define the function $e(p_1, p_2)$ (edge) to return 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 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.

Example: In the lower and upper parts of Figure 5 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: $e(Widget, size) = hasAtt$. To denote the fact that $hasAtt$ is the relation between the nodes $Widget$ and $Size$ we use the following notation: $hasAtt(Widget) = size$

Lexical information. 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. The library's vocabulary (I) is represented by the set of program element 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 developers and their users. The program and the lexical layers are linked through the function program-element-to-identifier (P_2I) that maps the program elements to their names. The function identifier-to-words (I_2W) is responsible for obtaining the set of normalized words contained in the identifiers' names. 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 5 we present an example of the lexical layer corresponding to the two APIs. It is centered around a set of words W that is obtained through the splitting the identifiers from API_1 (I_1) and API_2 (I_2). Examples of the function identifiers-to-words are: $I_2W('getHeight') = \{'Get', 'Height'\}$, $I_2W('drawAndMove') = \{'Draw', 'And', 'Move'\}$, $I_2W('XMLNode') = \{'Xml', 'Node'\}$.

4.3 Extraction algorithm

In order to extract the ontology from the APIs we use a graph matching algorithm. Our algorithm (presented in [14]) matches nodes in the API graphs with similar names and paths in the API graphs that correspond to the implementation of an abstract relation. Whenever a match is found, we identify a "subject – verb – object" triple: the subject and the object are the nodes and the relation among them is the verb. In Figure 4e we present the similarities between the ontology and program level relations.

Example: In Figure 5(right) the graph matching algorithm matches the nodes "Widget" from both of the API graphs and the nodes "getLocation" and "location". Between these nodes are compatible relations, namely $hasAcc$

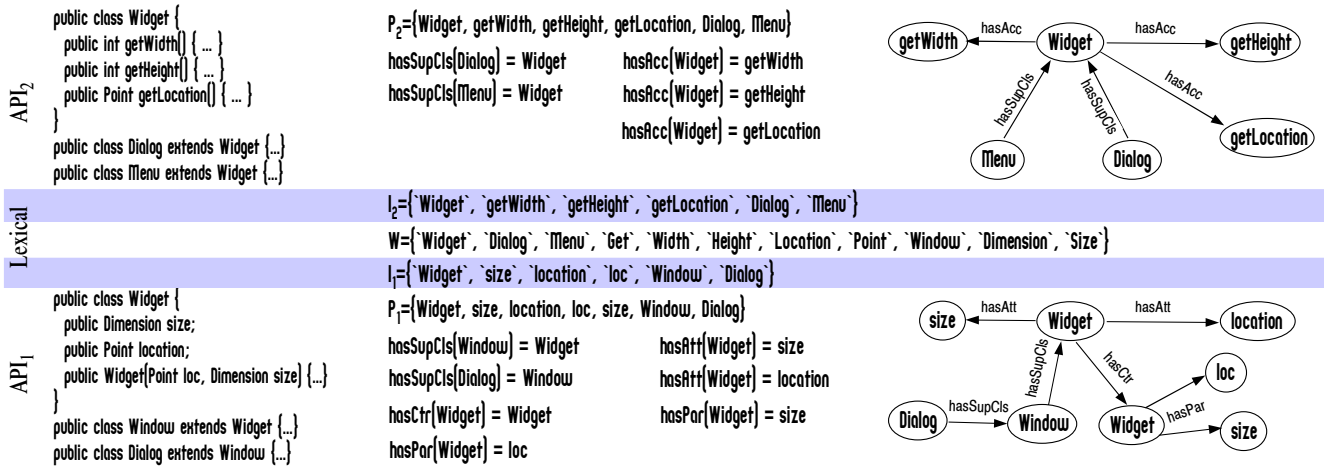

Figure 5. API Layers

and $hasAtt$ that are used to represent in programs the "hasProperty" abstract relation (see Figure 4e).

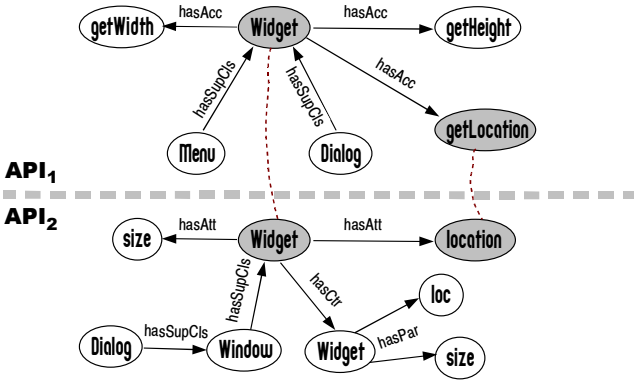

Figure 6. Identification of "Widget – hasProperty – Location"

4.4 Extraction Methodology

Our algorithm 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: *1) Establish the scope of analysis*, *2) Select the set of APIs*, *3) Run the concept identification algorithm*, *4) Eliminate the noise and validate the ontology*.

Transform to OWL. In order to make the ontology accessible by third parties we converted from the triples to OWL format by using the following steps: 1) every subject and object is a OWL class, for each of the verbs: isDoer, hasProperty and actsOn we define an OWL property; 2) every "subject – isA – object" triple is converted to the subclass relation between the corresponding OWL classes; 3) each of the other triples of the form "X – verb – object" (X is a set of concepts) is transformed into an OWL property (p) with domain the union of the OWL classes corresponding to the elements of X and the range the OWL class corresponding to the object. The property p is an OWL sub-property of the OWL property corresponding to the verb (as exemplified below). For the ontology fragment from Figure 3 we produce the following OWL fragment:

```
<owl:Class rdf:ID="Window"/>
<owl:Class rdf:ID="Title"/>
<owl:Class rdf:ID="Dialog">
 <subClassOf rdf:resource="#Window"/>
</Class>
<owl:Class rdf:ID="Refresh"/>
<owl:Class rdf:ID="Close"/>

<owl:ObjectProperty
              rdf:about="#hasProperty"/>
<owl:ObjectProperty rdf:about="#isDoer"/>
<owl:ObjectProperty rdf:about="#actsOn"/>

<owl:ObjectProperty rdf:about="#hasTitle">
 <rdfs:domain>
  <owl:Class>
   <owl:unionOf rdf:parseType="Collection">
    <rdf:Description rdf:about="#Window"/>
   </owl:unionOf>
  </owl:Class>
 </rdfs:domain>
 <rdfs:subPropertyOf
          rdf:resource="#hasProperty"/>
 <rdfs:range rdf:resource="#Title"/>
</owl:ObjectProperty>

<owl:ObjectProperty
                rdf:about="#actsOnWindow">
 <rdfs:domain>
  <owl:Class>
   <owl:unionOf rdf:parseType="Collection">
```

```
    <rdf:Description rdf:about="#Close"/>
   </owl:unionOf>
  </owl:Class>
 </rdfs:domain>
 <rdfs:subPropertyOf rdf:resource="#actsOn"/>
 <rdfs:range rdf:resource="#Window"/>
</owl:ObjectProperty>

<owl:ObjectProperty
               rdf:about="#performsRefresh">
 <rdfs:domain>
  <owl:Class>
   <owl:unionOf rdf:parseType="Collection">
    <rdf:Description rdf:about="#Window"/>
   </owl:unionOf>
  </owl:Class>
 </rdfs:domain>
 <rdfs:subPropertyOf
             rdf:resource="#isDoer"/>
 <rdfs:range rdf:resource="#Refresh"/>
</owl:ObjectProperty>
```

5 Experience

In the following we present our experience with building and using the knowledge repository. Our repository currently contains only two ontologies: an ontology that covers the domain of graphical widgets and one that covers the XML domain. This section follows two directions: at first we report on the building of these ontologies and then we present examples of applying these ontologies for concept location and the evaluation of the domain coverage of the Java AWT API.

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`[2] open source library for working with XML; `jdom` [3] library for accessing, manipulating, and outputting XML data; `xom` [4] tree-based API for processing XML and the XML processing API from the `.NET framework`. The second set of APIs implement the functionality related to graphical widgets: 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`. Later, we found two more GUI APIs that we decided to take into account – the `Biss-AWT`[5] and the `GTK`[6] APIs. We used the extracted ontologies for locating concepts in the JHotDraw framework (version 7.0.9).

Extracted ontologies. In Table 1 we provide the number of automatically extracted concepts and triples (raw) and the number of concepts and of triples after our manual validation. During the validation, we examined the triples and eliminated those that represented pure implementation noise as well as those that referred to very general concepts that are not specific to the targeted domain. We notice that in both of these cases ca. 50% of the extracted triples represented noise. However, this noise can be eliminated relatively easy through manual inspection (in the case of the XML ontology it took us ca. 1 hour and in the case of the GUI ontology it took us ca. 3 hours). These ontologies, both in the triple format as well as in the OWL format can be downloaded from the following web address: `www4.in.tum.de/~ratiu/knowledge_repository.html`.

	XML	GUI
Raw concepts	271	853
Raw triples	674	2709
Validated concepts	164	456
Validated triples	319	1351

Table 1. Quantitative overview

Having an ontology that contains knowledge about programming technologies and that is in a machine processable format is a gain per se. Obtaining such ontologies for all programming technologies (by analyzing other libraries) would cover a wide area of technical knowledge. In the following we give only hints of how such ontologies can be used in software maintenance activities such as concept location and evaluation of APIs.

Concept location. We use a method for locating concepts in code that is based on mapping program entities to ontologies [13]. 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. It contains 7358 model elements that belong to the public interface – public classes, attributes, methods and their parameters. 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 243 concepts in JHotDraw. These concepts were assigned to 1388 program elements from the public interface of JHotDraw.

By inspecting the program elements assigned to XML concepts, we discovered the fact that JHotDraw contains

[2] `www.dom4j.org`
[3] `www.jdom.org`
[4] `www.xom.nu/`
[5] `http://www.biss-net.com/`
[6] `http://java-gnome.sourceforge.net/`

classes that use the `nanoxml`[7] 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 7 we present an example of how was the ELEMENT concept identified.

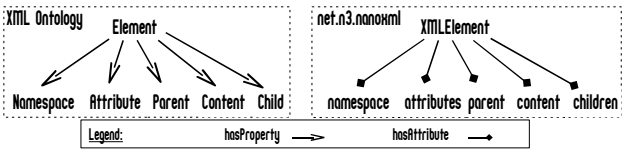

Figure 7. Concept identification example

Analyzing the coverage of APIs. In our second experiment we use the GUI ontology in order to identify concepts in the public interface of AWT. In Figure 8 we present a list of concept names that were not identified in the AWT. In order to find out the meaning of these concepts, we looked in the GUI ontology. In Figure 9 we present the triples from our ontology where the missing concepts from Figure 8 occur. We remark that many of the missing concepts are related with each other and that they represent advanced graphical features (tooltips, menu advanced accessibility, special dialogs, browsing support, etc). By consulting a comparison of AWT, SWING and SWT [19] we found out that we accurately identified the coverage of AWT. The explanation for the lack of advanced components is that AWT is the lowest-common denominator for GUI components defined for all Java host environments. Some commonly used components, such as Tables, Trees, Progress Bars, and others, are not supported. For applications that need more component types, you need to create them from scratch.

Accelerator	Editor	PrintToFile	SelectionIndex	ToolItem
Accessible	Empty	PrintDialog	SelectionMode	ToolTip
AccessibleDescription	EventSelection	Progress Bar	Slider	ToolTipText
Back	Expand	RadioButton	Spinner	Tree
Browser	FontDialog	RadioMenuItem	Table	TreeCollapse
CheckBox	Forward	Refresh	TableColumn	TreeEditor
ClearSelection	HtmlDocument	Scrollable	TableEditor	TreeExpand
Combo	NextNode	ScrollBar	ToggleAction	TreeNode
ComboBox	Node	ScrollEvent	Tool	
Copy	Paste	Selection	Toolbar	
Cut	PageSetupDialog	SelectionCount	ToolButton	

Figure 8. Concepts not implemented by AWT

Additionally, we further investigated the documentation of AWT. We found that some of the concepts that we identified to be missing were in fact (indirectly) supported by AWT (Figure 9 down). For example, even if AWT does not provide direct support for radio buttons, they can be simulated through check-boxes: the class `Checkbox` represents

[7] `http://nanoxml.cyberelf.be/`

Menu	hasProperty	Accelerator	Font Dialog	hasProperty	Font
Menu	hasProperty	Accessible Context	Font Dialog	isA	Dialog
Menu Item	hasProperty	Accelerator	Font Dialog	isA	Window
List	isDoer	Clear Selection	Print Dialog	hasProperty	Print To File
List	hasProperty	Selection	Print Dialog	isA	Dialog
List	hasProperty	Selection Index			
List	hasProperty	Selection Mode	Table	hasProperty	Column
			Table	hasProperty	Editor
Tree	hasProperty	Selection	Table	hasProperty	Selection
Tree	hasProperty	Selection Count	Table	isDoer	Select
Tree	isDoer	Tree Collapse	Table Editor	isA	Editor
Tree	isDoer	Tree Expand			
Tree Node	hasProperty	Next Node	Table	hasProperty	Tool Tip Text
Tree Node	hasProperty	Node	View	hasProperty	Tool Tip Text
			Tree	hasProperty	Tool Tip Text
Radio Menu Item	isA	Menu Item	Tool Tip	hasProperty	Initial Delay
			Tool Tip	hasProperty	Reshow Delay
Show	actsOn	Url	Tool Tip	hasProperty	Text
Browser	hasProperty	Url			
Browser	isDoer	Back	Progress Bar	hasProperty	Style
Browser	isDoer	Forward	Progress Bar	hasProperty	Text
Browser	isDoer	Refresh	Progress Bar	isA	Component
			Progress Bar	isA	Control
Page Setup Dialog	isA	Dialog	Progress Bar	isDoer	Paint
Text	isDoer	Copy	Radio Button	isA	Button
Text	isDoer	Cut	Radio Button	isA	Toggle Button
Text	isDoer	Paste			

Figure 9. Definition of the concepts not implemented in AWT

also radio buttons and to create a radio button one needs to create a checkbox and add it to a group. Another example, are the concepts CUT, COPY and PASTE. Figure 9 shows that these concepts belong to the set of actions that are done by/on texts. From here, we deduced that the AWT does not provide the functionality for copying text inside any of its component. The manual inspection of the AWT documentation revealed that the `TextArea` class from AWT provides this kind of functionality. However, it is implemented in another manner and that is why we could not find it automatically. The AWT implementation of these text operations (e.g. in the classes `TextComponent` and `TextArea`) is of algorithmic nature – the cut, copy and paste are implementable through a combination of selection, insert and replace.

6 Scaling-up with Semantic Technologies

The most obvious link between our work and the semantic web technologies is the usage of ontologies for sharing the domain knowledge. The research on ontologies performed in the semantic web and knowledge representation communities led to a lot of valuable knowledge, methods

and tools for dealing with big ontologies. Below we point out some of the most important problems and limitations of our approach based on our experience. We envision that semantic-web technologies can help in our endeavor of building, validating, enriching and evolving ontologies. Below we enumerate some of the most important open issues that we currently face:

1. Obtaining richer ontologies. Currently we obtain only light-weighted ontologies with a small number of relation types between their concepts. Our ontologies can be enriched along two directions: a) obtaining more relation types between the concepts and (e.g. we use currently the *hasProperty* both for properties and for parts of a concept) 2) towards heavier weighted ontologies with constraints between their concepts and relations.

2. Evolving the extracted ontologies. After the first version of an ontology is built, it will need to be evolved with new concepts and relations obtained by analyzing other APIs. The new (partial) ontology needs to be merged with the already existent ontology. This raises problems with respect to the consistency of the merged ontology in terms of conflicting relations between concepts or ambiguous terminology (e.g. synonymy).

3. Manipulating big ontologies. In order to efficiently manipulate (e.g. visualize, modify, check) big ontologies (e.g. such as the GUI ontology that has over 450 concepts) we need special tool support. This support goes beyond the boundaries of a single tool – we rather need more mature technologies for dealing with ontologies.

4. Validation of ontologies. In order to eliminate the subjective bias in validating the ontology, the validation should be performed independently by several domain experts. For example, some of the concepts belong to the core of the domain but many other belong to the margins of the domain but are (debatable) still relevant for describing the domain.

5. Enrichment of ontologies. Once an ontology is validated, it should be manually completed with the missing concepts and relations between them. This manual work requires a lot of effort and implies a high degree of subjectivity. Furthermore, when new APIs are available, their information should be easily added to a domain ontology.

6. Combination of ontologies. The peripheral concepts from one ontology can be central in other ontologies. For example, the concepts related to exception treatment are peripheral for GUIs but central for an ontology focused on errors treatment. Therefore, we need to detect the overlapping and between several domain ontologies.

Most of the above problems are central topics in the semantic web community. Therefore, our project of building a repository of ontologies about programming technologies could highly benefit from the semantic web research. For example, the existent ontology management tools can be used for manually inspecting, visualizing or enhancing the extracted ontologies; the already existent semantic-web technologies for ontologies merging and alignment [7] could be used for evolving and combining different ontologies.

7 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 [2] 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 [3]. [4, 12] 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.

In this paper we advocate that in order to increase the abstraction level at which the automatic analyses are done, the analysis tools need to be aware of the domain knowledge. One of the modalities to share and formalize the knowledge is through ontologies. In this paper we discuss the benefits and the challenges that a repository of ontologies that contain knowledge about the programming technologies bring in the reverse engineering.

Knowledge representation in programs. This paper is in continuation to our previous work on knowledge representation in programs. In [15, 13, 16] we investigate different problems related to the implementation of domain knowledge programs with focus on domain specific APIs. One of the preconditions for the automatic location of concepts and detection of API problems is the availability of domain ontologies. Building large domain ontologies that contain hundreds of concepts and relations between them is challenging. Once such ontologies are available, they can be used off-the-shelf for a wide variety of program analyses.

Ontologies in software maintenance. The LASSIE system [6] 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 and 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. [20] presents an approach for represent-

ing 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 through the analysis of the program's documentation.

The ontologies provided by our repository can be used as complementary sources of knowledge that addresses technical domains typically implemented in APIs.

Extracting ontologies. [18] presents a method for extracting an ontology that corresponds to an API by analyzing the javadoc comments. The motivation for this work is description of web services based on the functionality of their underlying implementation. In [21] is proposed an approach for extracting ontologies from legacy systems (COBOL in this example). The ontology is extracted through a transformation of program elements in entities of the ontology: records in classes, sub-record relation among records in the is-a hierarchy between their corresponding classes and of program functions in functions in the ontology. This approach provides a representation of a program as an ontology.

We advance in the direction of extracting ontologies from programs along two directions: Firstly, we capture the domain knowledge by analyzing multiple APIs. Secondly, we extract ontologies by analyzing the structure of the program (APIs) and not by performing natural language processing.

8 Conclusions and Future Work

In this paper we advocate that a repository of common programming technologies knowledge would increase the automation and abstraction level at which many of the current programming and maintenance activities are performed. We also present a method for efficiently building such a repository based on the semi-automatic extraction of domain knowledge from domain specific APIs. We present our experience with using ontologies for program analysis. Based on our experience we identify key issues in scaling up and where can the semantic technologies contribute.

We are aware that this work represents only the first steps in achieving a mature repository of programming technologies knowledge. However, once a repository is available, it will both increase the abstraction level of the current reverse engineering analyses and will enable new code analyses. Therefore, we strongly believe that such a repository is of interest for the maintenance community and that it should be built, validated and enhanced as a community process. We are currently working to extend the repository with knowledge about other technical domains (e.g. networking, databases) and by analyzing more APIs that are written in other languages (e.g. C++ and Smalltalk).

References

[1] N. Anquetil. Characterizing the informal knowledge contained in systems. In *WCRE'01*, pages 166–175. IEEE CS, 2001.

[2] N. Anquetil, K. M. de Oliveira, K. D. de Sousa, and M. G. B. Dias. Software maintenance seen as a knowledge management issue. *Inf. & Sof. Tech.*, 49(5):515–529, 2007.

[3] N. Anquetil, K. M. de Oliveira, M. G. B. Dias, M. Ramal, and R. de Moura Meneses. Knowledge for software maintenance. In *SEKE'03*, pages 61–68, 2003.

[4] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE'93*. IEEE CS, 1993.

[5] R. Clayton, S. Rugaber, and L. Wills. On the knowledge required to understand a program. In *WCRE'98*, page 69, Washington, DC, USA, 1998. IEEE CS.

[6] P. Devanbu, R. Brachman, and P. G. Selfridge. Lassie: a knowledge-based software information system. *Commun. ACM*, 34(5):34–49, 1991.

[7] M. Ehrig. *Ontology Alignment - Bridging the Semantic Gap*. Spinger, 2007.

[8] M. R. Genesereth and N. J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[9] 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.

[10] P. E. Hayes. Rdf semantics. Technical report, W3C Recommendation, 2004.

[11] D. L. McGuinness. Ontologies come of age. In *Spinning the Semantic Web*, 2003.

[12] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *IWPC'02*. IEEE CS, 2002.

[13] D. Ratiu and F. Deissenboeck. From reality to programs and (not quite) back again. In *ICPC'07*. IEEE CS, 2007.

[14] D. Ratiu, M. Feilkas, and J. Juerjens. Extracting domain ontologies from domain-specific APIs. In *CSMR'08*. IEEE CS, 2008.

[15] D. Ratiu and J. Juerjens. The reality of libraries. In *CSMR'07*. IEEE CS, 2007.

[16] D. Ratiu and J. Juerjens. Evaluating the reference and representation of domain concepts in APIs. In *ICPC'08*. IEEE CS, 2008.

[17] I. Rus and M. Lindvall. Knowledge management in software engineering. *IEEE Software*, 19(3):26 – 38, May-June 2004.

[18] M. Sabou. Extracting ontologies from software documentation: a semi-automatic method and its evaluation. In *ECAI-OLP'04*, 2004.

[19] SWT, Swing or AWT: Which is right for you. www.ibm.com/developerworks/grid/library/os-swingswt/, 2008.

[20] R. Witte, Y. Zhang, and J. Rilling. Empowering software maintainers with semantic web technologies. In *ESWC'07*, pages 37–52, 2007.

[21] 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.