Programs are Knowledge Bases

Daniel Ratiu and Florian Deissenboeck
Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
`{ratiu|deissenb}@in.tum.de`

Abstract

Gaining an overview of the concepts represented in large programs is very demanding as multiple dimensions of knowledge appear at different abstraction levels throughout the source code. To reduce the overall comprehension effort it is therefore desirable to make the knowledge once gained explicit and shareable. We tackle this problem by establishing a mapping between source code and conceptualizations shared as ontologies. To achieve this we regard programs themselves as knowledge bases built on the programs' identifiers and their relations implied by the programming language. Making these mappings explicit allows sharing knowledge about the concepts represented in programs. We exemplify our approach on Java programming language and the WordNet ontology and we report on our experience with analyzing an open source system.

1. Understanding Large Programs

Program comprehension accounts for about 50% of the expenses spent during software maintenance [9] which itself consumes the bulk of the lifecycle costs of successful, long-lived software-systems [13].

Most programmers would agree that understanding the high-level coherences of a program (as opposed to understanding localized code fragments like a single method) is one of the most challenging tasks in program comprehension. This is can be best exemplified by a (fictional) experiment: compare understanding an unknown program without any prior information to understanding the program after one of the developers gave you a 10 minute introduction on the program. Certainly the latter task is considerably easier than the former. This raises the question about the nature of the knowledge conveyed to you in the short introduction. As previous research on program comprehension suggests, this is mainly knowledge about the basic *concepts* represented in the program and their *interrelations*. In this context concepts don't necessarily have to be concepts of the application domain, e. g. an account number, they can as well be technical concepts like a stack or sorting algorithm or a part thereof. Unfortunately the conceptual information contained in the source code is often of implicit nature as these concepts are scattered over various locations and different abstraction levels within the source code. Thus the task of locating known concepts in the program as well as extracting concepts from the program are equally demanding.

We present a new approach for extracting concepts from code by mapping the identifiers and the relations between them to ontologies. As a result, we explicitly link the sources with the semantics contained in ontologies. We demonstrate our approach using on the one hand the relations within Java programs generated by the type and the module systems and on the other hand the WordNet ontology and we exemplify our preliminary experiences with a medium sized open-source Java system.

2. Related Work

There's highly valuable work on the program comprehension processes [15] and the knowledge required for understanding programs [3, 6]. To a large extent this knowledge is encoded in programs only in the names used for its entities. [1] highlights the importance of proper identifier naming and defines naming rules. [4] advances on this by providing a formal naming model and treating program identifiers as controlled vocabularies. One of the most important comprehension activities is the assignment of human-oriented concepts to their implementation in the source code [2]. This task is usually referred to as *concept location* or *concept extraction* (in the other direction). Most work agrees that identifier-based concept location strategies are the most intuitive [14] if the identifiers are chosen properly.

The LASSIE system [5] uses a knowledge base for intelligently indexing reusable components. The approach makes a distinction between the domain model and the code model. Although the code model is populated automatically, the domain model and its relation to the code model

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 **IEEE**

must be maintained manually. Such a system proved to support comprehension tasks but the overhead of manually synchronizing the models reduced the overall benefit.

[11] presents a method for manually reverse engineering the ontology of an application based on the features accessible through its user-interface. This work does not provide any link between the ontology and the source code.

3. Knowledge Sharing through Ontologies

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, which contains the set of objects and concepts together with their properties and interrelationships [7]. An ontology is defined to be an *explicit specification of a conceptualization* [8] 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" - we do not require any consistency checking, restrictions on properties or logical inference to be defined or that the terms obey that inference. In order to represent an ontology we use a graph language similar to the RDF graphs [10]. Entities within the ontology are the nodes of the graph. If there is a relation in the ontology between two entities then in the graph there is an edge between the nodes that denote the entities annotated with the name of the relation in the ontology.

4. Knowledge Representation in Programs

In order to efficiently perform maintenance tasks, one needs to have both the knowledge about programming language(s), paradigm(s), algorithms, libraries, program design, domain and how is it reflected in the code. We start from the assumption that an important part of this knowledge is manifested in the names of identifiers and the relations between them.

Even if from a formal point of view names are simple labels, from an informal point of view the names are in the center of program understanding by humans as they introduce a higher-level, not formally checked, conceptual layer in a program. To illustrate this we take a simple code example in which all variables are integers: `position = initialPosition + distance`. This example would be considered acceptable both by the type-checker and by a human, from a conceptual point of view. If we modify the example a bit and write: `position = initialPosition + temperature` we assume that everybody would consider this piece of code to be at least awkward if not wrong, even if it conforms to the programming language semantics. The additional semantics in this case is given by the names that are at a higher conceptual level than the level of checkable relations (e.g. types).

Through every declaration, programmers define new words by making use of the already existing ones. For example in the code snippet in Figure 2 we defined the word *Parent* in terms of the word *Person*. Once a word is defined, it enters the vocabulary of the program and can be subsequently used (e.g. for defining the word mother). The relation between a defined word and the ones used in its definition is (many times) similar to the relations between the concepts represented by these words in the real world.

Thus, we regard programs as knowledge bases where the knowledge representation language contains (a subset of) the programming language used. The knowledge itself is expressed in this language through the names of the identifiers.

5. From Sources to Ontological Entities

Understanding a program implies recovering the concepts that are present in the sources. As we presented in Section 3, ontologies are used for sharing conceptualizations. We assume that the concepts that we need to identify are represented by entities within an ontology. Thus, we identify concepts within a program by creating mappings between parts of it and parts of an ontology (i.e. by identifying ontological entities within a program).

As is illustrated in Figure 1, we use ontology mapping techniques for recovering information from programs. The input is twofold: on one hand the names of program elements and their relations and on the other hand the reference ontologies. The output is given by ontological entities which can be related to program elements.

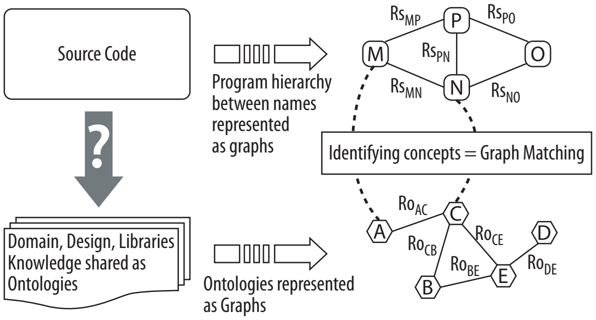


Figure 1. Proposed Approach

Ontologies are represented in different languages and formats. In order to overcome the syntactic heterogeneity between the representation of ontologies and programs, we choose to represent both the sources and the ontologies as labeled multigraphs (see Section 3).

We restrict our approach for identifying the concepts within the code to simply mapping parts of the multigraphs that represent the program to parts of multigraphs that represent ontologies. This is in fact the classical problem of finding a sub-graph homomorphism between multigraphs.

By making explicit the mapping between the code and the ontology, we obtain a new representation of the program centered on the identified ontological entities. This representation contains a higher level of knowledge than the code alone, namely the relations of the identified ontological entities with other concepts from the ontology. Furthermore, it provides a more natural decomposition of the program based on the mapped ontologies as different concepts scattered in the code can be localized within an ontology.

6. Mapping WordNet to Java Programs

We exemplify our generic approach for extracting ontological entities from programs, by choosing the WordNet [12] dictionary as knowledge base and a particular set of program relations from Java. To facilitate the presentation, we use a small example of how basic concepts of family members and their relations are represented in WordNet, how could they be represented in a Java program and how could the mapping between them be recovered.

WordNet WordNet is an online dictionary of English which organizes the words in function of their meanings, in sets of synonyms (synsets). The WordNet 2.0 contains over 150,000 words grouped in over 115,000 sets of synonyms, out of which more than 70% are nouns. Due to the words polysemy, every word can express more lexicalized concepts and due to the synonymy every lexicalized concept can be represented through more words. The synsets are organized hierarchically along the hyponymy / hypernymy (i.e. “is a kind of”) relation. Every noun definition consists of its immediate hypernym followed by meronyms (i.e. “part of”). Meronyms are features that can be inherited by hyponyms.

6.1. Relations between Java Program Elements

In order to map Java program entities to the WordNet ontology, we first need to identify program relations that are similar to the relations within WordNet. We classify the language defined relations between program elements according to two criteria: relations induced by the module system and induced by the type system.

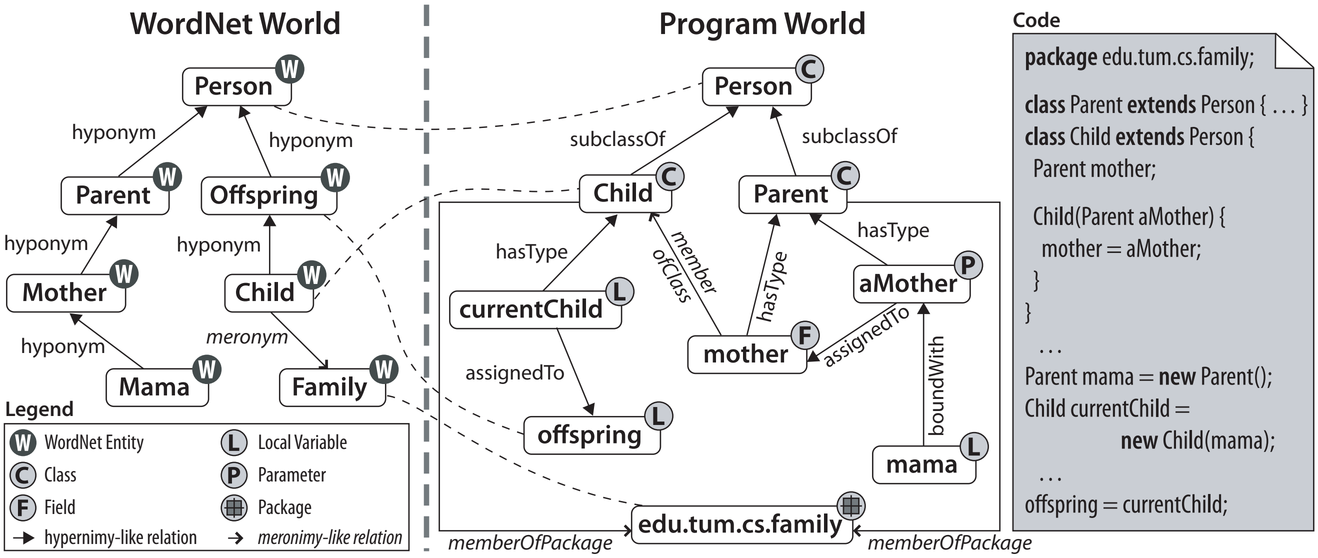
Module system induced relations The module system enables a structural decomposition of programs. It is also a mean to model structural relations from the modeled domain. Thus, relations between modules and their constituents are good candidates to be similar to meronymy relations from WordNet. We consider packages and classes the most important modules in Java. They determine the following relations: **memberOfPackage**, which holds between a package and all its containing classes or interfaces; **memberOfClass**, which holds between a class and all its containing attributes.

Type system induced relations The type system represents a static approximation of the behavior of programs at run-time. Due to the fact that in Java the type system is names based, a well-typed program enforces a certain level of consistency between names (i.e. a certain name in a certain naming context can be used only for certain actions). With the help of typing rules, a wide variety of relations between program elements is enforced. We enumerate below the most important relations: **subTypeOf** is the relation between two types, used as the main mechanism in object-oriented languages to model the *“is a kind of”* relations between real-world entities; **hasType** holds between a variable and its declared type - every variable in a program is explicitly declared to have a certain type and the variable can be used only in the contexts where its type can be used; **assignedTo** holds between a named member of an expression in the right side of an assignment and the assigned variable; **boundWith** holds between a member of an expression in the place of an actual parameter and the formal parameter - whenever an actual parameter is bound to a formal parameter, we have a dependency similar to an assignment in which the left side is the formal parameter and the right side is the expression for the actual parameter.

Based on these relations we represent Java programs as multigraphs. The nodes of the graph are the names of the identifiers and the edges are the above defined relations between their corresponding program elements. In the right part of Figure 2 shows the graph representation of a small code example that represents relations within a family.

6.2. Defining the Mappings

In the previous subsections we discussed the relations within the WordNet knowledge base and similar relations in Java programs generated by the module and the type system. Based on the current configuration (WordNet and Java) we define a mapping between the two multigraphs. A path in the WordNet graph is a sequence of “hypernym” or ”meronym” labels and in the program graph a sequence of relations defined in the previous section. We consider two paths to be compatible if within them is an alternation of



```
package edu.tum.cs.family;

class Parent extends Person { . . . }
class Child extends Person {
  Parent mother;

  Child(Parent aMother) {
    mother = aMother;
  }
}
  . . .
Parent mama = new Parent();
Child currentChild =
            new Child(mama);
  . . .
offspring = currentChild;
```

Figure 2. An Example of Concepts Identification

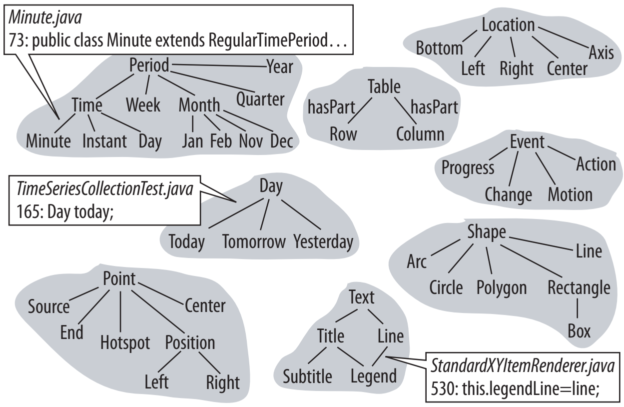
compatible relations. A node in the WordNet graph is an entry in the WordNet dictionary which is based on single words and a node in the program graph is the name of an identifier. We consider two nodes to be compatible if they have a common word.

In Figure 2 we present a hands-on toy-example of our approach. The left side shows the words representing members of a family and their relations as they are given in WordNet. Accordingly the right side shows the identifiers which appear in our code fragment and their relations. The dotted lines represent the links that were identified between program identifiers and concepts within the ontology. For clarity's sake not all links are shown. Through these links we make explicit not only the concepts but also the location in the code where they appeared.

A Real World Example

We present partial experimental results of our approach obtained from mapping the concepts present in the WordNet ontology to the sources of JFreeChart[1], an open source Java library for drawing charts. In Figure 3, Table (a) gives an overview of the quantitative aspects of the example. As tool support we used the Insider[2] and Bridge[3] tools.

In Figure 3 we illustrate some examples of ontological entities automatically extracted from JFreeChart. This figure presents the concepts' names and relations between them: the *"is a"* relation is illustrated through a line between two concepts and the *"part of"* relation through a line with a label attached. There are also Java code fragments to illustrate how do the identified entities and their relations appear in the code. As we can observe in this figure, there are several words that denote different concepts in the code. Such an example is the word "line" which is used to denote both a shape and a piece of text; another one is the word "day" which is used to refer both a period of time and a point in time. By identifying these words within an ontology we distinguish between their senses and thus eliminate the ambiguity generated by polysemy. Providing this information together with ontology enables disambiguation in communication between developers.



```
Minute.java
73: public class Minute extends RegularTimePeriod...
```

```
TimeSeriesCollectionTest.java
165: Day today;
```

```
StandardXYItemRenderer.java
530: this.legendLine=line;
```

Figure 3. Examples of Identified Concepts

In Tables (b,c) from Figure 3 we present the quantitative results obtained after we performed a manual inspec-

---

[1] `http://www.jfree.org/index.php`

[2] Insider is a reverse engineering platform developed at LOOSE Research Group, Technical University of Timişoara, Romania (www.loose.utt.ro)

[3] Bridge is a tool for extraction of concepts from code developed at Technische Universität München, Germany

tion of the automatically extracted concepts and we filtered out the false positives. The number of ontological entities that were identified in the code (172 in our case) represents the knowledge that the tool extracted from the system. It is worth observing that a considerable number of entities were discovered even though only a general ontology was used. However, this number taken alone does not say much about the results. As we can see in the concepts examples figure, we could identify many of the core concepts of the modeled domain (graphics and charts drawing). To roughly evaluate coverage we compare the number of words that were identified as WordNet entities and relate it with the total number of words - this gives a ratio of circa 10%. Interestingly these 10% of words appear in over 20% of the programs' identifiers, which is another sign that these words are central to the analyzed systems. The *Relations in Code* row in Table (b) shows the exact number of places in the code where instances of these concepts were identified (every code relation is determined by two concepts). In Table (c) we present a detailed view over the individual types of relations that appear in the code. It is worth to emphasize the high number of relations at the level of assignments and parameter binding as these capture conceptual coherences in operational contexts.

The last row of Table (b) shows the number of concepts that were identified at low levels of abstraction, i.e. whose names are not part of any class or package names. This category of concepts would be the most hard to identify manually because they are hidden deep in the code and thus require exhaustive search. These concepts are usually not at the core of the modeled domain, they rather represent marginal properties (e.g., colors: green, orange). Furthermore, documenting such concepts with JavaDoc-like tools would not help either because they are usually scattered through the code and so would their documentation be.

	#
Classes	738
Identifiers	7220
Words	1570
kLOC	211

Overview (a)

	#
Ontological Entities	172
Affected identifiers	1503
Relations in Code	826
Low level Concepts	65

Coverage (b)

	#
memberOfPackage	0
memberOfClass	2
subclassOf	70
hasType	282
assignedTo	436
boundWith	36

Relation Types (c)

Figure 4. Case Study Results

7. Conclusions and Future Work

We presented a novel approach for extracting the concepts from source code by mapping the sources on conceptualizations shared as ontologies. The core of our approach is the fact that programs can be regarded themselves as knowledge bases that contain the information in the names of program entities (identifiers) which are related through programming language specific relations. By establishing a mapping from a part of the program to an ontology we identify the part of the conceptualization that is expressed in the particular program fragment. The concepts that we extract are made *explicit* (part of a common accepted conceptualization) and *shareable* (through the conceptualization itself) and thus help to reduce further comprehension effort.

We believe that what we presented here is only the first step in the extraction of a conceptual decomposition from programs. The next enhancements will be targeted towards using more knowledge bases and more sophisticated mapping techniques. One application we're working on is the detection of synonymy and polysemy in identifier naming.

References

[1] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON '98*, page 4. IBM Press, 1998.

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

[3] 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 Computer Society.

[4] F. Deissenboeck and M. Pizka. Concise and consistent naming. In *IWPC '05*, pages 97–106, Washington, DC, USA, 2005. IEEE Computer Society.

[5] P. T. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. Lassie: a knowledge-based software information system. In *ICSE '90*, pages 249–261, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[6] M. B. Dias, N. Anquetil, and K. de Oliveira. Organizing the knowledge used in software maintenance. *Journal of Universal Computer Science*, 9(7):641–658, 2003.

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

[8] 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.

[9] C. S. Hartzman and C. F. Austin. Maintenance productivity: Observations based on an experience in a large system environment. In *CASCON '93*. IBM Press, 1993.

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

[11] I. Hsi and C. Potts. Ontological excavation: Unearthing the core concepts of an application. In *Proceedings of the 2003 10th Working Conference on Reverse Engineering*, pages 345–352, Nov. 2003.

[12] G. A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[13] T. M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996.

[14] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *IWPC '02*, page 271. IEEE CS, 2002.

[15] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.