A Unified Meta-Model for Concept-Based Reverse Engineering

Florian Deissenboeck and Daniel Ratiu

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

Abstract. While programming is modeling the reality, reverse engineering is concerned with recovering it from the code. Parts of this reality can be formalized as concepts and relations among them. As previous research suggests, the identification of these concepts is a key issue in automating program analysis. Their central role requires advance reverse engineering tasks to consider them first-class citizens. In this paper we unify the classical, structure-based reverse engineering meta-models with a meta-model describing concepts and their relations. Our unified meta-model establishes an explicit mapping between concepts and their implementations in a program. Instances of the meta-model are built in a semi-automatic manner by analyzing the program's identifiers. Using this model allows us to raise the abstraction level by viewing the program from the perspective of concepts it implements. This enables a higher degree of automation in the reverse engineering endeavor.

1 Introduction

One of the most frequently cited definitions of *reverse engineering* is:

> Reverse engineering is the process of analyzing a subject system to
> 1. identify the system's components and their interrelationships and
> 2. create representations of the system in another form or at a higher level of abstraction. [1]

Previous research on reverse engineering made great achievements concerning the first item of this definition and was quite successful in creating "representations of the system in another form".

Unfortunately we are still struggling with the latter part of item 2 of the above definition: Creating representations of the system that are not only of another form but also at a higher abstraction layer. Depending on one's definition of "a higher abstraction layer" the achievements so far can be called anything from *encouraging* to *quite disappointing*. In any case, we are currently unable to raise the abstraction level high enough to tackle a number of pressing issues in reverse engineering.

We believe that a promising approach to solve many of these problems lies in the identification of real-world concepts and the establishment of an explicit

mapping between the concepts and their related program elements. In our terms a concept doesn't necessarily have to be a concept of the application domain like an account number. It could as well be a technical concept like a stack or sorting algorithm or a part thereof [2].

1.1 Pressing Issues

This section presents well-known problems in reverse engineering that we consider solved unsatisfactorily today. Section 5 will point out how our approach can help to address these problems.

Analytic Quality Assurance A great number of quality problems are of semantic nature and cannot be detected by code analysis alone [3]. A prominent example is *logical* duplication within programs. As changes to duplicated code may result in unpredictable behavior, duplication has long been recognized as a quality problem that severely hampers software maintenance [4, 5]. Unfortunately today's approaches to find duplication *(clone detection)* are limited to detecting duplicated *code* [6, 7] or lexically similar program fragments [8]. These approaches have limitations as logical duplication is only partially manifested in the representations they focus on.

Another attribute of quality code regards the naming of program entities because incorrect and/or inconsistent naming is known to complicate program comprehension [9, 10]. Due to their inherent semantic nature identifiers almost completely elude automatic quality analysis. Quality assessments are thus limited to using heuristics based on the syntactical representation of identifiers [11].

Dominant Decomposition A number of challenging problems in reverse engineering are rooted in a fundamental problem of the modularization mechanisms of most programming languages: As programs can be modularized in only one dimension, concerns are scattered across multiple modules. This problem is often referred to as "The Tyranny of the Dominant Decomposition". Most approaches to identify these scattered concerns (or *concepts*), be it to assess their proper separation or to identify parts that could be factored out as *Aspects*, are limited to using heuristic methods based on static or dynamic program analysis [12].

The inability to reliably identify the location of the concepts severely hampers two crucial activities in software maintenance: concept location and impact analysis. Questions that typically arise when processing a change request like "Raise the value-added tax from 16%to 19%." are "Where in the program is the tax implemented?" and "What happens if I change this implementation?". Today the answer to these seemingly simple questions takes a developer a considerable amount of time reading source code and documentation as well as a lot of debugging after he implemented the change.

1.2 Programming Means Loosing Information

The failure to solve these problems comes at no real surprise as most reverse engineering methods use *source code* as their exclusive source of information.

Research on *design recovery* clearly points out, that this can be of limited success only since "[...] source code does not contain much of the original design information, which must be reconstructed from only the barest of clues" [13].

While reconstructing the lost information is a tough but solvable problem for human reverse engineers, it is impossible to be carried out by a tool in a fully automated manner. Consequently most methods and tools today try to tackle this problem by applying heuristics that enable them to reconstruct little bits of the lost information from the source and additional code-related artifacts like version management systems [14].

Even though this approach lead to a number of great tools and methods that effectively support reverse engineering, we still claim that when relying on source code as the single source of information, one cannot overcome the fundamental problem of information loss that occurred during programming.

1.3 Regaining the Lost Information

The strategies human reverse engineers apply when solving problems like the ones presented above provide a hint how to tackle these problems: Next to applying *intelligent reasoning*, which we will probably never be able to emulate in a tool, a human makes use of his *knowledge* about the problem as well as the solution domain. Unfortunately this knowledge is *not* stored in the source code. Therefore an important step towards solving many of the problems presented above is to

1. provide an explicit representation of (parts of) this knowledge, and to
2. find a suitable link to the information that *is* stored in the source code.

To realize this we use ontologies to specify the required additional knowledge and define a unified meta-model that includes the source code as well as the knowledge provided by the ontologies. As it is infeasible to manually establish the link between the ontologies and the source code, we developed a semi-automatic approach that creates the link by analyzing the program's identifiers.

1.4 Outline

After giving an introduction on ontologies and their usage for knowledge sharing in Sect. 2, we present our unified meta-model for concept-based reverse engineering in Sect. 3. Its instantiation is described in Sect. 4. Section 5 explains how the model helps to address the reverse engineering problems discussed in this introduction and reports on the experience we made with the model and the accompanying tool-chain. Section 6 relates our approach to previous work in the field. Finally, Sect. 7 summarizes our findings and gives a glimpse on future work.

2 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. A conceptualization is an abstract, simplified view of a domain which is to be described for a specified purpose. It contains a set of objects together with their properties and relations [15]. An ontology is defined to be an *explicit specification of a conceptualization* [16] and is used for sharing the knowledge about a domain by making explicit the concepts and relations within it.

The term "specification" implies that this conceptualization is defined in a rigorous manner. There is a wide spectrum through which the ontologies can be seen from the point of view of the specification detail [17]. 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.

From the point of view of the information that they carry, we consider an ontology to be a shared, formally defined and automatically accessible body of knowledge representative for a particular domain. Depending on the specification detail of this knowledge, there are a wide range of situations in which we can use ontologies [17]: We can use them for *vocabulary control* in order to enforce project standards. We can take advantage of the defined hierarchies available within an ontology in *browsing and navigating* flat sources of information (e.g. similar to "Google directories"). When the ontology is rich enough, then it can be used for *sense disambiguation* or for *ensuring consistency* among the used terms. By identifying the surrounding context of a concept given as input, an ontology can also be used to achieve *completion*.

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 "is-a" hierarchical relation. In order to represent an ontology we use a graph language similar to the RDF graphs [18]. Entities within the ontology are the nodes of the graph. Relations between them are represented as labeled arcs (Fig. 1b). Below we present an example of an ontology which defines a large number of concepts which are lexicalized in English. Subsequently we will use this ontology to exemplify our approach.

The WordNet Ontology. WordNet[1] is an online dictionary of English inspired by psycholinguistic theories of human lexical memory. Instead of organizing the words according to their form, like the majority of other dictionaries do, WordNet organizes the words according to the meaning of the concepts they denote in sets of synonyms (synsets) [19]. WordNet 2.0 contains over 150,000 words, of which more than 70% are nouns, grouped in more than 115,000 sets of synonyms. 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. WordNet defines two different types of relations between the concepts denoted through nouns:

Hypernymy/Hyponymy (Generalization). The synsets are organized hierarchically along the hyponymy/hypernymy (i. e. "is-a") relation. Every word definition consists of its immediate hypernym (superordinate) followed by distinguishing features. Hyponymy is the inverse relation of hypernymy. Both relations are transitive.

Holonymy/Meronymy (Aggregation). In the case of nouns the distinguishing features that are explicitly encoded in WordNet are the meronyms (i. e. "part-of"). Meronyms, which represent parts of a whole, are features that can be inherited by hyponyms. Holonymy is the inverse relation of meronymy. Both relations are transitive.

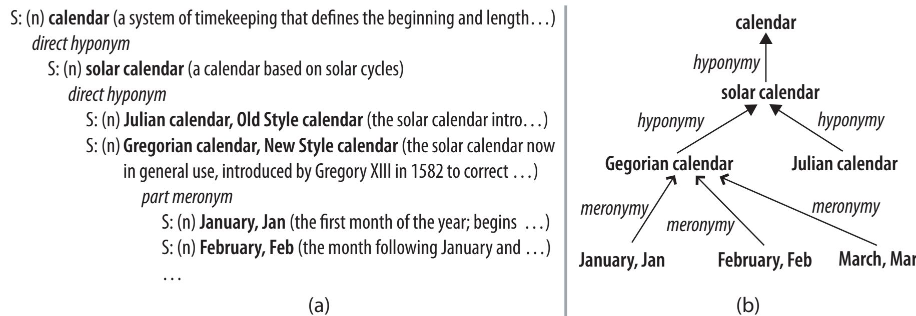

Fig. 1. Example of WordNet entries (a); Representation as a graph (b)

Figure 1a shows an example of how WordNet represents the calendar concept. We notice three hyponymy relations in the calendar hierarchy (e. g. solar calendar is a kind of calendar) and twelve meronymy relations (e. g. January is a part of Gregorian calendar).

3 A Unified Meta Model

In this section we present a unified meta-model that extends the structural meta-models with information related to the names which appear in the program and

[1] `http://wordnet.princeton.edu`

the concepts from an ontology representing the reality modeled in a program. In our view names are like glue between the program and the real-world concepts. Supposing that you would remove this glue (for example through names obfuscation) then the concepts and the program would not be bound any more. As presented in Fig. 2, our meta-model can be at best viewed as having three layers: The *program layer* contains the program entities and the relations among them as they appear in the source code. The *concept layer* contains concepts from an ontology which are relevant for the description of the program (not only the concepts which appear directly in the program but also the ones related to them). The link between the concepts and program elements is realized through the *lexical layer*.

We present our meta-model in a bottom-up manner starting from the level of code, continuing with the level of names and ending with the entities at the level of concepts. When we refer in the text to an element from our meta-model, we will write its name in small caps - e.g. IDENTIFIER.

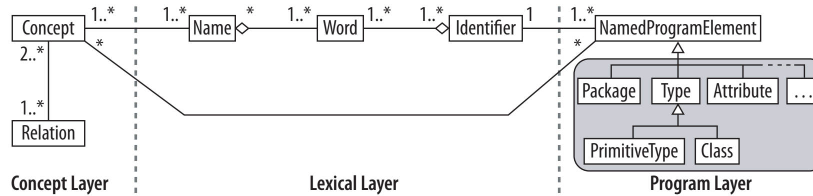

Fig. 2. The Bridge Meta-model

3.1 Program Layer

This level contains a representation of the structure of a program. Such a representation is in fact a structural meta-model for reverse engineering similar to the well-known FAMIX [20]. Bridge, our implementation of the unified meta-model is based on the Memoria [21] meta-model developed at "LOOSE Research Group", Romania [2]. The entities represented at this level are the program elements which have names. An example is given in the gray box in Fig. 2.

NAMEDPROGRAMELEMENT. This entity is the root of the hierarchy of the named program entities which are defined in the analyzed program. The named entities represent the program elements which can be referred through names (e.g., packages, classes), as opposed to the program entities which do not have names (e.g., for loops). Between the NAMEDPROGRAMELEMENT entities are relations corresponding to the semantic of the programming language (e.g. each CLASS entity has a superclass another CLASS) [21].

According to the definition of identifiers found in the specifications of programming languages, they are (almost) arbitrary sequence of characters used

[2] "Politehnica" University of Timişoara, Romania, www.loose.upt.ro

only to enable the reference of the defined program elements later in the program. At the program level we stick to this view. However, in order to be able to link the NamedProgramElements with the real world concepts through their names, we model the program identifiers and dedicate them a layer in our meta-model.

3.2 Lexical Layer

The purpose of this layer is to model the names of the program elements in order to assure the link between the program and the concepts. At this level we need to resolve problems like the equality of program element names (a.k.a. program identifiers), how can they be divided in atomic parts represented through words and how are multiple words used to denote a concept.

Word. Words are lexical entities which correspond to the words in the natural language. In the every-day life the words represent lexicalizations which denote the most frequently used concepts. We consider the Words to be the basic and indivisible entities which carry semantic at the lexical layer. This is why we perform no analysis at the level of parts of a word.

We build the Word entities through the lexical analysis of the program identifiers. We consider a word to be a sequence of characters which could denote a concept from the modeled domain. Depending on the modeled domain, a concept can be denoted through different characters; e.g. one can allow the numbers to be part of words or not.

When comparing more Words, we abstract from their capitalization. Furthermore, we also abstract from their morphological derivations. One possibility for this is to compare the stemmed values of their names instead of comparing their exact names (e.g., the words "house" and "houses" are the same since they have the same stem: "hous")

Identifier. This entity corresponds to the name of a program element. Since names of program elements can contain more words every Identifier contains a sequence of Word entities.

We consider two Identifiers to be equal when they contain the same Words in the same order. For example, the program element names 'calendar' and '_calendar' will be represented in our model through a single Identifier object that will contain only the Word representing the "calendar" string.

We consider that an Identifier, offers a lexical interpretation of the name of a program element as a sequence of Words. In defining the Identifier entities, we raised the abstraction level from the sequence of characters to sequence of Words.

Name. Name is an entity which represents the name of a concept from an ontology. The concepts within an ontology can be denoted through one or more words and thus each Name contains a sequence of Words. Two Names are equal

when the set of WORDs that they contain are equal and the order in which they appear is the same.

Since we restrict to lexicalized concepts, we do not take into consideration the concepts that are not named in the real world (e.g. those that are described only through sentences). We have a NAME in our model only when we can associate it to at least one concept. Due to the polysemy, a NAME can refer to multiple CONCEPT entities.

This implies that in the case of a code with obfuscated program element names, our model would not have any NAME entity even though we have IDENTIFIER entities. The main difference between an IDENTIFIER and a NAME is that the former is a simple sequence of words and the later always represents the name of a concept - e.g. in Fig. 4 due to the fact that there is no CONCEPT entity named "String" we have no NAME entity for "String".

3.3 Concepts Layer

The concept layer represents the core part of our meta-model. Building this layer is the main purpose of our endeavors. The explicit links between the concepts and the program elements are key issues in solving the problems presented in Sect. 1.1. The meta-model elements here are concepts and relations between them from an ontology. The concepts do not necessarily all appear in the code; there can also be neighbors of concepts implemented in the code.

CONCEPT. These entities correspond to the concepts defined in an ontology. The concept layer holds all the concepts from an ontology which are relevant for understanding a piece of code. Since we are dealing only with lexicalized concepts, every CONCEPT has one or more NAMEs. Every CONCEPT has one or more RELATIONs to other concepts.

CONCEPTs that are represented in the program have direct access to the NAMEDPROGRAMELEMENTs that implement them. In this manner we can look from a higher abstraction perspective to a piece of code - not only at exactly what concepts are implemented but also how do these concepts relate to others from the ontology.

RELATION. RELATIONs are entities which correspond to the relations between the concepts from the ontology. A concept is uniquely identified in the concepts hierarchy only through a name and the relations to other concepts (e.g. "isA" and "partOf" relations). As each RELATION has a source CONCEPT and a target CONCEPT, one can navigate between concepts.

We can regard the CONCEPTs and RELATIONs as a graph, in a similar manner with the representation of ontologies as graphs (Sect. 2).

4 Instantiating the meta-model

In the previous section we presented our meta-model in a declarative manner, by describing its entities, relations among them and the rationales behind. Up to

know we left underspecified how one can obtain an instance of the unified metamodel. As pointed out in the introduction, a manual construction of the model would be possible but infeasible. In this section we present our (semi-)automatic method for constructing the model.

Through every declaration, programmers define new names in the program by making use of the already existing ones. For example in the code snippet in Fig. 3 we defined the name "GregorianCalendar" in terms of the name "Calendar". Once a program name is defined, it enters the vocabulary of the program and can be subsequently used (e.g. for defining the name "January").

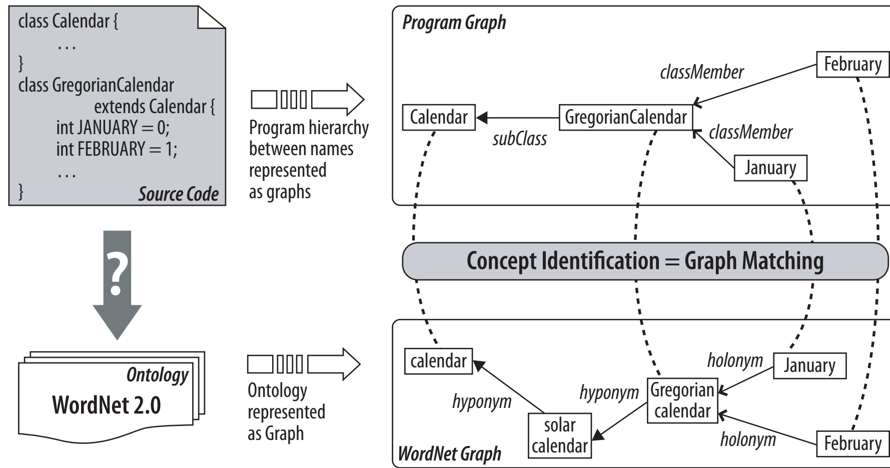

Fig. 3. Concept Identification Approach

In [22] we proposed a new view over programs by regarding them 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. We can then abstract from programs and represent them as graphs (e.g. upper-right part of Fig. 3) whose nodes are the identifiers and whose edges represent the program relations between these identifiers. As an example we consider here the relations generated by the type-system (e.g. subClass) and those generated by the module system (e.g. classMember).

As presented in Sect. 2, 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. Thereby we identify ontological entities within a program.

As illustrated in Fig. 3, we use ontology mapping techniques for recovering information from programs. The input is twofold: on the one hand the names of program elements and their relations and on the other hand the reference

ontologies expressed as graphs. The output is given by ontological entities which are related to program elements.

In Fig. 4 we present an example for recovering the *calendar*, *Gregorian calendar*, *january* and *february* concepts from the sources by mapping the program to the WordNet ontology.

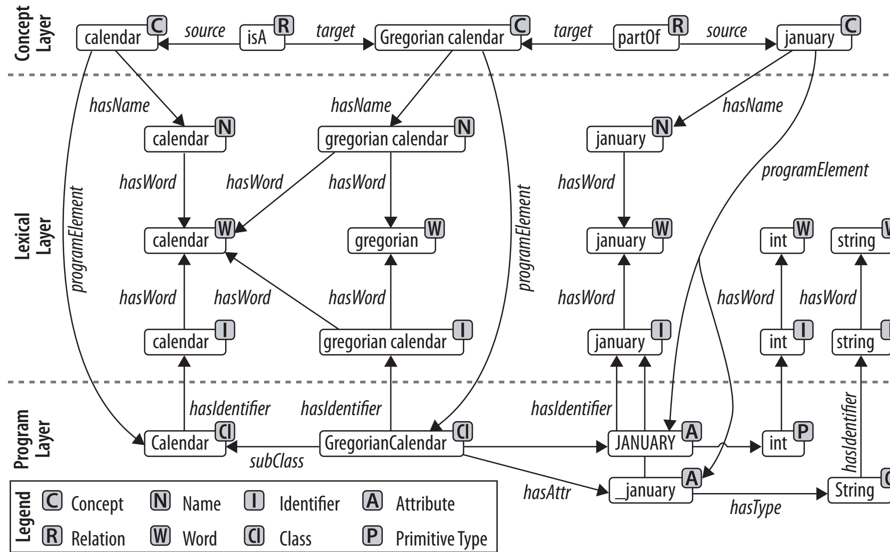

Fig. 4. An instance of the unified meta-model

5 Discussion

Our main motivation for defining the meta-model is to raise the abstraction level at which the automatic code analysis can be performed. In this section we detail on this twofold: Firstly, we present how the unified meta-model could support solving a set of pressing reverse-engineering problems. Secondly, we give concrete examples of the usages of the meta-model.

5.1 Benefits

In Sect. 1.1 we presented a list of reverse engineering issues that, in order to be tackled, require treating the code from a more abstract point of view. Furthermore, it would require explicit links between the concepts and the program entities that implement them. We describe here how can our unified meta-model, presented in the previous section, help in dealing with these issues.

The main innovation of our meta-model is that it brings together the low-level source code and the ontology which describes an intensional, high-level

information within it. Depending on the specification detail at which the ontology is available, the added value of the automatable analysis allows controlling the identifiers of a program, improves program comprehension, supports code navigation and detection of semantic defects which appear in the code. Part of these benefits were already extensively presented in [22, 23].

Code analysis. Raising the abstraction level at which the code is treated is a key step towards a more advanced analysis. Once the unified model is built, it enables us to regard the code from the perspective of the concepts it implements and vice versa to regard the concepts from the perspective of where they are represented in the program. In this way, some fundamental questions for performing the code analysis which regard the traceability of concepts (e.g. Which are the concepts implemented in a program? In which program parts is a concept implemented? What concepts does a program part implement?) are trivial to answer.

Our meta-model provides intra- and interlayer relations between its entities and thus supports navigability (Fig. 5). Once we are at the concepts level, we can use these relations to navigate to the interesting concepts and then once they are reached to go to the code level again e. g. starting from the *value-added tax* we can look at conceptual level what other taxes are related and where are they implemented in the program (e. g. navigate between program elements a and b in Fig. 5).

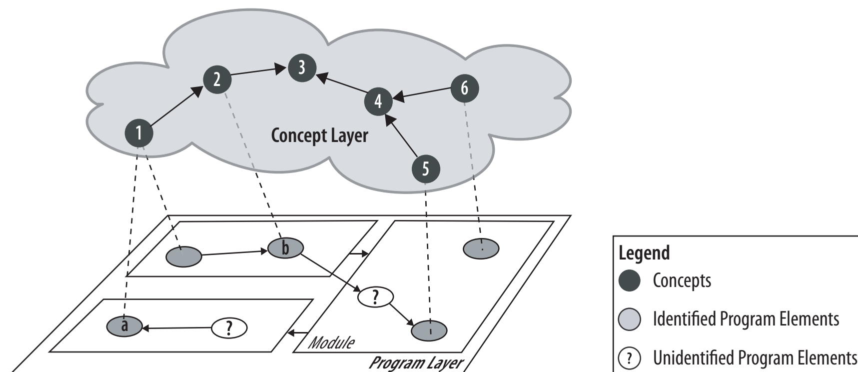

Fig. 5. Regarding the code from a conceptual perspective

Naming defects like synonymy and polysemy appear when there is no one-to-one mapping between the program identifiers and the concepts they represent. In the case of synonymy, a single concept is referred to through multiple names in the code. In the case of polysemy, multiple concepts are referred in a particular program through a single name. By making explicit the links between the concepts implemented in the programs and the names that are used for their implementation, we can easily detect the synonymy and polysemy flaws [23]; e. g. Fig. 6a presents the identification of synonymy.

Logical duplication defects appear whenever a concept is implemented in multiple places in the code (e. g. concept 1 in Fig. 5). There is a big difference between the low level "code clones" and the high-level "logical duplication" - while in the first case the duplication is mainly defined in terms of the source code, in the second case concerns the redundant representation of the real-world knowledge in programs. By making explicit the links between the conceptual and program layers we can automatically detect this class of defects [23]; e. g. Fig. 6b presents the identification of logical duplication.

Conceptual decomposition. The object-oriented programming languages support a single direction of the decomposition of the domain - which is known in the literature as the dominant decomposition. Using the unified meta-model we obtain a conceptual decomposition of programs. The identification of places in the code where a concept is implemented could support the detection of aspects.

5.2 Experience

One of the most important questions is whether it is feasible to automatically build the unified model. Our experiments performed so far show that the automatic construction of the complete model is not feasible but it is possible to soundly construct parts of it. Having constructed it even partially, the unified model still allows us to enhance the abstraction level of automatic analyses even though we loose completeness.

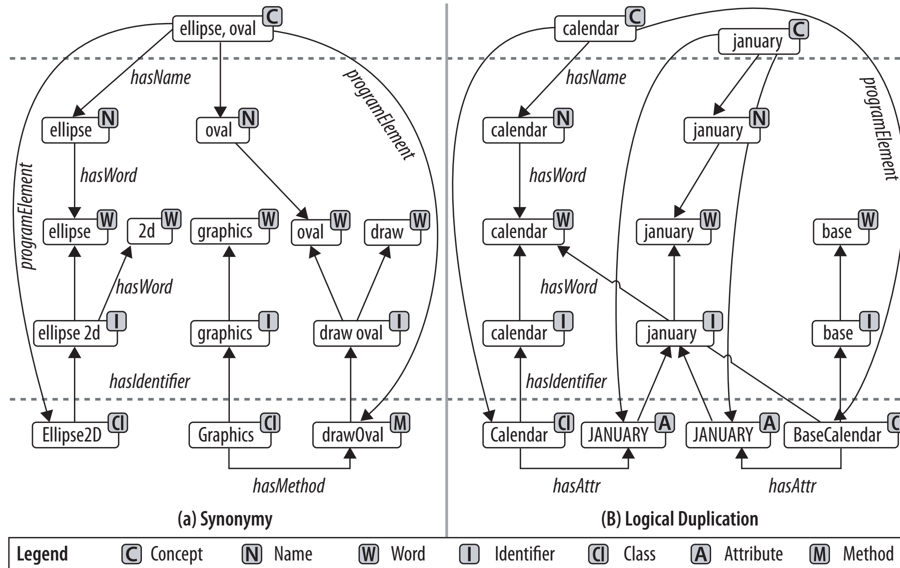

Fig. 6. Examples of the detection of synonymy and logical duplication

In Fig. 6 we present two concrete examples of the identification of semantic defects. These two examples are taken from the Java standard library. The left side of the figure shows the detection of the synonymy defect which we found in the *java.awt* package. Here two distinct names are used (i.e. ellipse, oval) for denoting the *ellipse* concept. The right side shows a case of logical duplication which we also found in the Java library. The *January* concept is implemented as attribute of the class *java.util.Calendar* as well as an attribute of the class *sun.util.calendar.BaseCalendar*. We used the WordNet ontology in order to build the unified meta-model. Although incomplete it proved to be powerful enough for the detection of these flaws.

6 Related Work

To be read conveniently the discussion of the related work is arranged along the levels of our meta-model.

Concept Layer While a lot of previous work well recognizes the importance of real-world concepts for reverse engineering, e. g. [2, 8, 24], we know of relatively little work that treats them as first-class citizens and makes use of external knowledge bases.

The LASSIE system [25] 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 must be maintained manually. Such a system proved to support comprehension tasks but the overhead of manually synchronizing the models reduced the overall benefit.

In [22] we initially presented the idea of considering programs as knowledge bases and described how the links between the program and the knowledge base are established in a (semi-)automatic manner. In [23] we detailed on the application of our meta-model to detect naming deficiencies and logical redundancy.

Lexical Layer There is a lot of highly valuable work that uses the programs identifiers to identify concepts in the source code. Examples are Formal Concept Analysis (FCA) and Latent Semantic Indexing (LSI). FCA is used to identify high-level dependencies in the code by finding groups of elements (called "concepts") that have the same properties [26]. LSI is a statistical approach for extracting semantical information from programs based on textual similarities between files, classes or methods. Sets of words that have a high cohesion in their usage and low coupling with other sets are named "concepts" [27]. Both techniques work only on the program and the lexical layers by detecting concepts described by words or properties that appear in the same configuration repeatedly.

[8] uses LSI for identifying similarities among different files and thus supports the detection of high-level concept clones. The detection and interpretation of clones is done manually.

Recently [11] proposed a method for identifying naming defects without using an external knowledge base. As this approach analyzes lexical characteristics only it is limited to a certain class of defects; it can, e. g., not detect homonymy.

Program Layer A number of meta-models were presented that model structural aspects of programs [20,21,28]. As these model do not provide an explicit representation of the concepts a program implements, they are limited in supporting the developer at reverse engineering activities like impact analysis.

Clone detection tools that work on the program layer have been proofed to be successful in detecting duplicated code [4,6,7], but fail to identify true logical redundancies as they do not explicitly take the implemented real-world concepts into account.

7 Conclusions

Today the automatic support for a number of frequent reverse engineering activities, like impact analysis, detection of semantic naming defects or detection of logical duplication, is unsatisfactory. We believe this is due to the existing approaches' inability to incorporate in their analysis other sources of information than the source code itself. As a good part of knowledge, that would be important for reverse engineering, is lost during programming, we cannot solve the well-known problems without enriching reverse engineering methods and tools with external information sources that make up for the lost knowledge. We propose the application of ontologies to specify parts of this knowledge as concepts and relations between them.

In this paper we presented a meta-model that unifies this conceptual view on programs with the classical structure-based reverse engineering meta-models and thereby enables the establishment of an explicit mapping between program elements and the real-world concepts they implement. We explained how the model is instantiated in a semi-automatic way, discussed how it can help to solve several well-known reverse engineering problems and exemplified its application in the real-life.

While we are convinced that our work is a step in the right direction, we are fully aware that it is only a small one. Significantly more work is not only needed to evaluate the different variation points of our approach but also in validating it more thoroughly with appropriate case studies as well as real-world applications.

References

1. Chikofsky, E.J., Cross, J.H.: Reverse engineering and design recovery: A taxonomy. IEEE Softw. **7**(1) (1990) 13–17
2. Rajlich, V., Wilde, N.: The role of concepts in program comprehension. In: IWPC '02, IEEE CS Press (2002) 271
3. Broy, M., Deissenboeck, F., Pizka, M.: Demystifying maintainability. In: WOSQ '06, ACM Press (2006)

4. Lague, B., Proulx, D., Mayrand, J., Merlo, E.M., Hudepohl, J.: Assessing the benefits of incorporating function clone detection in a development process. In: ICMS '97. (1997)
5. Beck, K., Fowler, M.: Bad Smells in Code. In: Refactoring - Improving the Design of Existing Code. Addison-Wesley, Reading, MA, USA (1999)
6. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: ICSM '98. (1998)
7. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. **28**(7) (2002) 654–670
8. Marcus, A., Maletic, J.I.: Identification of high-level concept clones in source code. In: ASE '01. (2001)
9. Deissenboeck, F., Pizka, M.: Concise and consistent naming. Software Quality Journal **14**(3) (2006) 261–282
10. Anquetil, N., Lethbridge, T.C.: Assessing the relevance of identifier names in a legacy software system. In: CASCON '98. (1998)
11. Lawrie, D., Feild, H., Binkley, D.: Syntactic identifier conciseness and consistency. In: SCAM '06, IEEE CS Press (2006)
12. van Deursen, A., Marin, M., Moonen, L.: Aspect mining and refactoring. In: REFACE '03, University of Waterloo, Canada (2003)
13. Biggerstaff, T.J.: Design recovery for maintenance and reuse. Computer **22**(7) (1989) 36–49
14. Gîrba, T., Ducasse, S.: Modeling history to analyze software evolution. International Journal on Software Maintenance: Research and Practice (JSME) **18** (2006) 207–236
15. Genesereth, M.R., Nilsson, N.J.: Logical foundations of artificial intelligence. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1987)
16. Gruber, T.R.: Toward principles for the design of ontologies used for knowledge sharing. Int. J. Hum.-Comput. Stud. **43**(5-6) (1995) 907–928
17. McGuinness, D.L.: Ontologies come of age. In: Spinning the Semantic Web. (2003)
18. Hayes, P.E.: Rdf semantics. Technical report, W3C Recommendation (2004)
19. Miller, G.A.: Wordnet: a lexical database for english. Commun. ACM **38**(11) (1995) 39–41
20. Tichelaar, S.: Modeling Object-Oriented Software for Reverse Engineering and Refactoring. PhD thesis, University of Berne (2001)
21. Ratiu, D.: Memoria: A Unified Meta-Model for Java and C++ (2004)
22. Ratiu, D., Deissenboeck, F.: Programs are knowledge bases. In: ICPC '06, IEEE CS Press (2006)
23. Ratiu, D., Deissenboeck, F.: How programs represent reality (and how they don't). In: WCRE '06, IEEE CS Press (2006) To appear.
24. Biggerstaff, T.J., Mitbander, B.G., Webster, D.: The concept assignment problem in program understanding. In: ICSE '93, IEEE CS Press (1993)
25. Devanbu, P.T., Brachman, R.J., Selfridge, P.G., Ballard, B.W.: Lassie: a knowledge-based software information system. In: ICSE '90, IEEE CS Press (1990)
26. Arévalo, G., Ducasse, S., Nierstrasz, O.: Lessons learned in applying formal concept analysis. In: ICFCA '05. Volume 3403 of LNAI., (Springer Verlag)
27. Kuhn, A., Ducasse, S., Gîrba, T.: Enriching reverse engineering with semantic clustering. In: WCRE '05. (2005)
28. Bischofberger, W.R., Kühl, J., Löffler, S.: Sotograph - a pragmatic approach to source code architecture conformance checking. In: EWSA 2004, Springer (2004)