

How Programs Represent Reality (and how they don't)

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

Programming is modeling the reality. Most of the times, the mapping between source code and the real world concepts is captured implicitly in the names of identifiers. Making these mappings explicit enables us to regard programs from a conceptual perspective and thereby to detect semantic defects such as (logical) redundancies in the implementation of concepts and improper naming of program entities. We present real world examples of these problems found in the Java standard library and establish a formal framework that allows their concise classification. Based on this framework, we present our method for recovering the mappings between the code and the real world concepts expressed as ontologies. These explicit mappings enable semi-automatic identification of the discussed defect classes.

1. Modeling the Reality

“When I choose a word,” Humpty Dumpty said in a rather scornful tone, “it means just what I choose it to mean – neither more nor less.”

Through the Looking Glass
Lewis Carroll

Programming is modeling the reality. Objects of the modeled domain and their relations are represented by correspondingly related program entities. Unfortunately there still is little tool and process support for establishing a concise and consistent mapping between the real world and programs. This is true even if formal or semi-formal modeling techniques are used as they typically focus on the core of the modeled domain and thereby leave significant parts of programs uncovered.

Today the representation of the external knowledge is usually not considered the central task of programmers. Especially for non-core concepts, their modeling is only a side

effect of the programming activities. Implementation decisions on how to represent the real world concepts in a program are often done in an ad-hoc manner. This leads to a skewed representation of the real world in programs that does not only complicate program comprehension but also introduces potentially dangerous logical duplications. As program comprehension accounts for about 50% of the expenses spent during software maintenance [12] and duplications are recognized as a serious problem [16] we consider tackling these issues a promising approach to improve the overall productivity of software maintenance activities. In order to do so one needs a thorough understanding of the nature of these defects and their consequences. To achieve this, this paper presents a formal framework that allows a unified classification of the naming defects and logical redundancy, that we consider two facets of the problem of concept representation in programs.

Due to the size of today's code bases manual detection of these semantic defects proves to be infeasible and automatic or semi-automatic means of detection are in demand. Therefore one needs to establish an explicit link between program elements and the real world concepts they model. This link, however, is only weakly defined as it strongly depends on the names chosen for program entities. This is exemplified by the source code snippets in Figure 1. Although all three snippets are equivalent from a Java compilers perspective they do differ clearly in terms of the domain they model. Even worse the domain modeled by the snippet in 1c is completely opaque to reader. Names are thus elements which glue together the information about the real world and the program.

Names of program elements can be freely chosen by the developers and are therefore an inherently fragile source of information. Nevertheless previous work [1] shows that in choosing names programmers are typically a bit more careful than Humpty Dumpty is in choosing his words. Furthermore, like words in natural languages, names in programs don't appear in isolation but in relation to other names. An example is the name 'part' in Figure 1. Although the meaning of name 'part' itself is rather ambiguous its context

<pre>class Library { List<Book> books; void addBook(String name, String author, int bookNo) {...} Book getBook(int bookNo) {...} }</pre> <p style="text-align: center;">a)</p>	<pre>class PartsDepot { List<Part> parts; void addPart(String name, String manufacturer, int partNo) {...} Part getPart(int partNo) {...} }</pre> <p style="text-align: center;">b)</p>	<pre>class A { B<C> D; void e(F g, F h, int i) {...} C j(int k) {...} }</pre> <p style="text-align: center;">c)</p>
--	---	---

Figure 1. The Significance of Names

(class `PartDepot`) helps to understand that ‘part’ refers to something like a spare part.

Using a set of carefully chosen heuristics we can exploit this fact to build a graph of program elements’ names and their relations. To detect semantic defects in the mapping between the real world and the program we then match this graph to an ontology that describes the real world’s concepts. This enables us to establish an explicit mapping between the program elements and the real world concepts and thereby paves the ground for a semi-automatic detection of representational defects.

Outline In the next section we present a list of motivating defects that we found by applying our approach to the Java library. In Section 3 we present a formal model that allows a classification of these defects and fosters their thorough understanding. Section 4 presents the semi-automatic method, based on mapping programs to ontologies, that we use to identify the real world concepts in the code. Based on the concepts identification we can automatically generate the list of suspects of semantic defects (Section 5). Section 6 discusses relevant related work. Finally, Section 7 summarizes our findings and gives a glimpse on future work.

2. Semantic Defects in the Java Library

In this section we present examples of defects that we semi-automatically identified in the Java library. We consider these defects to be generated by the flawed representation of the reality in programs. Our examples are divided into two categories: naming problems and logical redundancies. The naming problems we address (i. e. synonymy and polysemy) are well known in the literature to lower the homogeneity of the program, negatively affect the location of concepts and the traceability [9, 1]. The logical redundancies, the implementation of a concept in different parts of the program, negatively affect program’s changeability [3]. Furthermore, when the redundancies are not perfect, they also hamper the program homogeneity and can even lead to bugs when a redundant but not semantically equivalent implementation is substituted for another.

By choosing our examples from the standard Java library instead of making up a toy example, we want to point

out the pervasive appearance of these problems even in one of the most used libraries. The parts of the library which exhibit the defects used in our examples are presented in Figure 2.

In order to facilitate the reading of this paper we follow the following typographical conventions: the `<CONCEPTS>` are written in angled brackets with small caps fonts, the code entities with typewriter font and the ‘names’ are written with single quotation marks.

2.1. Synonymy

When a concept from the real world is implemented in a program, the programmers should refer to that concept by using only one name. This restricts the program ambiguity, increases its homogeneity and facilitates the location of concepts [9, 4, 23, 8, 7].

The Ellipse-Oval Defect Below we present an example from the Java AWT graphical library in which this rule is violated. The concept `<ELLIPSE>` is implemented in this library by using two names: ‘ellipse’ and ‘oval’. As we can see in Figure 2a this concept is implemented in a hierarchy of shapes through the class `Ellipse2D` and as drawing primitive through the method `drawOval()` of `Graphics`. When we look at the documentation of `drawOval()` we see that indeed this method refers to `<ELLIPSE>`. Furthermore, a closer inspection of these implementations reveals the fact that they are equivalent since they represent an ellipse by using the upper-left corner, the height and the width of the rectangle in which the ellipse is contained. This leads us to the conclusion that the `<ELLIPSE>` concept is defined using different names in two relatively close parts of the graphical library and at two levels of abstraction: Firstly it is introduced as a “first class citizen” through a class and secondly as a primitive method used in a drawing utility class¹.

2.1.1. Abbreviations Above we presented a classical example of synonymy. A more frequent variant of synonymy is the usage of abbreviations. In these cases, a concept is denoted both through its full name and through a short name.

The Second-Sec-Minute-Min Defect Our example in this case, refers to the concepts `<SECOND>` and `<MINUTE>` which appear in the public interface of the `Date` class. This class models the concept of `<DATE>` as an instance of time. On the one hand, in this class are methods for accessing the `<SECONDS>` and `<MINUTES>` coordinates

¹More recent versions of the AWT library contain the class `Graphics2D` that solves this problem by offering the method `draw(Shape)`. However, as the original class was not removed the synonymy defect still exists.

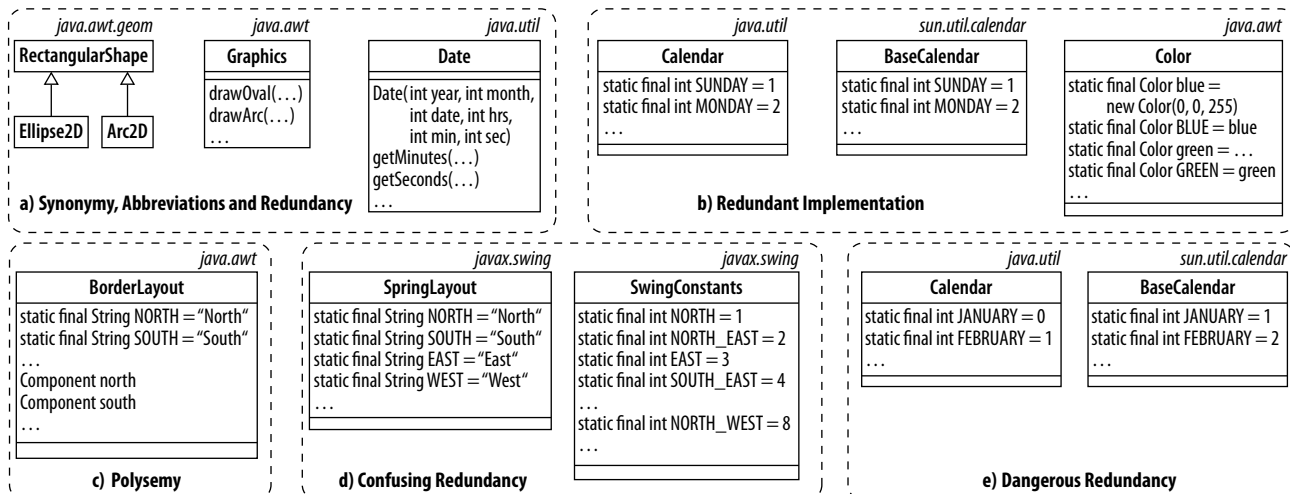


Figure 2. Java Library Fragments

of time named `getSeconds()` and `getMinutes()`. On the other hand, in the constructors of this class are parameters whose names are the abbreviations ‘sec’ and ‘min’ of these concepts - e.g. `Date(int year, int month, int date, int hrs, int min, int sec)`. After the manual code inspection, we remark that this constructor is deprecated and will be replaced by the method `Calendar.set(int year, int month, int date, int hourOfDay, int minute, int second)` in which the parameters use full names and thus the anomaly is eliminated.

2.2. Polysemy

When two distinct concepts are implemented in a program the names of their corresponding program entities should reflect their difference. The software understanding problems generated by the polysemy at the level of program entity names negatively affect location of concepts and increase the program ambiguity [23, 19, 17].

The NORTH-north Defect In the class `BorderLayout` we have two distinct concepts implemented as attributes and which are referred through the same name: ‘north’. The first concept refers to a position and is implemented through the attribute `NORTH : int`. The other sense refers to a component which is placed at the top of the layout and is implemented through the attribute `north : Component`. The only distinction between these names is their capitalization. Below is a code sequence taken from the library and which shows how these two concepts are defined. Interestingly even the author of these lines seems to have been confused enough to use an obviously incorrect JavaDoc comment for attribute `north : Component`

(lines 9–10). There are further ambiguities, e.g. in lines 16–22.

```

1. class BorderLayout {
2.   ...
3.   /**
4.    * The north layout constraint (top of container).
5.    */
6.   public static final String NORTH = "North";
7.   ...
8.   /**
9.    * Constant to specify components location to be the
10.    * north portion of the border layout.
11.    */
12.   Component north;
13.   ...
14.   public Object getConstraints(Component comp) {
15.     ...
16.     if (comp == north) {
17.       return NORTH;
18.     } else if (comp == south) {
19.       return SOUTH;
20.     } else if (comp == west) {
21.       return WEST;
22.     }
23.     ...

```

2.3. Redundant Implementation

A real world concept should be implemented only once in a program. Redundancies are known to negatively affect the implementation of changes [3, 16].

In comparison with the identification of *code* clones which is essentially a bottom-up approach, we address here the problem of *logical* duplications in a top-down manner: We are looking for multiple implementations of a concept in the code.

The Days-of-the-Week Defect The concepts of the day of the week: `<SUNDAY>`, `<MONDAY>`, `<TUESDAY>`, etc. are implemented identically as static constants in the classes `Calendar` and `BaseCalendar` (Figure 2b).

The BLUE-blue Defect The concept $\langle \text{BLUE} \rangle$ is implemented two times in the class `Color` through two attributes `blue` and `BLUE`. These constants are initialized to refer the same object. This situation is the same in the case of all color constants defined in the class `Color`. These redundancies are not code clones (Figure 2b).

2.3.1. Confusing Redundancy In certain cases multiple implementations of a concept can not be avoided. As slightly different implementations increase the ambiguity of the program, one should strive to implement the concept in a homogeneous way.

The Confusing-West Defect The cardinal points in the AWT layout classes are used to define the possible positions of graphical components. In the class `SwingConstants` we have the positions defined as integers while in `SpringLayout` the same concepts are implemented as strings (Figure 2d). A possible reason for this duplication could be the fact that the `SpringLayout` uses only a subset of the cardinal points for specifying the orientation. Still, we see no reason for the differing implementation (string vs integer).

The Duplicated-Arc Defect The $\langle \text{ARC} \rangle$ concept is defined two times in AWT: through the class `Arc2D` and through the method `drawArc()` of the class `Graphics` (Figure 2a).

2.3.2. Dangerous Redundancy If a concept is implemented multiple times in a program, through the same programming constructs, then the different implementations should be interchangeable both from the syntactic and semantic perspective. If two implementations can be interchanged from the compiler's perspective but not from the logical perspective, then they can favor bugs. This happens because programmers need to take care of the concepts that they use and of the implementation details; the compiler does not warn about the possible problems.

The January-January Defect An example in this category is the implementation of months in classes `Calendar` and `BaseCalendar`. As we can see in Figure 2e, they are implemented as public static constants with different values (e.g. `Calendar.JANUARY = 0` and `BaseCalendar.JANUARY = 1`). We emphasize that in the Java library these two classes are used several times in conjunction e.g. `BaseCalendar` is used in the implementation of `Calendar`. We let the following (imaginary) example of a bug originating from the interchanged usage of a quiz to our readers:

```
aCalendar = new java.util.Calendar();
void trickySetDate(int year, int month, int day) {
    aCalendar.setMonth(year, month, day)
}
...
trickySetDate(2006, BaseCalendar.JANUARY, 30);
...
```

3. Formal Defect Classification Framework

We propose a formal framework that allows a concise and unitary description of the semantic problems related to naming and logical redundancies. We regard these defects as originating from the inconsistent representation of the external knowledge in programs. Starting from this framework, we will present in Section 5 our semi-automatic method for identifying these defects.

3.1. The Three Layers of a Program

We define a three layers perspective of a software system: the *program layer*, the *lexical layer* and the *conceptual layer* (Figure 3). In the program layer we consider the named program entities. At the lexical layer we abstract from the code and we focus only on the set of the names that appear in the program. The conceptual layer is the most abstract one and contains the concepts that are implemented in the program.

As we can see in Figure 3, the lexical layer is in the middle and its entities play the role of glues between the program and the conceptual layers.

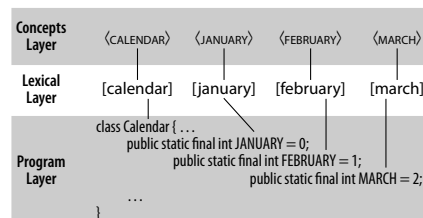


Figure 3. A Three-Layered View

Names Names n are strings and constitute set N . Normalized names are denoted as $\|n\|$, the set of normalized names is denoted as $\|N\|$. The purpose of names normalization is to eliminate their variations (e.g. morphological, capitalization) in order to enable string comparisons as the entry point to identify the correspondence between program and conceptual layer entities. The normalization function is deliberately underspecified as it depends on the granularity of the concept layer.

Concept Layer The concept layer consists of all concepts implemented in the program which constitute set C . A concept does not necessarily have to be a concept of

the application domain like an $\langle \text{ACCOUNT NUMBER} \rangle$. It could as well be a technical concept like a $\langle \text{STACK} \rangle$ or $\langle \text{SORTING ALGORITHM} \rangle$ or a part thereof. However, we impose two restrictions on the concepts and their usage in our model: Firstly, to be able to address them concepts need to have names (i. e. we deal only with lexicalized concepts). Secondly, we refer to a concept only through its name (and not for example through a formula, ID or prose text).

In a perfect world there would be a bijective mapping between names and concepts. Unfortunately in the natural languages names can be synonymous and polysemous. Therefore every concept has one or more intrinsic names. We assume that intrinsic names are normalized. The relation between intrinsic names and concepts is defined as:

$$\mathcal{N} \subseteq \|\mathcal{N}\| \times \mathcal{C}$$

The set of names of a concept c is defined as:

$$N_c = \{n \in \|\mathcal{N}\| : (n, c) \in \mathcal{N}\}$$

The set of concepts referred to by the name $n \in N$ is defined as:

$$C_n = \{c \in \mathcal{C} : (\|n\|, c) \in \mathcal{N}\}$$

Two names n_1, n_2 can be synonymous: $C_{n_1} \cap C_{n_2} \neq \emptyset$, and a name n can be polysemous: $|C_n| > 1$.

Lexical Layer The lexical layer contains all the normalized versions of the names used in the program. These names form a subset of $\|\mathcal{N}\|$.

Program Layer The program layer comprises the named program elements which appear in the program. These program elements constitute set P . As pointed out before, the set P holds no direct relation to the concept set \mathcal{C} but a relation to set of names N .

The relation between names and their corresponding program elements is defined as:

$$\mathcal{I} \subseteq \|\mathcal{N}\| \times P$$

The set of program elements named n is defined as:

$$P_n = \{p \in P : (\|n\|, p) \in \mathcal{I}\}$$

The name of a program element is defined as a set which contains only one element:

$$N_p = \{n \in N : (\|n\|, p) \in \mathcal{I}; |N_p| = 1\}$$

From Concepts to Program Elements Unfortunately the direct relations between concepts and program elements can not be established using the sets defined above due to polysemy and synonymy. In Section 4 we propose a semi-automatic technique to recover these relations from source

code. For the purpose of our formal discussion of defects we assume that these relations are established. With P_c we denote the set of program elements implementing concept c and with C_p the set of concepts implemented by program element p .

3.2. Defects Description

Using these sets we can now categorize the different types of relations between concepts, names and program elements. This illustrated in Figure 4 and detailed in the following paragraphs.

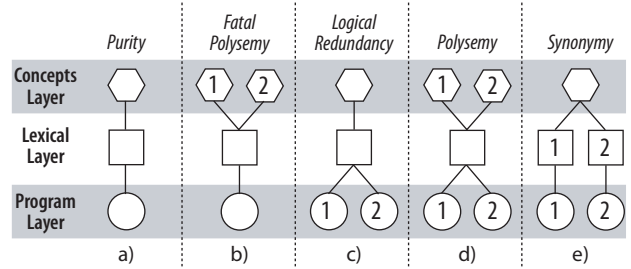


Figure 4. Types of Relations

Definition: Purity (a) A relation between a concept c , a name n and a program element p is *pure* iff

$$P_c = \{p\} \wedge C_p = \{c\} \wedge N_p = \{n\} \wedge n \in N_c$$

This is the ideal case. A concept is referred through a name and exactly one program element implements it.

Definition: Fatal Polysemy (b) The relation between two concepts c_1, c_2 , a name n and a program element p exhibits *fatal polysemy* iff

$$P_{c_1} = P_{c_2} = \{p\} \wedge C_p = \{c_1, c_2\} \wedge N_p = \{n\} \wedge n \in N_{c_1} \cap N_{c_2}$$

One program element is used to implement two concepts which have the same names. In this case the overloading of names through polysemy is combined with the overloading of the meaning of program elements.

A degenerate case of fatal polysemy is the *north* attribute (lower case) in the *NORTH-north* defect. The attribute actually refers to a component placed in north area of layout but its JavaDoc comment describes it to be a constant indicating the layout position. Thus, this program element is associated with two different meanings.

Definition: Logical Redundancy (c) The relation between two program elements p_1, p_2 , a name n and a concept c exhibits *logical redundancy* iff

$$P_c = \{p_1, p_2\} \wedge C_{p_1} = C_{p_2} = \{c\} \wedge N_{p_1} = N_{p_2} = \{n\} \wedge n \in N_c$$

We consider the logical redundancy as multiple places in the program where a concept is implemented by entities with the same name. An example of the pure redundancy problem is the *Days-of-the-Week* defect where the days are implemented exactly the same. Logical redundancy can be confusing (e. g. *Confusing-West* defect) or even dangerous (e. g. *January-January* defect).

Definition: Polysemy (d) The relation between two program elements p_1, p_2 , a name n and two concepts c_1, c_2 is *polysemous* iff

$$P_{c_1} = \{p_1\} \wedge P_{c_2} = \{p_2\} \wedge C_{p_1} = \{c_1\} \wedge C_{p_2} = \{c_2\} \\ \wedge N_{p_1} = N_{p_2} = \{n\} \wedge n \in N_{c_1} \cap N_{c_2}$$

This represents the case in which two distinct program elements with the same name implement two different concepts. The *NORTH-north* defect presented in Section 2 is an example for polysemy as the names of the two different attributes differ only in capitalization and these attributes implement different concepts.

Definition: (Synonymy e) The relation between two program elements p_1, p_2 , two names n_1, n_2 and a concept c is *synonymous* iff

$$P_c = \{p_1, p_2\} \wedge C_{p_1} = C_{p_2} = \{c\} \wedge N_{p_1} = \{n_1\} \\ \wedge N_{p_2} = \{n_2\} \wedge \{n_1, n_2\} \subseteq N_c$$

A concept is implemented in multiple places and denoted through different names. In comparison to the redundancy case, we require here that the program entities that implement the concept to have different names. Our example in this case is given by the *Ellipse-Oval* defect where the $\langle \text{ELLIPSE} \rangle$ concept is implemented both as a class and in a method and is referred in the program through two names. This represents a special case of redundancy as the same concept $\langle \text{ELLIPSE} \rangle$ is implemented two times at different levels of abstraction.

3.3. Compound Identifiers

We have presented our formal model starting from the assumption that every identifier in the program can be treated as an atomic string which represents (at least) one concept at the conceptual level.

In practice not all identifier names can be identified per se as names of concepts at the conceptual level. In a more realistic case we could expect that a program entity has compound name $[n_1, \dots, n_k]$ whose atomic parts are n_1, \dots, n_k . To each of the atomic part of the compound corresponds one or more concepts, thus the compound name $[n_1, \dots, n_k]$ denotes the set of concepts $\{c_1, \dots, c_l\}$. For example to the compound ‘drawOval’, which has the normalized parts [draw, oval], corresponds the concepts $\{\langle \text{DRAW} \rangle,$

$\langle \text{OVAL} \rangle\}$ (Figure 5). In this case, instead of a one-to-one correspondence between an identifier and a concept we have a one-to- N correspondence. Said with other words, to define the meaning of a compound identifier we use a set of concepts which approximate it.

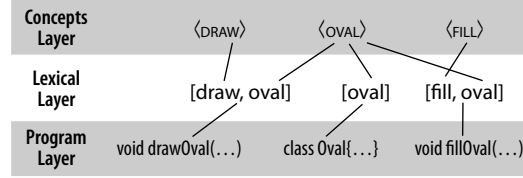


Figure 5. Compound Names

4. From Programs to Concepts

In the center of our formal framework for describing the defects is the assumption that by using the program entity names as glues we identify explicit relations between the program and conceptual layers. One could easily imagine that these relations could be created manually during a review. However, in order to make this approach scalable, we need automatic support in this endeavor.

In this section we present our semi-automatic method for bridging the conceptual and program layers. Firstly, we consider that the conceptual layer is described through entities from an ontology and relations between them. Secondly, we regard programs themselves as knowledge bases built on the identifiers and the relations between their corresponding program elements. The entities and relations of these two layers form two graphs and thus the identification of concepts can be reduced to graphs matching [24].

4.1. 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 [10]. An ontology is defined to be an *explicit specification of a conceptualization* [11] 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 of the specification detail which ranges from simple controlled vocabularies, to the specification of concepts through relations and arbitrary logical constraints [20].

In the present work we use an informal meaning of the term ‘‘ontology’’ - which we regard to comprise only concepts and relations between them. In order to represent

an ontology we use a graph language similar to the RDF graphs [13]. Entities within the ontology are the nodes of the graph. Relations between them are represented as labeled arcs. Figure 7 illustrates a fragment from the WordNet ontology that describes the <CALENDAR> concept as graph (see Figure 6).

The WordNet Ontology WordNet² is an online dictionary of English inspired by psycholinguistic theories of human lexical memory. Instead of organizing the words in terms of their form, like the majority of other dictionaries do, WordNet organizes the words in function of their meanings, in sets of synonyms (synsets) [21]. The WordNet 2.0 contains over 150,000 words grouped in more than 115,000 sets of synonyms, 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. WordNet defines two different types of relations between the concepts:

Hypernymy/Hyponymy (Generalization) The synsets are organized hierarchically along the hyponymy/hypernymy (i. e. “is a kind of”) 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.

Based on that we derive the *hyponymy equivalence*-relation that holds between all nouns that share a common hypernym.

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.

Based on that we derive the *meronymy equivalence*-relation that holds between all nouns that share a common holonym.

Figure 6 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>. Figure 7 shows the graph containing these concepts and relations alongside the derived equivalence relations.

²<http://wordnet.princeton.edu>

S: (n) **calendar** (a system of timekeeping that defines the beginning and length...)
direct hyponym
 S: (n) **solar calendar** (a calendar based on solar cycles)
direct hyponym
 S: (n) **Julian calendar, Old Style calendar** (the solar calendar introduced...)
 S: (n) **Gregorian calendar, New Style calendar** (the solar calendar now in general use, introduced by Gregory XIII in 1582 to correct an error in the Julian...)
part meronym
 S: (n) **January, Jan** (the first month of the year; begins 10 days...)
 S: (n) **February, Feb** (the month following January and...)
 ...

Figure 6. Example WordNet Entries

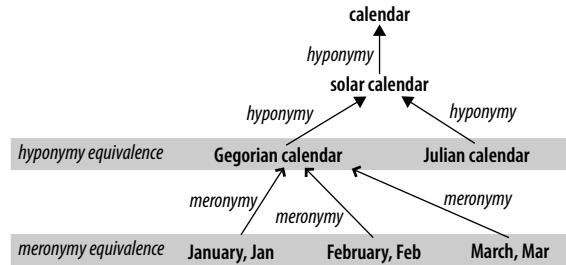


Figure 7. WordNet Relations as Graph

4.2. Representing Programs as Graphs

We combine the program and the lexical layers into a graph-based representation. The nodes of the program graph are the names from the lexical layer and the arcs are the relations among their corresponding program elements. We use two categories of *program relations*: based on the module system and based on the type system. The relations from the first category model the structural decomposition of the program domain and are similar to meronymy. The relations from the second category model the generalization hierarchies from program domain and are similar to hypernymy.

The relations that we used to recover the mappings between the code and the concepts in order to detect the defects presented in Section 2 are based on a set of heuristics. These heuristics are based on our empirical experience gathered in several case-studies, object oriented modeling practices [5] and basic linguistic principles [22]:

1. If two concepts in a generalization relation are implemented as types then the inheritance hierarchy reflects this relation.
2. If two concepts sharing a common superordinate are implemented in a restricted program scope as variables, then these variables have the same type.
3. If two concepts that share the same container in an aggregation relation are implemented as attributes then these attributes belong to the same class.

- If two concepts having the same superordinate are implemented with two compound names which differ through only one pair of atomic names then the compound names have a similar words sequence (detailed below).

Based on these heuristics we define the following relations that are used to build the program graph. These relations reflect the relations defined within the WordNet ontology.

Subclass-Of The inheritance relation between two types corresponds to the hypernymy relation between two WordNet entries (Heuristic 1). For example, the types `Ellipse2D` and `RectangularShape` in Figure 9 are in the “subclass-of” relation.

Same-Type-Variable The relation between two variables that are defined together (i. e. attributes of a class, local variables or parameters of a method) and which have the same type corresponds to the *hyponymy equivalence* relation between two WordNet entries (Heuristic 2). For example, this relation holds for all the pairs of parameters of the constructor of the `Date` class. In Figure 10 we show the relation between the parameters `sec` and `min`.

Same-Class-Attribute The relation between two attributes contained in a class corresponds to the *meronymy equivalence* relation between two WordNet entries (Heuristic 3). For example this relation holds for all pairs of the attributes of the class `Calendar`. In Figure 8 we show the relation between the attributes `JANUARY` and `FEBRUARY` of class `Calendar`.

Similar-Class-Member The relation between the members of a class (i. e. attributes or methods) whose names are compound identifiers which differ through only one word on the same position corresponds to the “hyponymy equivalent” (Heuristic 4). Intuitively, two similar compound words which differ through only one atomic name express the same class of concepts. More precisely, if we denote with $[n_1, \dots, n_k]$ and $[n'_1, \dots, n'_j]$ the names of the two compound identifiers, then the following logical expression holds for the names of class members in the *similar-class-member* relation:

$$k = l \wedge \exists i. 1 \leq i \leq k \wedge n_i \neq n'_i \wedge \forall j. 1 \leq j \leq k \wedge i \neq j \wedge n_j = n'_j$$

Examples in this category are presented in Figure 9: Methods `drawOval()` and `drawArc()` of class `Graphics` have compound names that differ by only one word (i. e. ‘oval’ and respectively ‘arc’).

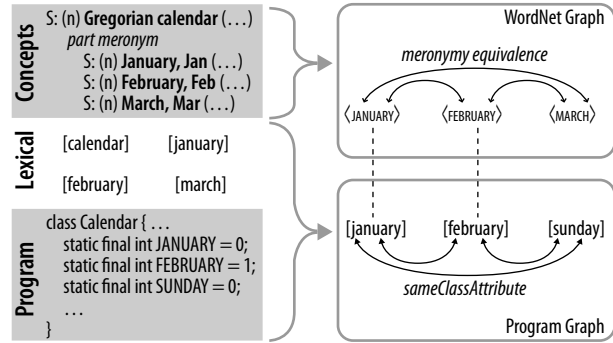


Figure 8. Concepts Identification Overview

4.3. Identifying Concepts

We identify concepts in a program by mapping the program and the ontology graphs. With every mapping we identify a possible occurrence of a concept in the sources. In Figure 8 we present examples of the mappings used for the identification of the `JANUARY` and `FEBRUARY` concepts. In this figure that `SUNDAY` constant couldn’t be mapped to a concept. The automatically extracted mappings are manually validated by a human user. Our bottom-up approach to automatically recover concepts from the code and link them with their implementation has the following steps:

- Step 1: Build the program graph.** In this step we abstract the program as a set of names and relations between them. This allows us to express the program as a labeled graph whose nodes are the names from the lexical layer. The lexical layer names are obtained by dividing the compound identifiers (e. g. based on their capitalization) into words and eliminating the variations that appear in these words (e. g. through stemming) in order to allow the further comparison with concept names. Two nodes are connected through a labeled arc if the corresponding program elements participate in one of the relations introduced above. The label indicates the type of the relation.
- Step 2: Concepts identification.** The purpose of this step is to extract concepts by finding similarities between the program and the WordNet graphs. Every identified mapping (illustrated as a dotted line in the figures) represents possible concepts in the program. In order to eliminate the false positives, we manually inspect the automatically obtained mappings. Figures 9 and 10 illustrate the identification of the concepts `ELLIPSE`, `OVAL`, `SHAPE`, `ARC` and respectively `SECOND`, `SEC`, `MINUTE`, `MIN`.

Once the location of a concept in the code is identified we can regard that part of code only from the perspective of

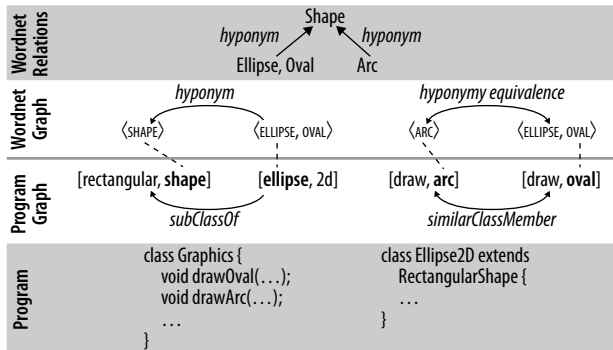


Figure 9. Identification of Shapes

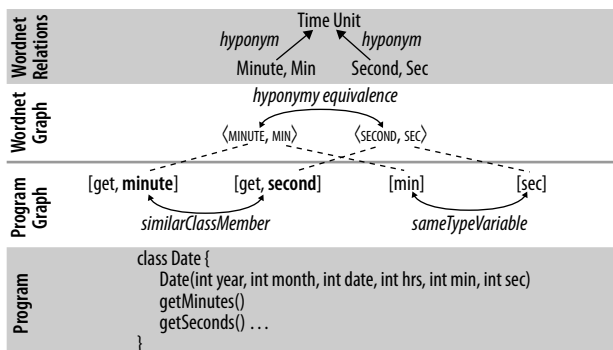


Figure 10. Identification of Time Units

the implemented concepts. This is the key step towards the detection of problems described in the next section.

5. Detecting Semantic Defects

The outcome of mapping concepts to program entities is a set of triples (p, n, c) where p is a program entity, n is an atomic name and c is the concept denoted by the name n . In the case of compound identifiers, their corresponding program element can appear in more triples - one triple for every identified concept.

The triples (p, n, c) express the relations needed by the formal model from Section 3.1. Thus, the next steps needed to identify the defects are straightforward:

Synonymy Find all triples (p, n, c) with the same concept and different names. The identification of the *Ellipse-Oval* and *Second-Sec-Minute-Min* defects was completely automatic. The inspection of the code and of its associated documentation proved indeed the existence of synonymy.

Polysemy Find all triples (p, n, c) with the same names and different concepts. The *NORTH-north* defect was detected semi-automatically as follows: the automatic steps identified the multiple occurrence of the concept $\langle \text{NORTH} \rangle$ at the levels of attributes of the class `BorderLayout`. The curiosity was that the concept was represented through a single name: 'north', but with different capitalization, very

similar with the $\langle \text{BLUE} \rangle$ concept in the *Blue-blue* defect. In the case of 'north' the manual inspection revealed the fact that only one of these occurrences denoted the position of a component in the `BorderLayout` and the other denotes the component itself which occupies this position. Furthermore, by inspecting the comment of the attribute `north` we noticed that it was itself associated with two concepts: one in its comment and the other in its usage.

Logical Redundancy Find all triples (p, n, c) with the same concept, same name and different program entities. The semantic equivalence of program entities was inspected by hand. For example, in the case of the *Days-of-the-Week* defect, the manual inspection revealed that the two implementations are fully equivalent. In the case of the *Confusing-West* defect and the *January-January* defect, the implementations proved to be not equivalent.

The semi-automatic detection of these defects is supported by the reverse engineering platform Insider³ and the concept extraction tool Bridge⁴.

6. Related Work

We divide the related work in two categories: work which deals with the defects that we identified in the code, namely naming problems and redundancies, and related methods for extracting the concepts from the source code.

Naming Problems The importance of identifier names is widely recognized in the literature to impact various facets of program understanding [1]. [4] and [23] highlight their relevance in the context of concept location. [9] elaborates on this by introducing a formal model that describes synonymy and polysemy. In this paper we advance on that by considering both naming deficiencies and logical duplications as consequences of skewed real word representation in programs as well as by providing an semi-automatic method for detecting these defects.

Code Redundancies The negative impact of redundancies in a program is widely recognized in the literature to have a negative impact on software maintenance [6]. However, most of the current approaches, e. g. [3, 14], focus on the detection of duplicated *code*. Our approach considers *logical* duplications which is at a higher abstraction layer. In comparison to the detection of redundancies through clones, our approach can detect multiple implementations of a concept in the sources even if it appears at different levels of abstraction - e. g. in the *Ellipse-Oval* defect the con-

³Insider is developed at LOOSE Research Group, "Politehnica" University from Timișoara, Romania (www.loose.upt.ro)

⁴Bridge is developed at Technische Universität München, Germany

cept of \langle ELLIPSE \rangle was found at two levels of abstraction: as a class and as a method which performs the drawing.

Concept Extraction We know of two other techniques for extracting high level semantically information from the source code: Formal concept analysis (FCA) is used to identify high-level dependencies in the code by finding groups of elements (called “concepts”) that have the same properties [2]. Latent Semantic Indexing (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 cohesion with other sets are named “concepts” [15].

The aforementioned 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. Thus, the concepts that appear seldom in the code are most likely to be ignored or treated as noise. Furthermore, in both cases the concepts are much coarser grained and are described by sets of words; the exact interpretation of concepts is a difficult issue left exclusively on the human user [2, 15].

[18] 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.

In [24] we presented a basic approach for recovering the mappings between the concepts from an ontology and the source code. In the current work we extend the set of program relations and we use the mappings to express the semantics of program elements in terms of the used ontology and thus to enable the (semi-)automatic detection of the semantic defects.

7. Conclusions and Future Work

Semantic defects like improper naming and logical duplication are known to have severe consequences for software maintenance as they hamper program comprehension and modification activities. We established a formal framework for pinpointing these kinds of defects based on an explicit mapping between the real world concepts, program elements and their names. Starting from this framework we presented our method for semi-automatic detection of the discussed defect classes. To validate our approach we gave examples of these defects found in the standard Java library by using our method.

Although the identification of these defect examples validates our approach, we are aware that in order to fully assess its practicability, there are many variation points that need to be investigated. As future steps we plan to perform an in-depth evaluation of our approach by applying it on more case-studies and by using domain specific ontologies.

References

- [1] N. Anquetil and T. C. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON '98*, 1998.
- [2] G. Arévalo, S. Ducasse, and O. Nierstrasz. Lessons learned in applying formal concept analysis. In *ICFCA '05*, volume 3403 of *LNAI*. Springer Verlag.
- [3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98*, 1998.
- [4] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE '93*, 1993.
- [5] G. Booch. *Object Oriented Analysis and Design with Applications*. The Benjamin Cummings Publishing Co. Inc., 2nd edition, 1994.
- [6] E. Burd and M. Munro. Investigating the maintenance implications of the replication of code. In *ICSM '97*, 1997.
- [7] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM '00*, 2000.
- [8] A. Cimitile, A. D. Lucia, G. A. D. Lucca, and A. R. Fasolino. Identifying objects in legacy systems. In *IWPC '97*, 1997.
- [9] F. Deissenboeck and M. Pizka. Concise and consistent naming. In *IWPC '05*, 2005.
- [10] M. R. Genesereth and N. J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.
- [11] 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.
- [12] C. S. Hartzman and C. F. Austin. Maintenance productivity. In *CASCON '93*, 1993.
- [13] P. E. Hayes. Rdf semantics. Technical report, W3C Recommendation, 2004.
- [14] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.
- [15] A. Kuhn, S. Ducasse, and T. Gırba. Enriching reverse engineering with semantic clustering. In *WCRE '05*, 2005.
- [16] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICMS '97*, 1997.
- [17] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *ICSE '01*, 2001.
- [18] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *ASE '01*, 2001.
- [19] A. Marcus, A. Sergeev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *WCRE '04*, 2004.
- [20] D. L. McGuinness. Ontologies come of age. In *Spinning the Semantic Web*, 2003.
- [21] G. Miller and et. al. Introduction to wordnet: an on-line lexical database. *International Journal of Lexicography*, 3:235–244, 1990.
- [22] W. O'Grady, M. Dobrovolsky, and F. Katamba, editors. *Contemporary Linguistics: An introduction*. Longman, London and New York, 3rd edition, 1996.
- [23] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *IWPC '02*, 2002.
- [24] D. Ratiu and F. Deissenboeck. Programs are knowledge bases. In *ICPC '06*. IEEE CS Press, 2006. to appear.