

The Reality of Libraries

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

Jan Jürjens
The Faculty of Mathematics and Computing
The Open University
Milton Keynes, Buckinghamshire
<http://www.jurjens.de/jan>

Abstract

Libraries provide implementation for the concepts of a particular domain. When programmers use a library, they do not work any more with the real-world concepts but with their implementation available within the used library. From this point of view, libraries represent the “reality” at which the programmers have to adapt. Ideally, in order to be easy to use, the implementation of concepts in libraries should accurately mirror the concepts and their relations from the real world. Unfortunately, this is not always realised and this results in a bias between the real-world concepts and their implementation. Depending on the kind of the bias, the users can adapt themselves or not to the library’s “reality”. In this paper we propose a method to describe and evaluate the bias of the library implementation of real-world concepts expressed within an ontology. We use our method to describe several primitive bias classes in a formal framework and to discuss how can they affect the library’s users. We present our results with the help of bias examples which we (semi-)automatically identified in the Java standard library.

1. Introduction

A library is a vocabulary designed to be added to a programming language to make the vocabulary of the programming language larger.

Growing a Language¹
Guy Steele

Libraries are the most widespread form of software reuse. On the one hand, they increase the abstraction level at which programs are written by providing ready-to-use implementations of the domain specific concepts. On the other hand, by using a library the programmers need to commit themselves to the particular implementation of the real world

concepts. Thus, libraries restrict their users and they represent the “reality” with which the programmers have to deal. In order to make use of a library the programmers need to think in terms of how were the real-world entities modeled in it. The bigger the difference between the library’s “reality” and the real world, the more the difficulties encountered by the library users are.

In the upper part of Figure 1 we present a well-known example of two possible implementations of the real-world concepts: rectangle and square². The implementations differ in the usage of inheritance: the first uses the type inheritance (Figure 1a) and the second uses class inheritance (Figure 1b). The library clients must regard the world of figures as it is defined by the library - when they do not do this, and regard the figures hierarchy as it is in the reality, their code can be wrong. For example, in the lower part of the Figure 1 are presented clients of this class hierarchy that compute the area of squares and rectangles. In the first case (Figure 1a) the implementation of clients works as one would expect: we need to compute the area only for rectangles and this functions automatically for squares. In the second case, however, (Figure 1b) we need to implement two times the functions that compute the area: when (1) is implemented alone then we can not compute the area for square objects and when (2) is implemented alone then it delivers bad results for rectangle objects. Thus, in Figure 1b we have an example of an easy to misuse library. This can not happen in the code from Figure 1a while a possible function `int area(Square)` can not be called for rectangle objects.

The library defined relations between its program entities from its public interface should be as close as possible to the real-world relations between the concepts implemented by them. Then, when using the library, the users can think in terms of the real-world concepts and not in terms of their particular implementation. Having achieved this, the library API will be in correspondence with the domain and thus easier to learn and to use and harder to misuse. These are

¹OOPSLA’98 invited talk

²In the Merriam-Webster Online Dictionary the definition of a square is: “a rectangle with all four sides equal”

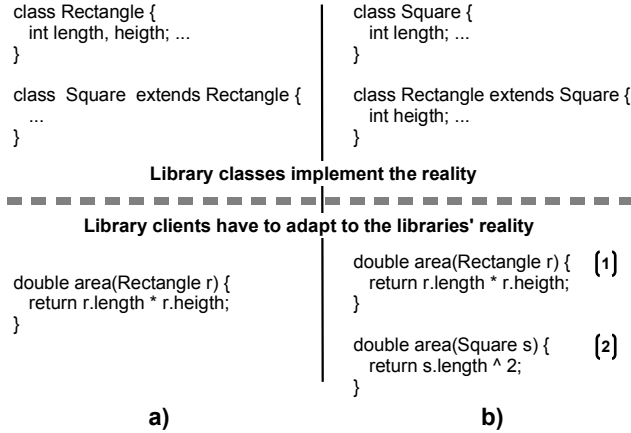


Figure 1. A library defines its “reality”

actually important quality attributes of libraries [7, 9, 1].

In this paper we present a formal framework which enables us to evaluate the implementation of the real-world concepts within a library. This framework is based on a common representation of ontologies (Section 2) and programs (Section 3). Using this framework we characterize general classes of mismatch between the concepts implementation and the real world (Section 4). In Section 5 we present relevant mismatches together with examples that we (semi-)automatically identified within the Java library. Our semi-automatic method for the biases identification, based on the mapping between the program elements and the ontology entities, is presented in Section 6. Section 7 presents the related work and Section 8 concludes the paper and gives hints about possible extensions.

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 [3]. An ontology is defined to be an *explicit specification of a conceptualization* [4] and is used for sharing the knowledge about a domain by making explicit the concepts and relations within it. 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.

The term “specification” implies that this conceptualization is defined in a rigorous manner. From the point of view of specification detail, ontologies may vary from controlled vocabularies to rich concept hierarchies with proper-

ties. Depending on their specification detail, there are different situations in which ontologies can be useful: from vocabulary control within a project up to ensuring the consistency and completion among the used terms [8].

In the present work we use a loose meaning of the word ontology. We define an ontology O to comprise a set of concepts (C) and a set of typed relations between these concepts (R^o):

$$O = (C, R^o), \quad C = \{c_1, \dots, c_m\}, \quad (1)$$

$$R^o = \{f^o_1, \dots, f^o_n\} \text{ with } f^o_i : C \rightarrow \mathcal{P}(C)$$

Each function f^o_i is used to represent a relation type and associates to each concept c a set of concepts to which it is related. This representation is similar to the RDF graphs [6]: entities within the ontology (i.e. elements of C) are the nodes of the graph and the typed relations between them (i.e. members of R^o) are labeled arcs. This view over ontologies is however rich enough for the evaluation of the consistency and completion of the implementations of concepts within libraries (Section 4).

In the following we give an example of an ontology which shares a large number of concepts lexicalized in English. Subsequently, we use this ontology to identify mismatches between the real world concepts and their implementation in the Java library.

The WordNet Ontology. WordNet³ 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 in sets of synonyms (synsets), according to the meaning of the concepts they denote [10]. WordNet 2.0 contains over 150,000 words, of which more than 70% are nouns, grouped in more than 115,000 sets of synonyms. 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.

Figure 2a shows an example of how WordNet represents the calendar concept. We notice four hyponymy relations

³<http://wordnet.princeton.edu>

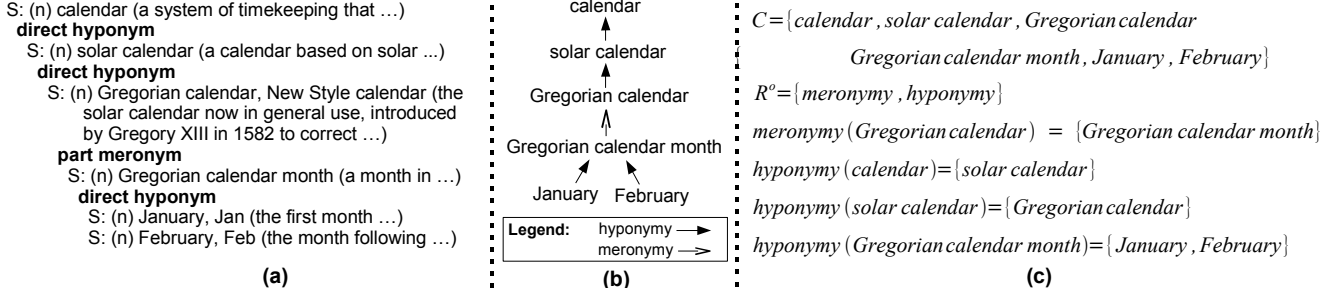


Figure 2. Example of WordNet entries (a); Graph-based Representation (b); Formalization (c)

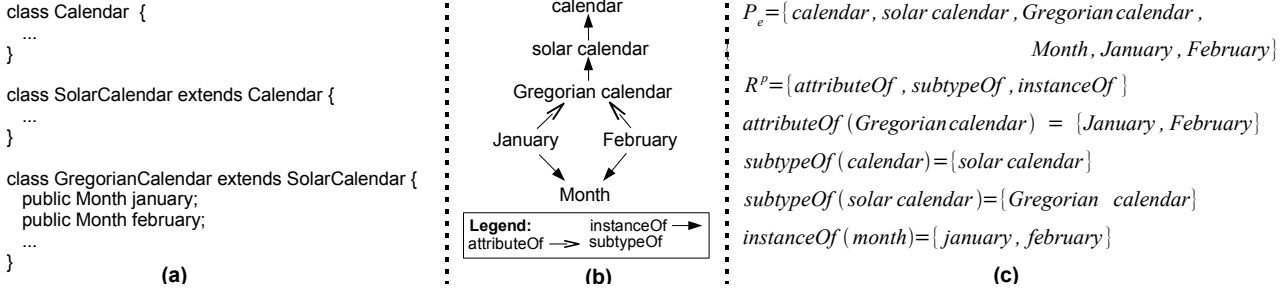


Figure 3. Example of a program (a); Graph-based representation (b); Formalization (c)

in the calendar hierarchy (e. g. solar calendar is-a calendar) and a meronymy relation (e. g. Gregorian Calendar month is part-of Gregorian calendar). In Figure 2b is the graph-based description of these concepts and in Figure 2c we give an example of how does our formalization (1) express these concepts and relations between them.

3. Libraries implement the domain knowledge

Libraries implement the real-world knowledge and share it through their public interface. We abstract programs as labeled graphs whose nodes are the program elements and whose arcs are a (sub-)set of the relations which the programming language define between these program elements [12, 11]. In the case of the libraries we are interested in the interface that they provide to their users and thus, we restrict the set of interesting programming elements to those that are public (e.g. public classes, attributes) and ignore all the others (e.g. local variables).

In a similar manner with the representation of ontologies (1), we formally define a program P to comprise a set of named program elements (Pe) and a set of typed relations between them (R^p):

$$P = (Pe, R^p), \quad Pe = \{p_1, \dots, p_m\}, \quad (2)$$

$$R^p = \{f^p_1, \dots, f^p_n\} \text{ with } f^p_i : Pe \rightarrow \mathcal{P}(Pe)$$

Each function f^p_i is used to describe a relation type which associates to each program element p a set of program elements to which it is related.

In Figure 3a we present a possible implementation of the calendar related concepts illustrated in Figure 2. Every element from this ontology has a corresponding program element which implements it. Each program element which implements a concept has the same name as the concept itself. In Figure 3b the abstract view over the program is presented as a graph whose nodes are program elements identifiers and the arcs are relations between their corresponding program elements. An example of a formal description of the code fragment is presented in Figure 3c. The program elements considered are the classes and their attributes and the program relations are those generated by the type system (i.e. subtypeOf, instanceOf) and the module system (i.e. attributeOf).

4. How programs represent the reality

In the previous two sections we defined an abstract representation of both programs and ontologies as labeled graphs. To formally link an ontology (O) to a program (P) we define in the following the functions I and τ .

Let $I : C \rightarrow Pe \cup \{\epsilon\}$ be a function which maps the entities from an ontology to the corresponding program elements which implement them. When a concept c is not implemented in the code then $I(c) = \epsilon$. If $c \in C$ is a concept, then $I[f^o_i(c)] = \{I(c') \mid c' \in f^o_i(c)\}$, i.e. $I[f^o_i(c)]$ denotes the set of program elements implementing all those concepts that are related via f^o_i to c .

Let $\tau : R^o \rightarrow \mathcal{P}(R^p)$ be a function which maps the relation types between the concepts to similar relation types

between program elements. If $p \in Pe$ is a program element and f^o_i is a conceptual level relation type then $[\tau(f^o_i)](p) = \{\cup f^p_k(p) \mid f^p_k \in \tau(f^o_i)\}$, i.e. $[\tau(f^o_i)](p)$ denotes the set of program elements related with p via one of the program level relations similar to f^o_i .

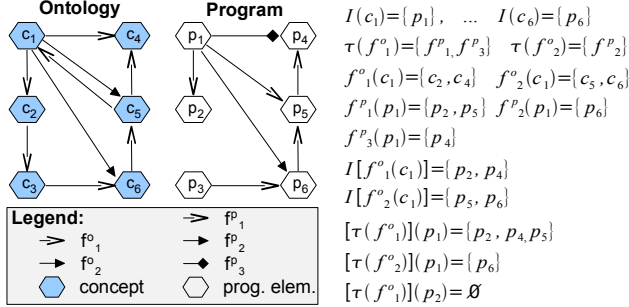


Figure 4. Formalization Example

In Figure 4 we present an example of what these formula mean. A concrete instance of the function τ in the context of the calendar example from Figure 2 and Figure 3 is to map the hyponymy to the type-system generated relations (e.g. $\tau(hyponymy) = \{subtypeOf, instanceOf\}$) and the meronymy to the relations generated by the module system (e.g. $\tau(meronymy) = \{attributeOf\}$).

Having defined the ontology and the program as graphs, and the relations between them through the functions I and τ , we will use principles from the graph-theory to characterize the relations between concepts and their related implementation.

Definition The implementation I of a concept c and relation f^o_i is *ideal* iff:

$$I[f^o_i(c)] = [\tau(f^o_i)](I(c)) \quad (3)$$

The *ideal* implementation represents the case in which there is a one-to-one correspondence between the entities and relations from the ontology and the entities and relations from the library (Figure 5a). Mathematically, the function I is an isomorphism⁴ and the conceptual and program worlds are completely indistinguishable as far as our chosen representations (O and P) are concerned.

Discussion: the *ideal* implementation can not be realised due to the difference between the high-level declarative representation of the real world knowledge (i.e. ontologies) that we have chosen and the low-level operational programming constructs. In order to bridge this gap, the programs contain many *implementation details*. Furthermore, an ideal implementation is not wanted in practice, due to

⁴Isomorphism is: a one-to-one correspondence between the elements of two sets such that the result of an operation on elements of one set corresponds to the result of the analogous operation on their images in the other set.

the high complexity of the modeled domain and the limited resources of the programmers. This leads to the implementation of only an *abstraction* of the world.

Definition The implementation I of a concept c and relation f^o_i is an *abstraction* iff:

$$I[f^o_i(c)] \supset [\tau(f^o_i)](I(c)) \quad (4)$$

The *abstraction* implementation represents the case in which a part of the concepts related to c or the relations between them are not reflected at the code level (Figure 5b).

Discussion: the implementation exhibits *abstraction* due to the pragmatic decisions that the programmers made when a system was planned and due to the way in which the system's boundaries were chosen. Depending on what was left out from the modeled domain during the abstraction process, and on the relations between these entities with the ones which have been already implemented, it can be quite difficult or even impossible to use, extend or adapt the library. In Section 5 we discuss several abstraction cases and how they influence the library's extension by answering to the question: What concepts from the ontology, related to the concepts that were already implemented in the code, are not represented in the library?

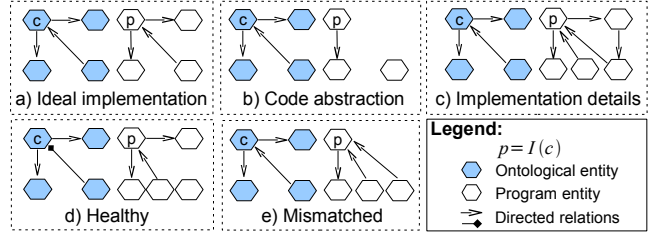


Figure 5. Reality of programs - An overview

Definition The implementation I of a concept c and relation f^o_i introduces *implementation details* iff:

$$I[f^o_i(c)] \subset [\tau(f^o_i)](I(c)) \quad (5)$$

The *implementation details* represent the case in which additional to the program elements and relations corresponding to the implemented concepts, other program elements and relations appear on the program side (Figure 5c).

Discussion: the *implementation details* are inherent due to the discrepancy between the declarative representations of the real-world knowledge and the operational implementation supported by the current languages. By introducing the implementation details that do not represent concepts from the modeled domain, the library loses its conciseness. Its vocabulary becomes cluttered with implementation related words and thus hard to learn, understand and use. In Section 5 we discuss several typical cases of implementation details and how they influence the users.

Definition The implementation I of two concepts c_1 and c_2 and a relation f^o_i is *healthy* iff:

$$I(c_1) = p_1 \wedge I(c_2) = p_2 \wedge p_2 \in f^p_k(p_1) \Rightarrow \exists f^o_i \in R^o . c_2 \in f^o_i(c_1) \wedge f^p_k \in \tau(f^o_i) \quad (6)$$

The *healthy* implementation does not deform the reality: if the implementations of two concepts are related then the relation is similar with the ontology relation between these concepts (Figure 5d).

Discussion: the *healthy* implementation can be both an abstraction and generate implementation details but does not deform the reality. A *healthy* implementation preserves the structure of the real-world in the program. As long as the structure of the implemented concepts is preserved, it can be an *abstraction* (i.e. not all concepts or their relations are implemented) or *implementation details* (i.e. other program elements and relations appear in the code) or both.

Definition The implementation I of a concept c and a relation f^o_i is *mismatched* iff:

$$I[f^o_i(c)] \setminus [\tau(f^o_i)](I(c)) \neq \emptyset \wedge [\tau(f^o_i)](I(c)) \setminus I[f^o_i(c)] \neq \emptyset \quad (7)$$

The *mismatched* implementation represents a combined situation when the implementation of a concept c in a program generates implementation details and not all the concepts related to c are implemented. (Figure 5e)

Discussion: the mismatched implementation is the most general case, comprising both *abstraction* and *implementation details* without any *healthy* conditions. In most of the cases, the implementation of concepts in the code exhibits the general *mismatch*.

5. Mismatches classification

In this section we present four classes of primitive mismatches between the real-world concepts and their implementation in programs (Sections 5.1 - 5.4). For each bias class we describe its variations by using the following structure: At first we formally define each mismatch, then we present an intuitive description of the bias, classify the bias according to the criteria from the previous section, discuss how does the bias affect the library users and last but not least present a real-world example which we (semi-)automatically detected in the Java library. We illustrate these biases intuitively in Figures 6 and 7. By using and discussing the real-world examples of these biases from a wide-spread library, instead of presenting hand-crafted pieces of code, we want to point out their widespread and their potential impact on the library’s clients.

Apart from the *equivalent* (i.e. non-bias) cases, all the others represent a form of reality mismatch and thus violate the domain correspondence quality attribute of libraries [2].

In the next of this paper we commit to the following typographical conventions: all program elements are written with typewriter fonts and all the concepts with SMALL CAPS.

5.1. Implementation of Relations

In the following we present a classification of the mismatches generated by different implementations of real-world relations in programs (Figure 6a). Through this classification we aim to answer the question: How is a directed relation between two concepts from the real-world reflected at the code level?

Definition The implementation I of the relation f^o_i between the concepts c_1 and c_2 is *equivalent* iff:

$$I(c_1) = p_1 \wedge I(c_2) = p_2 \wedge c_2 \in f^o_i(c_1) \wedge \wedge p_2 \in [\tau(f^o_i)](p_1)$$

This represents the ideal case where if two related concepts are implemented in the code then, the relation between their implementation reflects the relation from the real world.

Definition The implementation I of the relation f^o_i between the concepts c_1 and c_2 , is *inverted* iff:

$$I(c_1) = p_1 \wedge I(c_2) = p_2 \wedge c_2 \in f^o_i(c_1) \wedge \wedge p_1 \in [\tau(f^o_i)](p_2)$$

In this case, a directed relation between two concepts is implemented in the program through a similar relation but in the inverse sense. This case represents a form of a non-healthy mismatched implementation which can lead to a non-intuitive usage of the library.

Example: in the `java.util` package, the class `LinkedList` implements the interface `Queue` and thus generates a program relation between these classes similar to the hyponymy relation between concepts. This is in opposition with the real-world perception that a `QUEUE` is a special kind of a `LIST` (Figure 8a). The consequence of the program relation is that whenever a `Queue` object is needed, we can use a `LinkedList` object. The later can be easily used in a way that violates the `Queue` constraints. Below is an usage example of the `LinkedList` implementation for a `Queue` that breaks the ordering of elements within the queue: the `server_writeMessage` function writes always at the same end from which the client reads. The client will read the elements in an opposite order and thus our “queue” functions as a stack.

```
Queue<Message> aQueue = new LinkedList<Message>();
void server_writeMessage() {
    ((LinkedList)aQueue).addFirst(new Message()); ...
}
...
Message client_readMessage(Queue aQueue) {
    return aQueue.element();
}
```

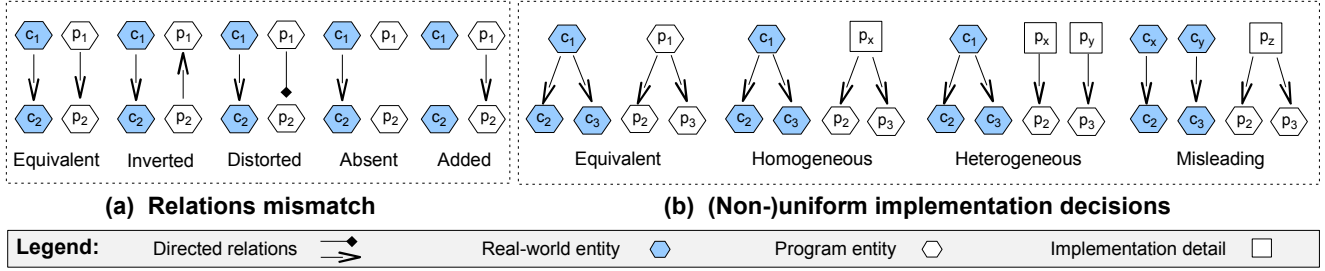


Figure 6. Relations mismatch and (non-)uniform implementations

Definition The implementation I of the relation f_i^o between the concepts c_1 and c_2 , is *distorted* iff:

$$I(c_1) = p_1 \wedge I(c_2) = p_2 \wedge c_2 \in f_i^o(c_1) \wedge \wedge f_k^p \notin \tau(f_i^o) \wedge p_2 \in f_k^p(p_1)$$

A relation between two concepts f_i^o is implemented in the code through a relation f_k^p with which it is not similar. This case represents a form of a non-healthy mismatch since another relation is used instead of the one from the real world.

Example: in the `java.awt` package the BOLD FONT concept is implemented through the attribute BOLD of the class `Font`. This is contrary to the real-world definition of BOLD FONTS given in WordNet according to which BOLD is-a FONT. (Figure 8b). Due to this mismatch, the extension of the library with new general font styles (e.g. UNDERLINE) is impossible and would require the modification of the `Font` class. Adding more FONT types in this class will cause it to grow and will clutter its interface.

Definition The implementation I of the relation f_i^o between the concepts c_1 and c_2 exhibits *absent relation* iff:

$$I(c_1) = p_1 \wedge I(c_2) = p_2 \wedge c_2 \in f_i^o(c_1) \wedge \nexists f_k^p \in R^p. p_1 \in f_k^p(p_2) \vee p_2 \in f_k^p(p_1)$$

In this case two related concepts are implemented in the code but the implementation does not reflect any relation between their implementations. This case represents a form of healthy abstraction.

Example: in the `java.util` package both the MONTH and the JANUARY concepts are implemented as static constants of the class `Calendar` (Figure 8c). Thus, the hyponymy relation between the concepts MONTH and JANUARY from the WordNet ontology is not reflected through any of our chosen code relations (i.e. `subtypeOf`, `instanceOf`, `attributeOf`).

Definition The implementation I of two unrelated concepts c_1 and c_2 exhibits *added relation* iff:

$$I(c_1) = p_1 \wedge I(c_2) = p_2 \wedge \nexists f_i^o \in R^o. (c_1 \in f_i^o(c_2) \vee c_2 \in f_i^o(c_1)) \wedge \exists f_k^p \in R^p. p_2 \in f_k^p(p_1)$$

In this case, there are relations at the code level between program elements which are not present between their corresponding concepts. This case represents an instance of non-healthy implementation decisions. Through the added relation we define unnecessary constraints at the level of the code between the concepts implementation and this can make the library harder to use.

Example: in the `java.util` package the `GregorianCalendar` class contains the attribute `second` (Figure 8d) even these two concepts are independent in the WordNet ontology.

5.2. Implementation of similar concepts

Definition We call two concepts c_2 and c_3 *similar* iff $\exists c_1 \in C \wedge f_i^o \in R^o. c_2 \in f_i^o(c_1) \wedge c_3 \in f_i^o(c_1)$.

Intuitively, two concepts are similar when there exists another concept to which they are related through the same relation type. Analogous, two program elements p_2 and p_3 are *similar* when there exists another program element p_1 to which they are related through the same relation type.

In this section we present a classification of the mismatches generated by reflecting the concepts similarity in programs (Figure 6b). Through this classification we aim to answer to the question: How is the similarity between concepts from an ontology reflected at the code level?

Definition The implementation I of the similar concepts c_2 and c_3 and relation f_i^o is *equivalent* iff:

$$\{c_2, c_3\} \subset f_i^o(c_1) \wedge p_1 = I(c_1) \wedge p_2 = I(c_2) \wedge p_3 = I(c_3) \wedge \exists f_k^p \in R^p. f_k^p \in \tau(f_i^o) \wedge \{p_2, p_3\} \subset f_k^p(p_1)$$

This is the ideal case in which both the concepts c_1 , c_2 , c_3 and their similarity relation are reflected in the code.

Definition The implementation I of the similar concepts c_2 and c_3 and relation f_i^o exhibits *homogeneity* iff:

$$\{c_2, c_3\} \subset f_i^o(c_1) \wedge p_2 = I(c_2) \wedge p_3 = I(c_3) \wedge \exists p_x \in Pe, f_k^p \in R^p. f_k^p \in \tau(f_i^o) \wedge \{p_2, p_3\} \subset f_k^p(p_x)$$

Intuitively, this represents the situation when similar concepts from the modeled domain are implemented in the library through similar program elements. By not implementing the concept c_1 we lose the precision. By having a similar implementation we ensure the uniformity of the library and thus improve its usability. This represents a case of a general mismatch.

Example: all months are uniformly implemented in the Java library as integer constants of the Calendar class from the `java.util` package (Figure 8e).

Definition The implementation I of the similar concepts c_2 and c_3 and relation f^o_i exhibits *heterogeneity* iff:

$$\{c_2, c_3\} \subset f^o_i(c_1) \wedge p_2 = I(c_2) \wedge p_3 = I(c_3) \wedge \\ \nexists p \in Pe, f^p_k \in R^p. f^p_k \in \tau(f^o_i) \wedge \\ \{p_2, p_3\} \subset f^p_k(p)$$

In this case the similarity between concepts from the modeled domain is not reflected by their corresponding implementations. This is an example of a violation of the uniformity quality attribute of libraries. This case represents an instance of a general mismatch.

Example: the CARDINAL POINTS in the `java.swing` package are implemented both as `String` constants of the class `SpringLayout` and as integer constants of the class `SwingConstants` (Figure 8f). Due to their different implementations, the library clients have to use them differently and to write supplementary code to perform conversions between these representations.

Definition The implementation I of two non-similar concepts c_2 and c_3 is *misleading* iff:

$$p_2 = I(c_2) \wedge p_3 = I(c_3) \wedge \\ \nexists c \in C, f^o_i \in R^o. \{c_2, c_3\} \subset f^o_i(c) \wedge \\ \exists p_z \in Pe, f^p_k \in R^p. \{p_1, p_2\} \subset f^p_k(p_z)$$

Intuitively, for the implementation of two different concepts are taken the same implementation decisions. By making the program elements p_2 and p_3 similar we define de-facto relations between them. This case appears often when p_z is a general purpose implementation stub. This represents a general mismatch.

Example: A classical example is that the class `java.lang.Object` is used as parent for many classes which are implementations of unrelated concepts.

5.3. Multiple Concepts Representation

In this section we present a classification of the mismatches generated by the implementation of sequences of concepts (Figure 7a). Through this classification we aim to answer the question: How are sequences of concepts and relations from the ontology reflected at the code level?

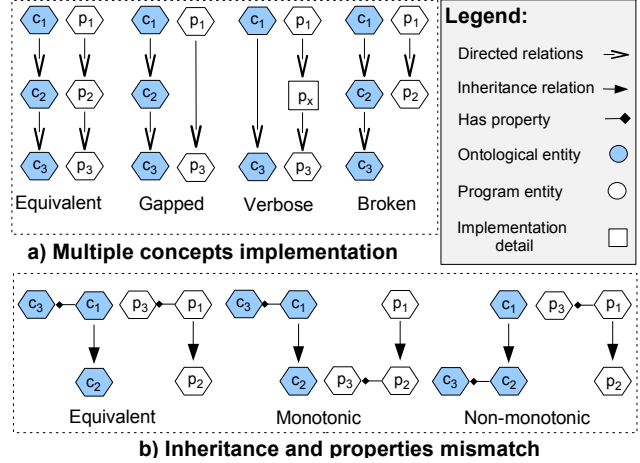


Figure 7. Special mismatches

Definition The implementation I of the concepts c_1, c_2 and c_3 related through the relation f^o_i is *equivalent* iff:

$$c_2 \in f^o_i(c_1) \wedge c_3 \in f^o_i(c_2) \wedge \\ I(c_1) = p_1 \wedge I(c_2) = p_2 \wedge I(c_3) = p_3 \wedge \\ p_2 \in [\tau(f^o_i)](p_1) \wedge p_3 \in [\tau(f^o_i)](p_2)$$

This is the ideal case in which all concepts and relations between them are accurately implemented in the code.

Definition The implementation I of the concepts c_1, c_2 and c_3 related through the relation f^o_i is *gapped* iff:

$$c_2 \in f^o_i(c_1) \wedge c_3 \in f^o_i(c_2) \wedge \\ I(c_1) = p_1 \wedge I(c_3) = p_3 \wedge p_3 \in [\tau(f^o_i)](p_1)$$

In this situation, from a sequence of three concepts, the middle concept is left out by the implementation. The *gapped implementation* appears many times naturally when the implemented relation is transitive - and then it is an instance of healthy abstraction. When the relation is non-transitive then the implementation is a non-healthy abstraction. Most of the times it is impossible for a client to extend the library with the concept left out by a *gapped* implementation since this requires modifications of the library relations.

Example: the implementation of the CALENDAR related concepts in the `java.util` package is done through classes `Calendar` and `GregorianCalendar` (Figure 8g). This implementation is *gapped*, as the SOLAR CALENDAR concept defined in WordNet between CALENDAR and GREGORIAN CALENDAR is not implemented in the library. Suppose now that a library client would need the SOLAR CALENDAR concept. He/She would have to either modify the library (which in this case is impossible), commit to another library or re-implement the calendar hierarchy. We believe that neither of these options is appealing.

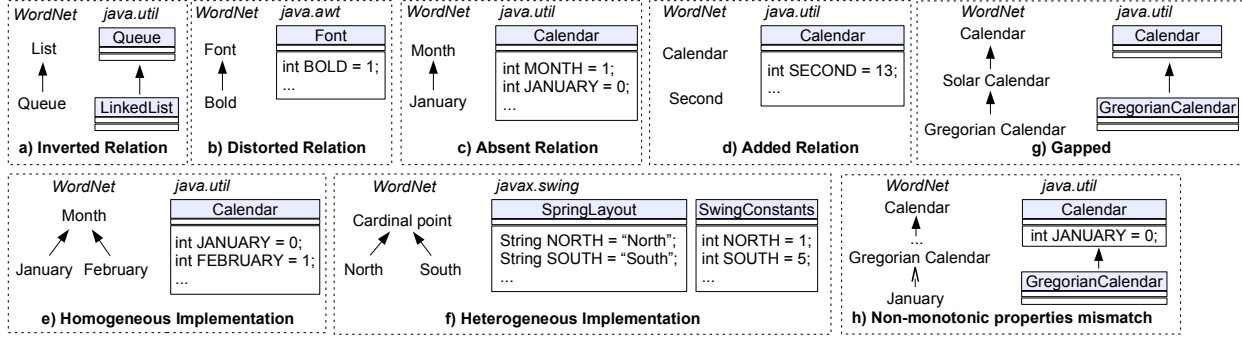


Figure 8. Mismatch examples

Definition The implementation I of the concepts c_1 and c_3 related through the relation f^o_i exhibits *verbosity* iff:

$$c_3 \in f^o_i(c_1) \wedge I(c_1) = p_1 \wedge I(c_3) = p_3 \wedge \\ \exists p_x \in P_e, f^p_k \in R^p, f^p_l \in R^p. \\ p_x \in f^p_k(p_1) \wedge p_3 \in f^p_l(p_x)$$

In this case the relation between two concepts is implemented via an additional program element situated in-between their corresponding program elements. This additional program element corresponds to implementation (e.g. design) decisions. This case represents an instance of a healthy mismatch.

Definition The implementation I of the concepts c_1 , c_2 and c_3 related through the relation f^o_i is *broken* iff:

$$c_2 \in f^o_i(c_1) \wedge c_3 \in f^o_i(c_2) \wedge I(c_1) = p_1 \wedge \\ I(c_2) = p_2 \wedge I(c_3) = \epsilon \wedge \\ \exists f^p_k \in R^p. f^p_k \in \tau(f^o_i) \wedge p_2 \in f^p_k(p_1)$$

In this case we have a partial implementation of the concepts sequence in which a concept is left out. Depending on the relation type, the library can or can't be extended with the c_3 concept. In the case of concepts implemented in an inheritance hierarchy, if c_3 is a hyponym of c_2 then the library can be easily extended by its clients (i.e. create a subtype of p_2). However, if the c_3 is a hypernym of c_2 then the library can not be extended by its clients. This case corresponds to the *healthy abstraction*.

5.4. Implementation of inherited properties

Below we present a classification of mismatches generated by different implementations of the properties in an inheritance hierarchy. We denote the inheritance relation at the conceptual level through “hyponymy” and at the code level through “subtypeOf”.

Definition The implementation I of concepts c_1 and c_2 situated in the hyponymy relation and the concept c_3 which is

a property of c_1 is *equivalent* iff:

$$c_2 \in hyponymy(c_1) \wedge c_3 \in f^o_i(c_1) \wedge I(c_1) = p_1 \wedge \\ I(c_2) = p_2 \wedge I(c_3) = p_3 \wedge p_2 \in subtypeOf(p_1) \wedge \\ p_3 \in [\tau(f^o_i)](p_1)$$

This represents the ideal case in which the concepts, their properties and relations are accurately reflected in the code.

Definition The implementation I of concepts c_1 and c_2 situated in the hyponymy relation and the concept c_3 which is a property of c_1 exhibits *monotonic properties mismatch* iff:

$$c_2 \in hyponymy(c_1) \wedge c_3 \in f^o_i(c_1) \wedge I(c_1) = p_1 \wedge \\ I(c_2) = p_2 \wedge I(c_3) = p_3 \wedge p_2 \in subtypeOf(p_1) \wedge \\ p_3 \in [\tau(f^o_i)](p_2)$$

Intuitively, in this case the properties which belong to the super-concept are reflected in the code to belong to the implementation of the sub-concept. Due to the fact that the properties of a super-concept are also properties of all its sub-concepts, this case represents a healthy mismatch.

Definition The implementation I of concepts c_1 and c_2 situated in the hyponymy relation and the concept c_3 which is a property of c_2 exhibits *non-monotonic properties mismatch* iff:

$$c_2 \in hyponymy(c_1) \wedge c_3 \in f^o_i(c_2) \wedge I(c_1) = p_1 \wedge \\ I(c_2) = p_2 \wedge I(c_3) = p_3 \wedge p_2 \in subtypeOf(p_1) \wedge \\ p_3 \in [\tau(f^o_i)](p_1)$$

This is the case in which the properties of the subconcept are reflected in the code to belong to the superconcept. This situation is not healthy.

Example: The concept of MONTHS, which in the ontology belong to the GREGORIAN CALENDAR concept, are implemented as attributes of the Calendar class (Figure 8h). Since the Calendar class is intended to be the base class for all calendar implementations, this implementation of months can generate problems for the subclasses of the Calendar implementing calendar systems which define

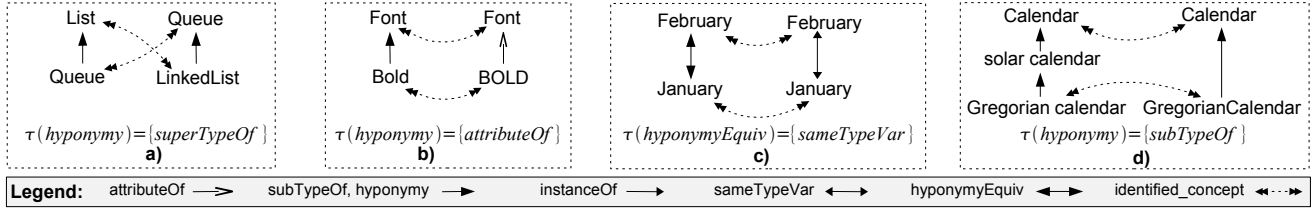


Figure 9. The identification of mismatches

the months completely different. An example would be the implementation of the HEBREW CALENDAR concept, which has 13 months, as subclass of Calendar. For the clients of the HebrewCalendar class, it would be unnatural to see the “classical” months.

6. Semi-automatic identification of biases

In [12, 11] we propose a semi-automatic method for identifying the ontological entities in the code based on identifying mappings between the ontology and the program graphs. With every mapping is identified a possible occurrence of a concept in the program. Each automatically identified occurrence needs to be subsequently validated by a human.

Between the WordNet relation types (Figure 2) and the program relation types from (Figure 3) we define the following correspondence: $\tau(hyponymy) = \{subTypeOf, instanceOf\}$ and $\tau(meronymy) = \{attributeOf\}$. This follows from the similarity between the real-world generalization relations and the type system relations on the one hand and between the real-world aggregation relations and module-system relations on the other hand [12].

In Figure 10 we present examples of the mappings used for the identification of the CALENDAR related concepts: The CALENDAR and GREGORIAN CALENDAR concepts were identified since in the ontology they are in the hyponymy relation, in the program in the subtyping relations and the hyponymy and subtyping relations are similar ($subTypeOf \in \tau(hyponymy)$). The JANUARY and FEBRUARY concepts were identified since in the ontology they are hyponyms of the same concept and in the program they are attributes with the same type.

Using our method for identification of concepts within programs we could discover most of the mismatch examples presented in the previous section. There were however special cases when we needed to adapt our method:

- To identify the *relations mismatches* we changed the mapping rule (i.e. function τ) between the ontology and the program relations types (i.e. between R^o and R^p): for identifying the *inverted relation* mismatch we used the function $\tau(hyponymy) = superTypeOf$

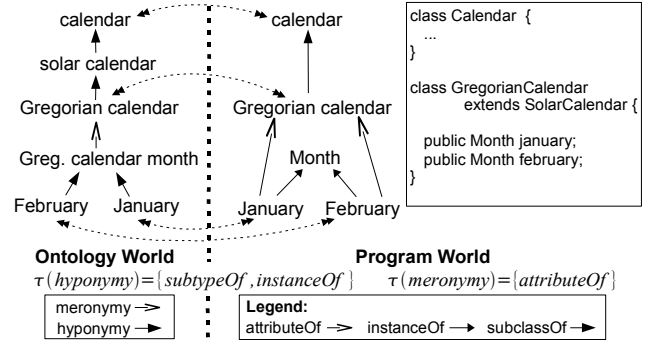


Figure 10. Identifying concepts with Bridge

(Figure 9a); for identifying the *distorted relation* bias we used the function $\tau(hyponymy) = attributeOf$ (Figure 9b).

- To identify the *homogeneous implementation* mismatch we defined new relations at the ontology and program levels: between concepts with the same hypernym we defined the *hyponymyEquiv* relation and respectively between variables with the same type we defined the *sameTypeVar* relation. Corresponding to these relations we defined the mapping: $\tau(hyponymyEquiv) = sameTypeVar$. The *heterogeneous implementation* was identified by searching for two attributes that implement similar concepts and that have different types.

For finding these examples we used the Bridge⁵ tool and the Insider⁶ reverse engineering platform. Bridge identifies concepts in the code by mapping program elements to ontological entities.

7. Related Work

We structure the literature related to our work along two dimensions: library design and ontologies.

Library Design Despite the high importance of the libraries usability, we could found only a few studies which

⁵Developed at Technische Universität München, Germany

⁶Developed at the LOOSE Research Group (LRG) from Timisoara, Romania, www.loose.upt.ro

deal with it. Korson [7] identifies an extensive set of desirable attributes of software libraries. Thus, libraries should be complete (i.e. provide coverage of an entire concept), consistent (i.e. each facet of the library follows an uniform approach), easy-to-learn for novice users, easy-to-use (i.e. information and code are easy to find), extendable, intuitive (i.e. the design corresponds to the intuition of a domain expert). Meyer [9] also discusses a set of high-level library quality attributes like: uniformity of component interfaces or functional completeness and he defines informal guidelines that the library builders should follow: “all the components of a library should proceed from an overall coherent design, and follow systematic, explicit and uniform conventions”. Clarke [2] evaluates API usability by using the cognitive dimensions framework which also contains the consistency, extensibility and domain correspondence as desired qualities of a good API. He performs user interviews and studies how well does the library support its clients in writing programs for particular use-cases.

By comparing the interface of a library to an ontology we propose a framework for expressing and evaluating the distance between the real-world concepts and their implementation in the library. In this way we provide a method to formally express and (semi-)automatically evaluate the above quality attributes.

Ontologies Domain ontologies are used in [5] to evaluate the suitability of a modeling language to model a set of real-world phenomena in a given domain. The evaluation is made by comparing the level of homomorphism between the modeling language constructs and a reference ontology of the domain. In a similar manner, we identify the mismatches in the implementation of concepts in libraries by comparing the level of homomorphism between the ontology and the public interface of the library.

In [12] we presented a method to locate concepts in the code based on mapping programs to ontologies. In [11] we used the explicit mapping between the ontology, program and program entities names to identify semantic defects such as bad naming of program entities and logical duplication. The present work continues in the same direction and identifies mismatches in the representation of concepts and relations between them within the public interface of libraries.

8. Conclusions

Important quality attributes of libraries, such as usability and extensibility, can be evaluated in terms of the mismatches between the real-world concepts and relations and their implementation in the library’s public interface. In this paper we defined a formal framework to describe mismatches in the implementation of concepts. This framework

proved to be powerful enough to enable the description of several classes of interesting mismatches. We discussed the effect of particular mismatches on the library clients and provided real-world examples of mismatches that we (semi-)automatically identified in the Java library.

As future work we plan to extend the set of biases presented in this paper and to better understand their impact on the library’s quality. We also plan to use domain specific ontologies as basis for libraries evaluation. In the long-term, we plan to use the identification of biases as starting point for improving libraries interfaces.

Acknowledgements: The authors would like to thank Diana Ratiu, Martin Feilkas, Maria Spichkova and the anonymous reviewers for providing valuable comments.

References

- [1] J. Bloch. How to design a good API and why it matters. Keynote speech at Library-Centric Software Design (LCSD’05), 2005.
- [2] S. Clarke. Measuring API usability. *Dr. Dobbs Journal Windows/.NET Supplement*, pages 7–9, 2004.
- [3] M. R. Genesereth and N. J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.
- [4] 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.
- [5] G. Guizzardi, L. F. Pires, and M. van Sinderen. An ontology-based approach for evaluating the domain appropriateness and comprehensibility appropriateness of modeling languages. In *MoDELS ’05*. LNCS, Springer Verlag, 2005.
- [6] P. E. Hayes. RDF semantics. Technical report, W3C Recommendation, 2004.
- [7] T. Korson and J. D. McGregor. Technical criteria for the specification and evaluation of object-oriented libraries. *Softw. Eng. J.*, 7(2):85–94, 1992.
- [8] D. L. McGuinness. Ontologies come of age. In *Spinning the Semantic Web*, 2003.
- [9] B. Meyer. *Reusable software: the Base object-oriented component libraries*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
- [10] G. A. Miller. WordNet: a lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.
- [11] D. Ratiu and F. Deissenboeck. How programs represent reality (and how they don’t). In *WCRE ’06*. IEEE CS Press, 2006.
- [12] D. Ratiu and F. Deissenboeck. Programs are knowledge bases. In *ICPC ’06*. IEEE CS Press, 2006.