

From Reality to Programs and (Not Quite) Back Again

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

Making explicit the mappings between real-world concepts and program elements that implement them is an essential step in understanding, using or evaluating the public interface of programs, libraries and other collections of classes that model core domain concepts. Unfortunately, due to the big abstraction gap between the modeled domain and today's programming languages, the mapping is most of the times ambiguous as concepts and relations from the real world are distorted and diffused in the code. In this paper we present a comprehensive formal framework for describing the many-to-many mappings between domain concepts and the program elements, real-world relations and program relations and the real-world concept names and program identifiers. This framework allows us to describe and discuss typical classes of diffusion of the domain knowledge in code. Based on our formal framework we describe an algorithm to recover the mappings between entities from an ontology and program elements. We illustrate the framework by using examples from the Java standard library.

1. Introduction

The high abstraction distance between the real world and programs as well as the flexibility of today's general purpose programming languages, opens up plethora of different ways of implementing concepts in code. Consequently, in many existing programs, the mapping between real world concepts and program elements suffers from varying degrees of *diffusion*: domain concepts are commingled with implementation concepts, crammed within single program elements, scattered throughout the program or referred to through non-suggestive names. As humans think in terms of real-world concepts, dealing with such program elements demands additional comprehension efforts.

Below we present an example that illustrates a common type of diffusion. Although both code snippets implement the concepts OVAL, RECTANGLE and DRAW, each imple-

mentation uses different program elements for representing the concepts and thereby exhibits different characteristics in terms of modularization, usability and extensibility. These characteristics, in turn, influence crucial maintenance activities like *concept location* [13] and *concept assignment* [3] as well as the addition of new features.

```
class Oval extends Shape {...}
class Rectangle extends Shape {...}
class Painter {
  draw (Shape aShape){...}
}

class Painter {
  drawOval (float x, float y, int width, int height){...}
  drawRectangle (float x, float y,
    int width, int height) {...}
  drawRectangle (float x1, float y1,
    float x2, float y2){...}
}
```

We are aware that programs built with today's general purpose programming languages will always exhibit a fair degree of diffusion. However, we claim that the *public interface* of programs, libraries, frameworks and collections of classes that model core domain concepts must reflect the modeled concepts and their interrelations as accurately as possible. Only if there is a cycle from reality to programs and back again, programmers can directly make use of their domain knowledge and thereby efficiently carry out common software maintenance tasks.

In order to evaluate and improve the mapping between the real world and the public interfaces of programs, a thorough understanding of the intricacies involved in this mapping is imperative. Based on our work in identifier naming and knowledge representation in programs [5, 6, 16–18], this paper now presents a comprehensive formal framework that describes how real world concepts are mapped to program elements and vice versa (Section 2). We thereby fill the gaps in the work presented earlier and introduce a solid foundation for the discussions of representation ambiguities by providing a formal classification of diffused reflection of concepts in code (Section 3). We explain how the framework can be instantiated using ontologies to describe the real world concepts (Section 4) and present an algorithm to identify concepts in programs (Section 5). We illustrate this approach with the results of two experiments with the Java standard library (Section 6).

2. A Formal Representation of Programs

Many program comprehension activities start from the identification of domain concepts in code [15]. Program identifiers play an important role in this process since they represent the bridge between the program and the domain worlds [1]. In Figure 1 we present a unified meta-model that explicitly bridges the domain concepts to the code via identifiers [6]. In order to describe the relations between the domain concepts, the names and the program elements, we divide the meta-model into three layers. We assume that instances of this meta-model, or parts thereof, are (implicitly) built in many of the comprehension activities, and thus, investigating it in detail helps in understanding problems encountered in the comprehension.

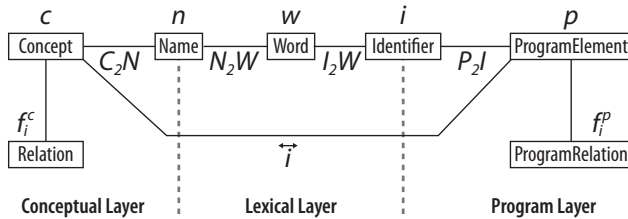


Figure 1. The unified meta-model [6]

In the following we present a formalization of this meta-model that allows us to describe the ambiguities introduced by the diffusion of real world concepts in the code (Section 3). The names of the most important entities and relations from our formalization are presented in Figure 1 next to their corresponding meta-model entities and relations.

Notations: If M is a set, through M^* we denote the set of all finite sequences formed from elements of the set M . If $f : M \rightarrow N$, and $M' \subset M$ then $f[M'] = \bigcup_{m \in M'} f(m)$. If $f_{1\dots n} : M \rightarrow \mathcal{P}(M)$ is a set of functions and $m \in M$ then $\langle f_i f_{i+1} \dots f_{i+k} \rangle (m) = f_{i+k}[\dots f_{i+1}[f_i(m)]]$

Program layer The program layer contains a set of named program elements P and a set of typed relations between them R^p . Each relation type is modeled through a function f_i^p that maps a program element to the set of related program elements. The relations between program elements are defined by the programming language or through programming conventions. The set of identifiers of all the program elements from P is denoted through I . Every program element has a unique identifier - given by the function P_2I . The identifiers belong both to the program and the lexical layers. All together the program layer Π is formally described as:

$$\begin{aligned} \Pi &= (P, R^p), \quad \text{with } P = \{p_1, \dots, p_m\}, \\ R^p &= \{f_1^p, \dots, f_n^p\}, \quad f_i^p : P \rightarrow \mathcal{P}(P) \quad \text{and} \\ I &= \{i_1, \dots, i_k\}, \quad P_2I : P \rightarrow I \end{aligned}$$

Example: In the lower part of Figure 2 we present an example program (left) and based on it an instantiation of the program layer (center). In this example, program elements are public classes, attributes, methods and their parameters. We have three language defined relations: *hasSubClass*, *hasAttribute* and *hasParameter* and one relation defined through programming conventions: *hasAccessor*. The program elements and their relations can be seen as a typed graph (right). As we notice, some program elements (e.g. the method `clone`) are isolated in the program graph since none of the above relations link them to other program elements.

Conceptual layer The conceptual layer contains a set of domain concepts C that are relevant for understanding the program. Beside the concepts directly implemented in the program, this layer also contains concepts that are strongly related to them. The conceptual layer contains also a set of typed relations among concepts R^c . Each relation type is modeled through a function f_i^c that maps a concept to a set of related concepts. The set of names of all concepts from C is denoted through N . Due to the synonymy, each concept can have more names; due to the polysemy, every name can refer to more concepts. This many-to-many relation is captured by two functions: concepts-to-names (C_2N) and names-to-concepts (N_2C). The concept names belong both to the conceptual and lexical layers. All together the conceptual layer Ω is formally described as:

$$\begin{aligned} \Omega &= (C, R^c), \quad \text{with } C = \{c_1, \dots, c_m\}, \quad (1) \\ R^c &= \{f_1^c, \dots, f_n^c\}, \quad f_i^c : C \rightarrow \mathcal{P}(C) \quad \text{and} \\ N &= \{n_1, \dots, n_l\}, \quad C_2N : C \rightarrow \mathcal{P}(N), \quad N_2C : N \rightarrow \mathcal{P}(C) \end{aligned}$$

Example: In the upper part of Figure 2 we present an example of a set of concepts (left) and based on them an instantiation of the conceptual layer (center). A graph representation is shown on the right. In this example we have nine concepts and two relations: *isA* and *hasProperty*. As we notice, there are two additional concepts (i.e. WIDTH and HEIGHT) beside those represented directly in the program.

Lexical layer The lexical layer is centered around a set of morphologically normalized words denoted through W . The words from this layer are the reunion of the words obtained from the names of all concepts and the words obtained by splitting all the identifiers. The function identifiers-to-words (I_2W) maps the identifiers to a sequence of normalized words; the function names-to-words (N_2W) maps the concept names to their corresponding sequence of normalized words. The lexical layer is described as:

$$\begin{aligned} W &= \{w_1, \dots, w_s\}, \\ I_2W &: I \rightarrow W^*, \quad N_2W : N \rightarrow W^* \end{aligned}$$

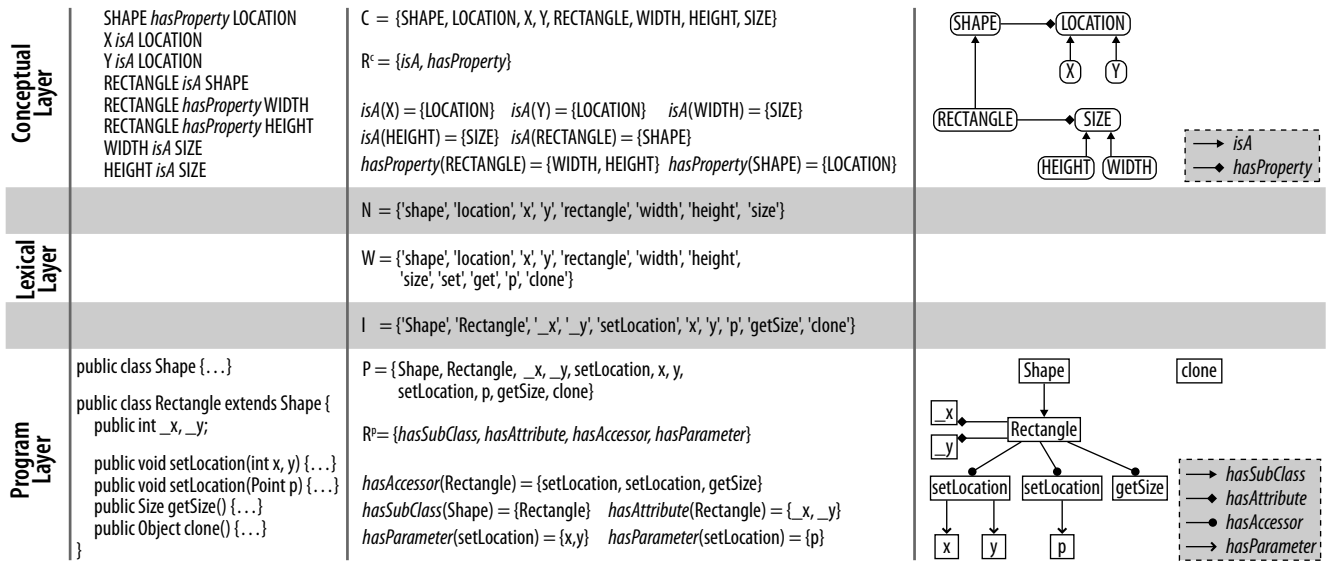


Figure 2. Program layers example

Mapping concepts and program elements In the following we define a set of functions that directly connect the conceptual and the program layers. We consider that there is a many-to-many relation between program elements and the concepts that they refer to. We capture this mapping through the following two functions: the *implementation function* \vec{i} denotes the set of program elements that refer to a concept; the *interpretation function* \overleftarrow{i} denotes the set of concepts to which a program element refers:

$$\vec{i} : C \rightarrow \mathcal{P}(P) \quad \overleftarrow{i} : P \rightarrow \mathcal{P}(C)$$

The duality between these functions is expressed through the following equation:

$$\overleftarrow{i}(p) = \{c \in C \mid p \in \vec{i}(c)\}$$

In the context of program comprehension, the function \vec{i} represents *concepts location* and the function \overleftarrow{i} *concepts assignment*. When we refer to both of these two functions we use the following notation: \overleftrightarrow{i} .

Examples: Based on the layers configuration from Figure 2, below we exemplify several instances of \vec{i} and \overleftarrow{i} :

$$\begin{aligned} \vec{i}(SHAPE) &= \{Shape\}, \vec{i}(x) = \{_x, _x\}, \vec{i}(SIZE) = \{getSize\}, \\ \vec{i}(LOCATION) &= \{setLocation, setLocation\}, \vec{i}(WIDTH) = \emptyset \\ \overleftarrow{i}(getSize) &= \{SIZE\}, \overleftarrow{i}(Shape) = \{SHAPE\}, \overleftarrow{i}(_x) = \{x\} \\ \overleftarrow{i}(setLocation) &= \{LOCATION\}, \overleftarrow{i}(x) = \{x\}, \overleftarrow{i}(clone) = \emptyset \end{aligned}$$

Observation: Intuitively, the functions \overleftrightarrow{i} are similar to the knowledge of a project guru that knows exactly how and where are the domain concepts implemented. If no guru is available, these functions can be defined by using other

sources of information such as: reading the documentation or the source code. The only restriction of the functions \overleftrightarrow{i} is made by the limited set of concepts available at the conceptual layer – i.e. if the project guru does not know a domain concept then he can not find it in the code.

Mapping real-world and program relations There is a many-to-many mapping between sequences of real-world relations and sequences of program relations: a relation or a sequence of relations between real-world concepts can be reflected in the code through several sequences of relations between program elements and vice versa. We capture this correspondence through two functions: *relations implementation* \vec{t} and *relations interpretation* \overleftarrow{t} :

$$\vec{t} : R^{c*} \rightarrow \mathcal{P}(R^{p*}), \quad \overleftarrow{t} : R^{p*} \rightarrow \mathcal{P}(R^{c*})$$

Definition: Let $f : M \rightarrow \mathcal{P}(M)$ be a function describing a relation. Through $-f$ we denote the function that describes the *reverse* of this relation such that $m_2 \in f(m_1) \Leftrightarrow m_1 \in (-f)(m_2)$ ¹. For example, if the relation “RECTANGLE *hasAttribute* $_x$ ” holds then “ $_x$ *-hasAttribute* RECTANGLE” also holds.

In Figure 3 we present examples of mappings between sequences of relations from real world and sequences of relations from the program. In (a) an attribute is used to implement a *part* of a concept, in (b) an attribute is used to implement a *property* of a concept. In (c) the property of a concept is implemented through a sequence of program relations (attribute of subclass). In (d), two consecutive *isA* relations are implemented through a single

¹For each function f representing a relation, we will use in the following both prefix and infix notations: $m_2 \in f(m_1) \equiv m_1 f m_2$

hasSubClass relation (OFFSPRING is left out). In (e) two consecutive *isA* relations are implemented through two consecutive *hasSubClass* relations.

Observation: The function \vec{t} can be seen as the translation of a path from the graph at the conceptual layer into a set of paths from the program graph.

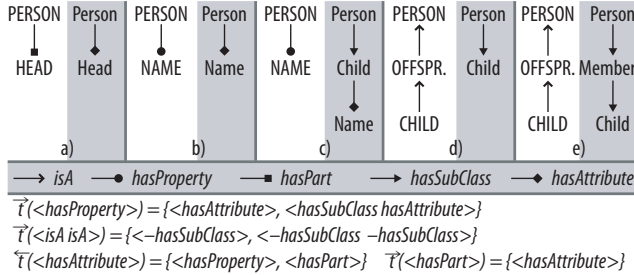


Figure 3. Examples of relations mapping

Naming clues In real-life projects the functions \overleftarrow{i} are not computable. In practice one of the most important sources of information for recovering the mappings between the program and the conceptual layers is given by the identifiers names. In this paragraph we discuss how identifiers offer clues in the mapping between certain domain concepts and the program elements that they represent. The link between the compound identifiers and the concept names that they contain is given by the function *CandNms* (candidate names): given an identifier, this function computes a set of concept names that are referred to by the identifier. Based on the *CandNms* we define the *CandCts* (candidate concepts) function: given a program element this function computes based on its name the possible concepts that the program element refers to.

$$\begin{aligned} \text{CandNms} : I &\rightarrow \mathcal{P}(N), & \text{CandCts} : P &\rightarrow \mathcal{P}(C) \\ \text{CandCts}(p) &= \{c \mid c \in C \wedge i = P_2I(p) \wedge C_2N(c) \cap \\ &\quad \text{CandNms}(i) \neq \emptyset\} \end{aligned}$$

Below we give a specification of the *CandNms* function. Intuitively, a concept name *n* is referred by an identifier *i* iff the sequence of words that make up the name is a subsequence of the sequence obtained from splitting the compound identifier.

$$\text{CandNms}(i) = \{n \in N \mid N_2W(n) \sqsubseteq I_2W(i)\}$$

In Figure 4 we present examples of these two functions.

Observation: While the functions \overleftarrow{i} represent the real mapping between the concepts and the program elements. The function *CandCts* represent an approximate mapping on the basis of names similarity and hence, plays a central role in all name-based comprehension activities.

Conceptual Layer $C = \{\text{CALENDAR, GREGORIAN CALENDAR, MONTH}\}$

$N = \{\text{'calendar', 'gregorian calendar', 'month'}\}$

Lexical Layer

$W = \{\text{'calendar', 'gregorian', 'set', 'month', 'getmonth', 'equals'}\}$

$I = \{\text{'Calendar', 'GregorianCalendar', 'setMonth', 'getmonth', 'equals'}\}$

Program Layer

```
class Calendar {...}      public void setMonth(...){...}
class GregorianCalendar {...} public void getmonth(){...}
                           public boolean equals(){...}
```

$\text{CandNms}(\text{'Calendar'}) = \{\text{'calendar'}\}$	$\text{CandNms}(\text{'setMonth'}) = \{\text{'month'}\}$
$\text{CandNms}(\text{'getmonth'}) = \emptyset$	$\text{CandNms}(\text{'equals'}) = \emptyset$
$\text{CandNms}(\text{'GregorianCalendar'}) = \{\text{'gregorian calendar', 'calendar'}\}$	
$\text{CandCts}(\text{Calendar}) = \{\text{CALENDAR}\}$	$\text{CandCts}(\text{setMonth}) = \{\text{MONTH}\}$
$\text{CandCts}(\text{getmonth}) = \emptyset$	$\text{CandCts}(\text{equals}) = \emptyset$
$\text{CandCts}(\text{GregorianCalendar}) = \{\text{GREGORIAN CALENDAR, CALENDAR}\}$	

Figure 4. Naming Clues

3. Diffusion of reality in programs

In the previous section we presented a general framework for describing the mappings between entities from the conceptual, lexical and program layers. In the following we use this framework to express typical diffusions of the real-world into programs. Our formal framework contains three orthogonal sources of diffusion: the many-to-many mapping between the real world concepts and program elements; the many-to-many mapping between the real-world relations and the relations in the programs and the ambiguity of reflecting the real-world concepts through the names of program elements.

For each of these diffusion sources we start our presentation with the ideal (non-diffusion) case and continue with a list of diffusions. We present each diffusion in a formal manner, after that we present its intuition and discuss its implications for comprehension and maintenance. At the end of each category we present examples from different parts of the Java library.

3.1. Diffusion of concepts

A concept $c \in C$ has a *direct implementation* iff

$$\overleftarrow{i} \left[\overrightarrow{i}(c) \right] = \{c\}$$

Intuitively, every time when a concept *c* is referred in the code, the program elements that refer to it do not refer to any other concept. This represents the ideal case when individual concepts from the real world have individual counter-parts among the program elements within the public interface. In real world programs many concepts are weaved with other concepts in the same program elements: $\left| \overleftarrow{i} \left[\overrightarrow{i}(c) \right] \right| \neq 1$. In the following paragraphs we discuss in details the diffusions of concepts.

3.1.1. Pure details. A program element $p \in P$ exhibits *pure details* iff: $\overleftarrow{i}(p) = \emptyset$.

Intuitively, pure details means that according to our conceptual level knowledge the program element can not be assigned to any concept. From the public interface clients' point of view, the pure details represent accidental complexity in the library with which they have to deal. Even if the pure details do not have any meaning for the modeled domain, many times they belong to the essential machinery of the programming language. By introducing accidental complexity, pure details are a burden for the users of program interfaces.

3.1.2. Absent implementation. A concept $c \in C$ exhibits *absent implementation* iff: $\overrightarrow{i}(c) = \emptyset$.

Intuitively, absent implementation means that there are concepts from the modeled domain that are not referred by any program element. From the public interface clients' point of view, the absent implementation represents functionality that is missing. If a concept c exhibits absent implementation, it needs to be reconstructed from the implementation of its neighbor concepts.

3.1.3. Implicit implementation. A concept $c \in C$ exhibits *implicit implementation* iff:

$$\overrightarrow{i}(c) = \emptyset \wedge \exists p \in P, \forall c' \in f_i^c(c). \overrightarrow{i}(c') \cap [t(f_i^c)](p) \neq \emptyset$$

Intuitively, instead of referring to a concept explicitly through a program element, it is referred only implicitly through a set of related program elements that refer to its neighbors. This is a typical case of abstraction lost in a program when instead of using an explicit representation of a concept (denoted as c in the formula) we use it implicitly only through its neighbors.

3.1.4. Compacted implementation. Let $p \in P$ and $c_1, \dots, c_n \in C$. The program element p represents a *compacted implementation* iff:

$$\overleftarrow{i}(p) = \{c_1, \dots, c_n\}$$

Intuitively, program elements that refer to several distinct concepts exhibit compacted implementation. Ideally a program element should refer to only one concept and in programs the concepts should be composed exclusively by using the composition mechanisms provided by the language (e.g. class membership, method call, parameter passing). In the case of compacted implementation more concepts are weaved into a single program element. In many cases it is difficult to determine which parts of the program element implementation belong to a concept and which to the others. It is even more difficult to find where and how is the

composition between them realized. Program elements that represent compacted implementation negatively affect the modularization of a program by intermingling more concepts.

3.1.5. Intimate program neighbors. A set of concepts $\{c_1, \dots, c_n\} \subseteq C$ represents *intimate program neighbors* of a concept $c \in C$ iff

$$c_i \neq c \wedge c_i \in \overleftarrow{i}[\overrightarrow{i}(c)]$$

Intuitively, intimate neighbors of a concept c are all those concepts c_i that are referred by program elements that also refer to c . Only because of the fact that these concepts are implemented together, it is created a dependency between them at the code level. Since program elements are "indivisible" units of program, when a program element refers to several concepts at a time, then these concepts can not be dealt with individually. In order to understand how are these concepts combined, one needs to inspect also the implementation of their intimate neighbors. The intimate neighbors negatively affect the modularization by creating implicit and hidden coupling at the code level between the implementation of distinct concepts.

Examples: An example of the pure details diffusion is the method `Calendar.hashCode` that even if it does not implement any domain concept, has an essential role in Java.

The `RoundRectangle2D` class compacts the concepts `ROUND` and `RECTANGLE`. As explained above, different parts of the implementation of the classes that represent compacted implementation refer to different concepts. For example, the concept `ROUND` is referred in the interface of this class through four methods (i.e. two `setRoundRect`, `getArcHeight`, `getArcWidth`).

The implementations of concepts `DRAW` and `FILL` is compacted with several geometrical figures in the methods of the `Graphics` class (e.g. `Graphics.drawOval(...)`, `Graphics.fillPolygon(...)`). Even if there is a clear relation between drawing actions and the geometrical figures (e.g. the draw action is performed over figures), these relations are implicit in the body of the method. Furthermore, due to the compact implementation, we notice also a pollution of the interface of `Graphics`: we have several program elements that implement combinations between these concepts (e.g. `drawOval()`, `drawRect()`, `fillOval()`, `fillRect()`). These methods take as parameters the properties of the figure to be drawn. Thus, the geometrical figures are expressed only implicitly through their set of properties (i.e. position and size). In the public interface of the more recent version of the `Graphics` class, namely `Graphics2D`, the concepts are explicitly represented and composed: there is a class hierarchy for shapes and a general method for drawing them: `Graphics2D.draw(Shape)`.

3.2. Diffusion of names

A program element $p \in P$ has an ideal name iff:

$$\overleftarrow{i}(p) = \text{CandCts}(p)$$

Intuitively, all the concepts that a program element refers to are represented as parts of its name. Many of the real world situations represent cases when the names offer only clues about the implemented concepts. In the following paragraphs we present diffusions of the concepts names.

3.2.1. Non-suggestive naming. A program element $p \in P$ has *non-suggestive name* iff:

$$\exists c \in C. c \in \overleftarrow{i}(p) \wedge c \notin \text{CandCts}(p)$$

Intuitively, one of the concepts that are implemented is not referred by the program element name. In this case, the concepts that the program element refers to can be identified only by reading the code or the documentation and this negatively affects the usability of the interface.

3.2.2. Clueless naming. A program element $p \in P$ exhibits *clueless naming* iff:

$$\overleftarrow{i}(p) \neq \emptyset \wedge (\text{CandCts}(p) \cap \overleftarrow{i}(p) = \emptyset)$$

Intuitively, a program element refers to some domain concepts but its name does not give any clue about any of these concepts. When the names of all program elements are clueless, then we have a typical case of code obfuscation. If the program element with clueless naming belongs to the public interface of the library, the only way to use this element is to inspect other sources of information beside its name - e.g. read the documentation, read the implementation.

3.2.3. Misleading naming. A program element $p \in P$ has *misleading name* iff:

$$c \notin \overleftarrow{i}(p) \wedge c \in \text{CandCts}(p)$$

Intuitively, a program element has a misleading name if the name refers to concepts that are not implemented in the program element. This represents a worst sub-case of code obfuscation since one can easily misuse this program element.

3.2.4. Partially known implementation. A program element $p \in P$ exhibits *partially known implementation* iff:

$$\exists w \in I_2W(P2I(p)). w \notin N_2W \left[C_2N \left[\overleftarrow{i}(p) \right] \right]$$

Intuitively, only a subset of the words of an identifier can be identified as belonging to some concepts. This situation can happen because of two main reasons: either because of bad naming such as the usage of abbreviations or acronyms or because of the unknown words refer to the implementation concepts and they are not covered by the domain knowledge present in our conceptual layer.

Examples: The method `Rectangle.outcode(...)`, that for a given point determines its relative position with respect to the rectangle, has a clueless name. The name of the method `getActualMinimum` of the class `Calendar` represents an example of non-suggestive naming – we can recognize the concept `MINIMUM` but not the other concepts it refers to. In order to use these two methods, one needs to have a deep knowledge of the implementation details of the `CALENDAR` and `RECTANGLE` concepts in the Java library. An example of misleading name is the parameter `date` of the method `Calendar.set(int year, int month, int date)`. Here, the name “date” is used to denote the concept `DAY OF THE MONTH` instead of a particular point in time as we would expect. The word “set” from the name of the method `Rectangle.setSize` is unknown to the geometrical domain and thus the name of this method represents a case of partially known implementation.

3.3. Diffusion of relations

The relation $f_i^c \in R^c$ is *ideally implementable* iff:

$$\exists f_k^p \in R^p. \left(\overrightarrow{t}(\langle f_i^c \rangle) = \langle f_k^p \rangle \right) \wedge \left(\overleftarrow{t}(\langle f_k^p \rangle) = \langle f_i^c \rangle \right)$$

Intuitively, there is a one-to-one correspondence between real-world relation and program relations. Due to the big abstraction gap between the real-world and programs most of the times the relations are diffused.

3.3.1. Relations approximations. The relation $f_i^c \in R^c$ is *approximated* iff:

$$\begin{aligned} \exists f_k^p \in R^p. \langle f_k^p \rangle \in \overrightarrow{t}(\langle f_i^c \rangle) \wedge \\ \exists \langle f_i^p \dots f_j^p \rangle \in R^{p*}. \langle f_i^p \dots f_j^p \rangle \in \overrightarrow{t}(\langle f_i^c \rangle) \end{aligned}$$

Intuitively, a relation from the real world does not have a direct correspondent among the program relations. This happens because there are much more relations in the real world as the ones directly definable in the programs. Unless special conventions (e.g. naming conventions) for implementing complex real-world relations are used, they can only be approximated in the code.

3.3.2. Ambiguous interpretation. The sequence of relations $\langle f_i^p \dots f_j^p \rangle \in R^{p*}$ is *ambiguous* iff:

$$|\overleftarrow{t}(\langle f_i^p \dots f_j^p \rangle)| > 1$$

Intuitively, a sequence of relations from the program can be interpreted in more ways at the conceptual level. This impacts negatively the reverse engineering of relations represented within programs.

Examples: A possible direct relation between concepts with the same superordinate is the *hasSibling*. This relation is not directly expressible by using program relations and needs to be expressed through a sequence of relations: $t(\langle hasSibling \rangle) = \{ \langle -hasSubClass \ hasSubClass \rangle \}$. The *hasAttribute* relation between a class and one of its attributes can be interpreted in two different ways: as the *hasProperty* relation between a concept and one of its properties and as *hasPart* between a concept and one of its parts.

4. Materializing the conceptual layer

In the previous sections we presented our formalism for evaluating the representation of real-world knowledge in programs. In order to make use of it in the practice, we need to find an appropriate representation for the conceptual layer. Following our formalization of real world as concepts and labeled relations between them, we propose in the following to use light-weight ontologies as a materialization of the conceptual layer.

4.1. Ontologies in a nutshell

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 particular purpose. It contains a set of objects together with their properties and relations [7]. An ontology is defined to be an *explicit specification of a conceptualization* [8] and is used for sharing the knowledge about a domain by making explicit the concepts and relations within it. In the present work we use an informal meaning of the term “ontology” - which we regard to comprise only concepts and relations between them, among which the most important is the “*isA*” hierarchical relation. In order to represent an ontology we use a graph language similar to the RDF graphs [9]. Entities within the ontology are the nodes of the graph and relations between them are represented as labeled arcs. Thus, ontologies match our view on the conceptual layer (Equation 1 on page 2) very well.

4.2. Sources of ontologies

Having an appropriate ontology is of capital importance for the practical application of our framework (e.g. to evaluate the public interface of a collection of classes). Even if ontologies are envisioned to provide means for sharing large quantities of knowledge from different domains, current off-the-self ontologies cover only restricted parts of some domains. Usually the ontologies are built for a particular purpose and represent the world concepts from a certain perspective that fits at best for achieving their purpose. Similarly, programs are also built for particular usage-scenarios and their interfaces offer better support for the implementation of central concepts. Thus, even if the number of the ontologies is growing rapidly, most of the times there aren't available ontologies that can be used to evaluate a program. In order to perform an ontology driven evaluation of an interface of an arbitrary program, we have no option but to build this ontology.

Our approach to manually build an ontology is based on the fact, that the domain knowledge described by the public interface of a program is rather small (in comparison to the program itself). Furthermore, we do not aim to cover an entire domain, but rather to analyze how different parts of it are reflected in the program's interface. Thus, the scope of the ontology is restricted to contain only an interesting subset of the domain that is implemented by the program. Below we present our approach for manually building the ontology that constitutes the conceptual layer. This approach is similar to the one presented in [14].

Step 1: Establish the analysis scope. According to the particular analysis scope, we need to decide on the kinds of concepts that we want to represent through our ontology.

Step 2: Gather concepts. From the list of identifiers belonging to the public interface program elements, extract a list of names that are recognized as representing concepts relevant for the analysis; for each name add one or more concepts in the ontology. For each concept, give in addition to its name a brief description in order to avoid confusion due to polysemy. Also record all the synonyms that refer to a concept.

Step 3: Build taxonomy. Arrange these concepts within a taxonomy based on the *isA* relation. Whenever there are sibling concepts and their parent name was not identified among the public identifiers add the parent of these concepts to our taxonomy.

Step 4: Build relations. Add additional relations between these concepts. We restrict ourselves to only two classes of additional relations: *hasProperty* and *hasPart*.

Step 5: Evaluation. Iteratively evaluate the consistency and the accuracy of the ontology.

Step 6: Refinement. Iteratively refine the ontology by adding missing concepts and relations. The ontology should

contain all the concepts and relations that are useful for the particular analysis. Due to the names ambiguity such as polysemy, it happens that some concepts were not added in the ontology in Step 2. It is also possible that not all relevant relations are added in Step 4. In order to capture these situations, after mapping the ontology to programs (see next section), we should inspect the program elements that could not be mapped to any concept even if their names denote concepts from the chosen ontology scope.

5. Identifying concepts within programs

In the following we present a concrete instance of our formal framework by specifying the set of relations at the conceptual (R^c) and program levels (R^p), the concrete function for mapping between them (\vec{t}) and the concrete functions for obtaining the set of words from the identifiers and names (I_2W and respectively N_2W). Following this, we present an algorithm for recovering the functions \overleftarrow{i} .

5.1. Instantiating our framework

Conceptual layer. The conceptual layer (Ω) is an ontology built according to our method presented in the Section 4.2. $R^c = \{isA, hasProperty, hasPart\}$.

Program layer. The program layer contains all elements belonging to the public interface of the program (i.e. classes, attributes, methods and parameters) together with the following relations: $R^p = \{hasSubClass, hasAttribute, hasMethod, hasParameter, hasAccessor\}$. The relation *hasAccessor* holds between a class and its accessor methods (i.e. methods whose name start either with “set” or with “get”).

Instantiating \vec{t} . Below we define a concrete set of mappings between sequences of real-world relations and correspondent sequences of relations in the code.

$$\begin{aligned} \vec{t}(\langle isA \rangle) &= \{\langle -hasSubClass \rangle, \langle -hasSubClass - hasSubClass \rangle\} \\ \vec{t}(\langle isA isA \rangle) &= \{\langle -hasSubClass \rangle, \langle -hasSubClass - hasSubClass \rangle\} \\ \vec{t}(\langle hasProperty \rangle) &= \{\langle hasAttribute \rangle, \langle hasSubClass hasAttribute \rangle, \\ &\quad \langle hasAccessor \rangle, \langle hasSubClass hasAccessor \rangle\} \\ \vec{t}(\langle hasPart \rangle) &= \{\langle hasAttribute \rangle, \langle hasSubClass hasAttribute \rangle, \\ &\quad \langle hasAccessor \rangle, \langle hasSubClass hasAccessor \rangle\} \\ \vec{t}(\langle isA - isA \rangle) &= \{\langle -hasParameter hasParameter \rangle, \\ &\quad \langle -hasMethod hasMethod \rangle, \\ &\quad \langle -hasAttribute hasAttribute \rangle\} \end{aligned}$$

Obtaining the words The function that maps the identifiers to words (I_2W) is defined by using classical conventions for splitting identifiers into words (e.g. CamelCase) followed by a morphological normalization algorithm for words. The function (N_2W) is defined similarly.

5.2. Concepts identification algorithm

Having instantiated our framework, the mapping between concepts and program elements is done by using a graphs matching algorithm that maps the program elements with their candidate concepts and sequences of edges from the program graph to equivalent sequences of edges from the concepts graph. Whenever a match between both nodes and relations is found we identify concepts in the code. Below we present our algorithm in pseudo-code and in Figure 5 we present an example of how does this algorithm work:

1. **for-each** $c \in C$ **do**
2. $reflection(c) = \{p \in P \mid c \in CandCts(p)\}$
3. **for-each** $\langle f_i^c \dots f_j^c \rangle \in R^{c*}$ **with** $\vec{t}(\langle f_i^c \dots f_j^c \rangle) \neq \emptyset$
4. $neighbors(c, \langle f_i^c \dots f_j^c \rangle) = \{c' \in C \mid c' \in \langle f_i^c \dots f_j^c \rangle(c)\}$
5. $reflection(c') = \{p' \in P \mid c' \in CandCts(p')\}$
6. **for-each** $p \in reflection(c), p' \in reflection(c')$
7. **check-if** $(\exists \langle f_k^p \dots f_l^p \rangle \in \vec{t}(\langle f_i^c \dots f_j^c \rangle))$ **with** $p' \in \langle f_k^p \dots f_l^p \rangle(p)$
8. **if-yes** (found mapping $(c, p), (c', p')$)

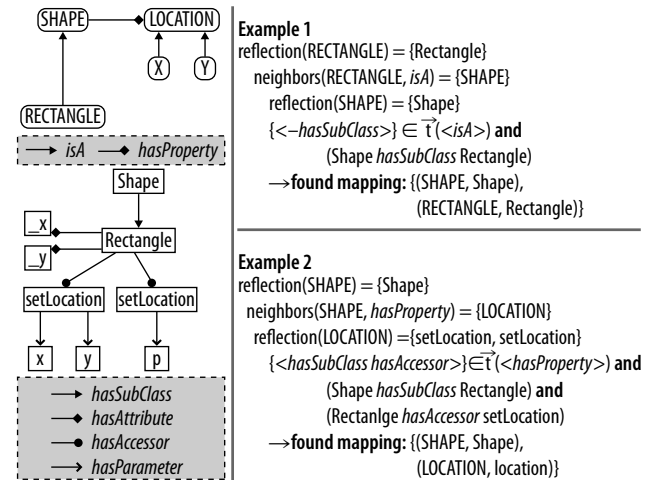


Figure 5. Concepts identification examples

Observations: 1) This algorithm is similar to algorithms for finding subgraph isomorphism which are known to be NP-hard. However, due to the fact that our mapping of paths is based on the function t , that in the practical situations is very limited, this algorithm does not pose any serious computational problems.

2) An important part of this algorithm is based on matching the identifiers names to the concepts names. Thus, our algorithm can not map concepts to program elements if the names of these program elements do not offer strong enough clues that they implement the concepts. Moreover, we can recover only those implementations where the sequence of relations from the real world is reflected in the code. Say it with other words, we recover the functions \overleftarrow{i} , defined in the Section 2, only partially.

6. Experiments

To illustrate the applicability of the formal framework, we present the results of two experiments with the semi-automatic concept extraction and diffusion detection. The experiments were carried out by using the reverse engineering platform Insider² and the concept extraction tool Bridge³. Subject of the experiments were the implementation of the calendar related concepts in the “Java Util” library and the implementation of geometry concepts in the “Java AWT” library.⁴

Before running our tools we construct the ontology that represents the conceptual layer. For the “calendar” experiment we have chosen the scope of our analysis to be the implementation of time related concepts (e.g. second, January, day, date) – we ignored the highly specific concepts (e.g. DAYLIGHT SAVING TIME or TIME ZONES). The scope of the “AWT” experiment is given by the geometry related concepts (e.g. shape, position, size) – we ignored the concepts related to advanced painting (e.g. colors). The construction of the ontologies took about two hours in the case of the calendar experiment and four hours in the case of the geometry experiment.

	calendar	awt.geom
#PublicClasses	5	25
#PublicMethods	99	454
#PublicAttributes	43	68
#PublicParameters	113	755
#Identifiers	128	290
#Words	88	190
#Names	51	100
#ConceptualLevelConcepts	47	89
#ConceptualLevelRelations	55	101
#KnownProgramElements	97	855
#IdentifiedConcepts	38	60
#TotallyKnownProgramElements	78	650
#PartiallyKnownProgramElements	19	205
#ImplemDetailsProgramElements	103	288
#IntimateProgramNeighbours	4	8
#CompactedImplementation	4	11

Table 1. Experiments Overview

Quantitative results. In Table 1 we present an overview of the experiments. The top-most part of the table presents quantitative information about the program, lexical and conceptual layers. The second part of the table presents a summary of the results of applying our algorithm for recovering

²Insider is developed at LOOSE Research Group, “Politehnica” University from Timișoara, Romania (www.loose.upt.ro)

³Bridge is developed at Technische Universität München, Germany

⁴In the calendar experiments we’ve chosen the following classes from the java.util package: Calendar, GregorianCalendar and Date. In the AWT experiments we have chosen the following classes from the java.awt package: Graphics, Graphics2D, Point, Polygon, Rectangle, Shape and from the java.awt.geom package: Arc2D, Arc2D.Double, Arc2D.Float, Ellipse2D, Ellipse2D.Double, Ellipse2D.Float, Line2D, Line2D.Double, Line2D.Float, Point2D, Point2D.Double, Point2D.Float, Rectangle2D, Rectangle2D.Double, Rectangle2D.Float, RectangularShape, RoundRectangle2D, RoundRectangle2D.Double, RoundRectangle2D.Float

\overleftrightarrow{i} on these two library parts: #KnownProgramElements is the number of program elements that could be associated with at least one concept; #IdentifiedConcepts represents the number of concepts from the conceptual layer that could be mapped on program elements.

Once the functions \overleftrightarrow{i} are identified, the identification of diffusion cases follows immediately from their formal description. The third part of our table presents a summary of diffusions: #TotallyKnownProgramElements represents the number of program elements for which all parts of their identifiers could be mapped to concepts; #PartiallyKnownProgramElements represents the number of program elements from the public interface whose identifiers contain words that could not be mapped to any concept; #ImplemDetailsProgramElements represent the number of public interface program elements that have names which could not be associated to any known concepts ($CandNms(p) = \emptyset$)⁵; #IntimateProgramNeighbours represent the number of concepts that are intimate neighbors at the program level; #CompactedImplementation represents the number of program elements that implement more concepts.

Diffusion identification example: In Figure 6 we present an example of the detection of the *compacted implementation* diffusion. The class RectangularShape compacts the implementation of the two concepts RECTANGULAR and SHAPE. This diffusion was identified by automatically recovering the function \overleftrightarrow{i} (illustrated in the figure with dotted lines).

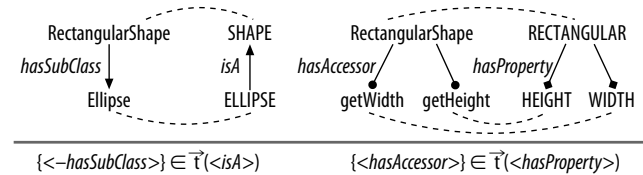


Figure 6. A Compacted Implementation

7. Related Work

Mapping domain concepts to source code has long been recognized as an important task in program comprehension and maintenance [3, 13, 15]. These works already point out that this mapping exhibits a certain degree of blurriness. We advance on this by providing a formal foundation that allows us to explicitly describe the mapping between the concepts and program elements and classify frequently encountered sources of ambiguities.

⁵In the geometry case are three categories of implementation details program elements: firstly are the classical Java implementation specific elements (e.g. the method equals), secondly are program elements with bad names (e.g. the failures of I_2W function in the case of names like “xpoints”) and thirdly are the program elements that implement concepts that do not belong to geometry and thus were not included in our conceptual layer (e.g. fonts, colors)

An important basis for the present work was provided by the research on the relevance of identifiers [1, 5, 11]. These papers acknowledge the fundamental role of the identifiers in program understanding and present first approaches for their systematic analysis. However, these papers do not capture explicitly the intricacies of the many-to-many mapping between domain concepts and their corresponding program elements as well as between concepts names and their reflection through identifiers. By doing so in this paper, we provide concrete means for evaluating naming quality.

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 [4, 10, 12]. 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. Furthermore, in both cases the concepts are coarse grained and are described by sets of words only. In this paper, we used our formal framework for defining a new algorithm for extracting concepts from programs by mapping entities from an ontology to program entities.

The basic method of linking programs to ontologies was presented in [17]. In [6] we initially presented the unified meta-model that serves as a basis for the formalization within this paper. In [16] we illustrated how the unified meta-model can be used to identify logical duplication, synonymy and polysemy in programs. In [18] we used the meta-model for the definition and detection of defects caused exclusively by mismatches in the implementation of real-world *relations*. However, the formalization in these papers is incomplete and does not allow to fully express the ambiguities generated through the mapping between the real world and programs. The current framework is more general and allows to describe all the defects mentioned in the last two papers.

8. Conclusions and Future Work

We presented a comprehensive formal framework that captures the many-to-many relations between concepts and program elements, conceptual relations and program relations, concept names and identifiers. The framework is targeted towards expressing typical real-world situations in the reflection of concepts in programs. Using this framework we characterized a set of ambiguities in the representation of the real-world concepts, relations and names in the interface of a program. Based on the formal framework we describe an algorithm for semi-automatic identification of concepts in the code and present our experiences with two

fragments of the Java library. In the future we plan to extend our framework by extending the set of relation types in programs and at the conceptual layer. We will also continue our work in the direction of building ontologies that are fit for analyzing the domain appropriateness of programs interfaces. For applying our approach on large systems in an industrial context, the cost-effectiveness of building ontologies remains to be evaluated.

References

- [1] N. Anquetil and T. C. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON '98*. IBM Press, 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] T. J. Biggerstaff, B. G. Mitbender, and D. Webster. The concept assignment problem in program understanding. In *ICSE '93*. IEEE CS, 1993.
- [4] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.
- [5] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, September 2006.
- [6] F. Deissenboeck and D. Ratiu. A unified meta-model for concept-based reverse engineering. In *ATEM '06*. Johannes Gutenberg-Univ. Mainz, 2006.
- [7] M. R. Genesereth and N. J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.
- [8] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum.-Comput. Stud.*, 43(5-6):907–928, 1995.
- [9] P. E. Hayes. Rdf semantics. Technical report, W3C Recommendation, 2004.
- [10] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *WCRE '05*, 2005.
- [11] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *SCAM '06*. IEEE CS, 2006.
- [12] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *ASE '01*, page 107. IEEE CS, 2001.
- [13] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *IWPC '05*, pages 33–42. IEEE CS, 2005.
- [14] N. F. Noy and D. McGuinness. Ontology development 101: A guide to creating your first ontology. *Stanford KSL Technical Report KSL-01-05*, 2000.
- [15] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *IWPC '02*. IEEE CS, 2002.
- [16] D. Ratiu and F. Deissenboeck. How programs represent reality (and how they don't). In *WCRE '06*. IEEE CS, 2006.
- [17] D. Ratiu and F. Deissenboeck. Programs are knowledge bases. In *ICPC '06*. IEEE CS, 2006.
- [18] D. Ratiu and J. Juerjens. The reality of libraries. In *CSMR '07*. IEEE CS, 2007.