Evaluating the Reference and Representation of Domain Concepts in APIs

Daniel Ratiu
Institut für Informatik
Technische Universität München, Germany
ratiu@in.tum.de

Jan Jürjens
Department of Computing
The Open University, Great Britain
http://www.jurjens.de/jan

Abstract

As libraries are the most widespread form of software reuse, the usability of their APIs substantially influences the productivity of programmers in all software development phases. In this paper we develop a framework to characterize domain-specific APIs along two directions: 1) how can the API users reference the domain concepts implemented by the API; 2) how are the domain concepts internally represented in the API. We define metrics that allow the API developer for example to assess the conceptual complexity of his API and the non-uniformity and ambiguities introduced by the API's internal representations of domain concepts, which makes developing and maintaining software that uses the library difficult and error-prone. The aim is to be able to predict these difficulties already during the development of the API, and based on this feedback be able to develop better APIs up front, which will reduce the risks of these difficulties later.

1 Introduction

Libraries are today the most important form of software reuse. As reuse is highly desirable, the big projects depend usually on many libraries. Since successful projects live much longer than their initial development, libraries play an important role along the software development life-cycle, from design and development to the evolution and maintenance phases. Once the code is written, it is read many times during many years and by many programmers. Together with the code written within the project, maintainers read and have to understand a significant amount of APIs. Thus, *beside their function as interface between the humans and machines, the libraries play an important role in communicating among programmers*. When the developers work with an API they have to work with the implementation of the domain concepts rather than with the concepts themselves. Thus, libraries enforce the usage of a certain conceptualization of the problem domain and thereby define a "reality" in which programmers have to live and to which they have to adapt (Section 2). In this paper we propose a framework to describe the implementation of concepts in an API along two directions: the manner in which they are referenced and the manner in which they are represented within the API (Section 3). The reference of concepts directly affects the explicitness and intentionality of the API and thereby the capacity of its users to find the corresponding implementation of a concept; the representation of concepts affects the domain appropriateness and the capacity of users to manipulate and combine the (implementations of) concepts at the API level. In Section 4 we use the framework to characterize and quantify how explicit are the domain concepts referenced and how faithful are they represented in an API. We exemplify our framework with parts of the Java standard API. Section 5 presents the related work and Section 6 concludes the paper.

2 APIs quality through practitioners eyes

Even if the quality of APIs dramatically affect the success of software projects, there is a surprisingly few literature that deals with characterization and quantification of APIs usability. In the same time, searching the literature addressed to programmers, we can find a plethora of informal, practitioners-oriented guidelines published in form of developer sites columns or blogs. Below we present examples of guidelines that deal with the API usability:

– According to Mathieu Jacques [6] an usable API should exhibit: good visibility – the users should easily see what it can be used for; good conceptual model – the use of helpful, consistent and complete abstractions representing concepts that the users are familiar with and that help them create a proper mental model of the system; and a good mapping – use of natural analogies for representing the concepts, actions and results.

– Bill Venners [14] compares APIs with real-life machines. The interface of machines has two parts: shape and semantics. The shape represents how does the user see the machines' parts and the semantics represents the meaning of these parts. In the case of APIs, the shape is given by the names of its types, fields and methods and parameters. The

978-0-7695-3176-2/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPC.2008.11

IEEE computer society

semantics of APIs is given by the domain concepts implemented by the API.

Systematizing the practitioners knowledge. Each of the above authors emphasize on the simplicity and coherency of the conceptual model of the API and on the importance that this model is in a direct analogy to the domain. The names of program elements are an important communication mechanism between developers and the simplicity of names are healthy signs for the good conceptual weight of the API. In the following we synthesize the above quality desiderata through the following questions:

Q1) What is the explicitness of the implementation of concepts in the API? and *What is the conceptual complexity of the API?* These questions address the situations when many concepts are combined with each other and the user can access only these combinations. Thus, instead of explicitly accessing, manipulating and combining individual concepts, the API users need to deal with the overhead of understanding the pre-defined combinations of concepts (Section 4.1).

Q2) How uniform can the API users combine the concepts at the API level? and *How difficult is it to make errors by realizing combinations that make no sense?* These questions address the correspondence between the API representation of concepts and the concepts themselves. Depending on how are the individual concepts represented, the API can allow or prevent combinations of concepts that would be meaningful and desired in the modeled domain (the right call is available in the right place). Furthermore, too much freedom enables combinations of program elements that do not make sense from the point of view of the domain knowledge (Section 4.2).

3 From Domain Concepts to APIs

In order to tackle the above questions we need to characterize the APIs from the point of view of how explicit and consistent they reference and represent the domain concepts. To do this, we need an adequate model of APIs that takes into account explicitly the concepts that they implement.[1]

3.1 APIs Formalization in a nutshell

In the following, we adapt our formalization of programs [12] to describe APIs. For assuring a consistency with our previous work on mapping programs to ontologies, we present in the following an overview over our framework. We use a layered representation of APIs that comprises in a unified model the API program elements (program layer) and the concepts that they refer to (conceptual layer). The link between the conceptual and the program worlds is done with the help of similarities between the names of concepts and of program entities (lexical layer). In Figure 1a we present an example of the API layers.

Program layer. We describe an API as a labeled directed graph whose nodes are program elements accessible to the APIs users (e.g. public classes, public methods) and whose arcs are program relations among these program elements (e.g. 'hasType' between a variable and its type). An intuitive example of the program graph can be visualized in Figure 1a (down-right). A complete formalization of this layer can be found in [12]. For the purpose of this paper it is enough to see the program layer only as a set of program elements P that are accessible through the API's interface. We will use in this paper only the relation 'hasType'. In order to keep our presentation concise we do not take into account the program elements that belong to the public interface of the inherited classes.

Conceptual layer. We specify the domain of an API through a domain ontology described through a set of triples of the form: "concept – relation – concept" [12]. These triples can be represented as a labeled graph whose nodes are the concepts and edges are relations among these concepts (e.g. 'isA' is a relation between a sub-ordinate and its super-ordinate concept). An intuitive example of the program graph can be visualized in Figure 1 (up-right). For the purpose of this paper it is enough to see the conceptual layer only as a set of concepts C that are accessible through the API's interface.

Lexical layer. The similarity between the names of program elements and of concepts is the most important glue between the program and the conceptual layers. The library's vocabulary (I) is represented by the set of the names of the program elements from P. The names of the concepts from C are captured by the set N. The lexical layer is centered around a set of lexically normalized words (W) that are the link between the identifiers and concepts names. The lexical layer represents the "skin" of the library and is used by the library developers to communicate with the library users.

3.2 Reference and representation of concepts

Referencing Concepts. A program element can refer to more concepts and a concept can be referred by more program elements. This many-to-many relation between the API program elements and the concepts that they refer to is captured through the following two functions:

$$\overrightarrow{Ref} : C \to \mathcal{P}(P), \quad \overleftarrow{Ref} : P \to \mathcal{P}(C)$$
$$\overrightarrow{Ref}(c) = \{p \in P \mid c \in \overleftarrow{Ref}(p)\} \tag{1}$$

Given a concept $c \in C$, $\overrightarrow{Ref}(c)$ represents the program elements from the public interface that refer to c. Analogous, given a program element $p \in P$, $\overleftarrow{Ref}(p)$ represents

[1] Notation: in order to distinguish between program elements and concepts we will use in the textual part of our paper the following typographical convention: the CONCEPTS are written with small-caps and the `program elements` are written with type-writer fonts.

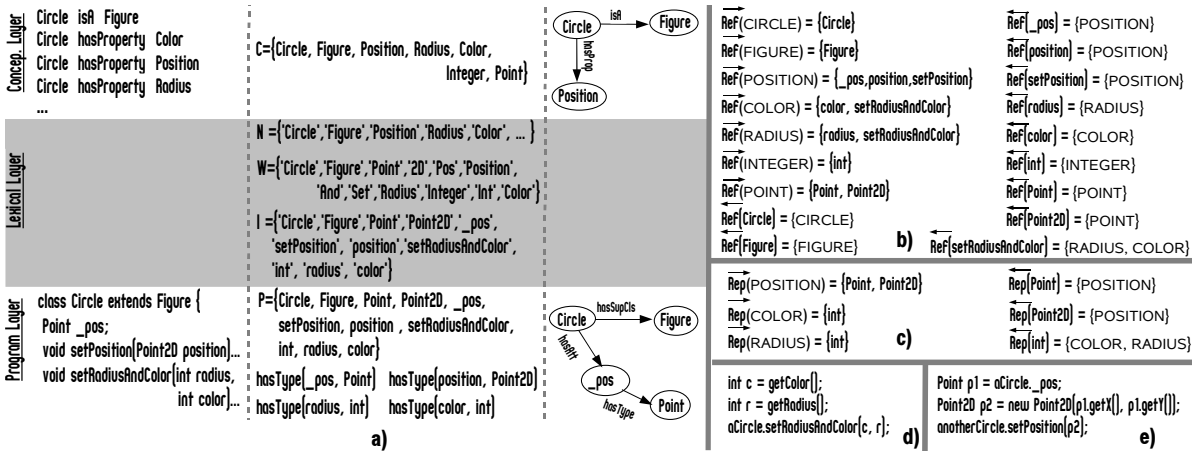

Figure 1. Example of API layers (a); Reference and representation of concepts (b,c); Examples (d,e)

the set of concepts that this program element refers to. We can understand the $\overleftarrow{Ref}$ function to be the knowledge of an API guru that knows exactly what concepts does a program element refer to. In Figure 1b are examples of the functions $\overrightarrow{Ref}$ and $\overleftarrow{Ref}$. For example, the concept COLOR is referred two times in the API: by the method `setRadiusAndColor` and by the parameter `color`.

Representing concepts. In order to manipulate the concepts, an API provides representations for them. In the following we define two representation functions for the concepts referred through variables (attributes or parameters). We consider that a concept is represented in the API through the type of the variable that refers to it (cf. Equation 2). There is a many-to-many correspondence between the concepts and the data types that are used to represent them: a concept can be represented in different API parts through different data types and a data type can be used to represent more concepts. We capture this correspondence with the help of the following two functions:

$$\overrightarrow{Rep} : C \rightarrow \mathcal{P}(P), \quad \overleftarrow{Rep} : P \rightarrow \mathcal{P}(C)$$

Given a concept $c \in C$, $\overrightarrow{Rep}(c)$ are the data types from the public interface that are used to represent it in the API. For a data type $p \in P$, $\overleftarrow{Rep}(p)$ are the concepts that are represented by p. The definition of these functions is:

$$\overrightarrow{Rep}(c) = \{p \in P \mid \exists p' \in P.\, \overleftarrow{Ref}(p') = \{c\} \wedge hasType(p') = p\} \quad (2)$$

$$\overleftarrow{Rep}(p) = \{c \in C \mid p \in \overrightarrow{Rep}(c)\} \quad (3)$$

In Figure 1c are examples of $\overrightarrow{Rep}$ and $\overleftarrow{Rep}$ – for example, the concept POSITION is represented through the classes `Point` and `Point2D` and the primitive type `int` is used to represent the concepts COLOR and RADIUS. In Figure 1d,e are presented two examples of anomalies which originate from using the same representation for two different concepts or using different representations for the same concept. In the first case (d) the API users can easily make uncaught mistakes: the value of a color is used to set the radius and vice-versa. In the second case, not wanted conversions are necessary in order to transform between representations.

Observations: **1)** While the reference of concepts affect their identification in the library, the representation of concepts affect the manners in which the API clients can manipulate and compose the concepts at the API level. The reference of concepts represent the classical problems of concepts location ($\overrightarrow{Ref}$) [10] and concepts assignment ($\overleftarrow{Ref}$) [2]. The representation of concepts is the manner in which the API regards the concept internally. From the practitioners point of view (Section 2), the reference represents the shape of the API and the representation is (part of) the semantics of the API since it enables (or not) possible combinations of the API parts. In our example, the shape of our API is given by concepts such as COLOR, CIRCLES, FIGURE and the semantics through program abstractions such as `int`, `Point` and `Point2D`.

2) All the functions defined above can be expressed in terms of $\overleftarrow{Ref}$ (cf. Equations 1-3). Therefore, the key for automatizing the computation of $\overrightarrow{Ref}$, $\overrightarrow{Rep}$ and $\overleftarrow{Rep}$ is the automation of $\overleftarrow{Ref}$. In our previous work [12] we proposed a method for automatically computing $\overleftarrow{Ref}$ by mapping the source code to ontologies. Unfortunately, in the majority of situations there is no off-the-shelf ontology available that is fit for locating concepts in the code. However, the metrics that we propose in the next section give only hints about the API problems and therefore rougher approximations for $\overleftarrow{Ref}$ can be used (e.g. by considering that there is an exact correspondence between the words contained in identifiers and the domain concepts).

4 Evaluation of APIs

In the following we use our framework to characterize the reference (Section 4.1) and representation (Section 4.2) of concepts in APIs and thereby to tackle the questions from Section 2. We illustrate each characteristic and the computation of each metric with the API example from Figure 1. In the case of representation defects we present as examples two fragments of the Java standard library.

4.1 Characterizing the reference

Reference explicitness. A concept $c \in C$ is *explicitly referred* in the API through a program element $p \in P$ iff:

$$\overleftarrow{Ref}(p) = \{c\}$$

In order to maintain the correspondence between the domain concepts and the program elements from the API, every domain concept should have an explicit implementation in the API through a program element that refers only to it. The concepts that are not explicitly referenced in the API are available to the API users only in combination with other concepts and this influences negatively the visibility of concepts at the API level. For example, the concept POSITION is explicitly referenced through the parameter `position` and the attribute `_pos` (Figure 1).

Reference explicitness ratio. The *reference explicitness ratio (ER)* for an API is:

$$ER = \|\{p \in P \mid \exists c \in C.\ \overleftarrow{Ref}(p) = \{c\}\}\| \ / \ \|P\|$$

Intuitively, we characterize the reference explicitness for an entire API by computing the ratio of program elements that refer to single concepts. ER take values between zero and one – the higher the ER (closer to one), the more explicit is the API as a whole and thereby the concepts have a good visibility. For example, the explicitness ratio for the code fragment from Figure 1, is ER = 0.91 (= 10 / 11). From Figure 1b we can see that out of 11 program elements accessible through the API, 10 reference a single concept.

Conceptual complexity. The *conceptual complexity (CC)* of an API is:

$$CC = \sum_{p \in P} \|\overleftarrow{Ref}(p)\| \ / \ \|P\|$$

Intuitively, the conceptual complexity of an entire API is the average of the number of concepts referenced by program elements. The CC is always bigger as one since each program element refers to at least one concept. In an ideal case, CC = 1 and this means that every program element references a single concept. The bigger the CC the more complex combinations of concepts are used in the API. Complex combinations of concepts are hard to understand and the program elements implementing them are useful only in special cases. For example, the conceptual complexity for the code fragment from Figure 1, is CC = 1.09 (= 12 / 11). From Figure 1b we can see that out of 11 program elements, one reference two concepts and the others reference only one concept.

4.2 Characterizing the representation

Representation faithfulness. A concept $c \in C$ is *faithfully represented* iff:

$$\overrightarrow{Rep}(c) = \{p\} \ \wedge \ \overleftarrow{Rep}(p) = \{c\}$$

Intuitively, a type is used in the API to represents only one concept and each concept is represented through only one type. This is an ideal case when an API offers distinct representations for distinct concepts and the same representation whenever a concept is referenced more times in the API. Failing to do this makes the combination of the concepts at the API level unnatural (Figure 1d,e). In the next paragraphs we quantify the representation anomalies.

Representation overloading. The type $p \in P$ exhibits *representation overloading* iff:

$$\exists c_1, c_2 \in C.\ p \in \overrightarrow{Rep}(c_1) \ \wedge \ p \in \overrightarrow{Rep}(c_2) \ \wedge \ c_1 \neq c_2$$

Intuitively, a type exhibits an overloaded representation if it is used to represent more concepts (i.e. there are several variables with this type, each variable refers to a single concept (c_1 and c_2) and the concepts referred by these variables are distinct $c_1 \neq c_2$). In the practice, many times the APIs do not offer distinct representations for concepts. Instead, several concepts are represented through the same type. The more concepts are represented through a single type, the more freedom in composing concepts at the API level we have and the easier it is for the library users to make uncaught logical mistakes. In this case, the APIs are easy to misuse. For example, in the case of the API fragment from Figure 1 the type `int` exhibits representation overloading since it is used to represent both the COLOR and RADIUS. As explained in the previous section, this fact leads to unwanted combinations of concepts at the API level (Figure 1d).

A dangerous case of overloading is when two parameters of a method have the same type. Then, the language mechanisms enable the API clients to call the method with exchanged parameters and thereby to introduce bugs. In Figure 2 we present an example of two parts of the Java library that implement functionality related to class loading. In this code fragment the type `String` represents an overloaded representation since it is used to represent both the CODEBASE address as well as the NAME of the

class to be loaded. Furthermore, the order of parameters of this two methods is changed: the first parameter of the `RMIClassLoader.loadClass` has the meaning of the second parameter of `Util.loadClass` and vice-versa. Due to this fact, there are very high chances of misuse of the `RMIClassLoader` by the programmers that already used the `Util` class and vice-versa.

```
package java.rmi.server;
public class RMIClassLoader { ...
  public static Class loadClass(String codebase,
       String name, ClassLoader defaultLoader) {...}
... }
```

```
package javax.rmi.CORBA;
public class Util { ...
  public static Class loadClass(String className,
    String remoteCodebase,  ClassLoader loader) {...}
... }
```

Figure 2. Dangerous overloading example

Overloading degree. We define the *overloading degree (OD)* for a type $p \in P$ to be:

$$OD(p) = \|\overleftarrow{Rep}(p)\|$$

Intuitively, the overloading degree quantifies the number of concepts represented through the same type. We use this metric to characterize the measure in which a type $p \in P$ is overloaded. In the case of types that do not represent any concept ($\overleftarrow{Rep}(p) = \emptyset$) the value of OD is zero, otherwise the value of this metric is $\geq$ than one; in the case when OD(p) = 1 then the type exhibits no overloading. For example, in the case of the API fragment from our example from Figure 1 the overloading degree of `int` is 2 since it is used to represent two concepts.

In the case of the Java SWING library the types with the highest overloading degree are: `int` and `String`. The very high values for the overloading degree of basic types show a high level of underspecification. For example, in the case of `int` our manual inspection revealed that it is used to represent a wide variety of concepts such as positions, sizes, indexes, constants, periods and so on. For many situations, even at first sight the type `int` offers an appropriate representation, the negative values make no sense from the point of view of the concepts that it represents. Furthermore, it allows the combination of concepts (e.g. assignments) that are completely non-sense from the point of view of the modeled domain.

Overloading ratio. For an API, the *overloading ratio (OR)* is defined as:

$$OR = \sum_{p \in P} OD(p) \; / \; \|\{p \in P \mid OD(p) > 0\}\|$$

Intuitively, the overloading ratio measures the average of the overloading degree for an entire API. OR can take values only bigger or equal with one. In the case when OR = 1 then the API represents the concepts ideally and has no overloading. The higher the overloading ratio the higher the underspecification of an API is. For example, in the case of the API fragment from our example from Figure 1, OR = 1.5 (3 / 2) since there are two types which are used to represent three concepts.

Representation ambiguity. A concept $c \in C$ exhibits *representation ambiguity* iff:

$$\exists p_1, p_2 \in P.\; p_1 \in \overrightarrow{Rep}(c) \;\wedge\; p_2 \in \overrightarrow{Rep}(c) \;\wedge\; p_1 \neq p_2$$

Intuitively, a concept is ambiguously represented in the API if it is represented through distinct types. Consequently, the library users cannot use uniformly this concept at the API level and many times they should convert between its different representations. In this manner is introduced a significant redundancy and an encoding bias that negatively affects the API users. For example, in the case of the API fragment from our example from Figure 1, the concept POINT exhibits representation ambiguity since there are two distinct types used to represent it: `Point` and `Point2D`. As explained in the previous section, this fact leads to additional, not wanted complexity in combinations of concepts at the API level (Figure 1e).

In Figure 3 we present an example of ambiguity in the Java SWING API: the concepts that refer to cardinal points are represented both as strings and as integers. Thereby, the SWING users can not work with positions uniformly and need to convert between different types of the representation (here `int` and `String`).

```
package javax.swing;
public class SpringLayout {
   public static final String NORTH = "North";
   public static final String SOUTH = "South";
}
```

```
package javax.swing;
public class SwingConstants {
   public static final int NORTH = 1;
   public static final int SOUTH = 5;
}
```

Figure 3. Ambiguity example

Ambiguity degree. Given a concept $c \in C$, we define the *ambiguity degree (AD)* to be:

$$AD(c) = \|\overrightarrow{Rep}(c)\|$$

Intuitively, the ambiguity degree is defined as the number of distinct representations used for a single concept. For a concept $c \in C$ that is not represented in the API (i.e. $\overrightarrow{Rep}(c) = \emptyset$) the AD(c) = 0; otherwise, AD is bigger or equal to one. In the case of a concept $c \in C$ for which AD(c) = 1 then this concept is unambiguously represented. We use AD to quantify the ambiguity in the representation of a concept. For example, in Figure 1, AD(POSITION)=2 while the concept POSITION is ambiguously represented through the types `Point` and `Point2D`.

Ambiguity ratio. For an API, the *ambiguity ratio (AR)* is defined as:

$$AR = \sum_{c \in C} AD(c) \; / \; \|\{c \in C \mid AD(c) > 0\}\|$$

Intuitively, the ambiguity ratio is the average of the ambiguity degrees of all concepts that are represented in the API (i.e. $\overrightarrow{Rep}(c) \neq \emptyset$). AR can take values ≥ 1, in the case when $AR = 1$ then the API exhibits no representation ambiguity. For example, in the case of the API fragment from Figure 1, AR = 1.3 (4 / 3) since there are three concepts represented in the API and AD(POSITION) = 2, AD(COLOR) = 1 and AD(RADIUS) = 1.

5 Related Work

Design of APIs. The importance of quality of APIs was recognized in the early '90s [8] in the work of Meyer who presents a set of lessons learnt from building the Eiffel library. He emphasizes on consistency in choosing and combining the library features and in the quality of names. [9] also discusses a set of high-level library quality attributes like: uniformity of component interfaces or functional completeness and 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". Korson [7] identifies an extensive set of desirable attributes of software libraries such as: libraries should be 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). In this paper we provide an analytical method for evaluating aspects of the conceptual complexity and consistency of APIs and thereby for guiding the APIs designers towards more usable APIs.

Evaluating APIs usability. Despite the high importance of the APIs usability, we could found only relative few works which deal with this subject. There is a series of articles [3, 5, 13] that evaluate API usability by matching the characteristics provided by the API to the ones expected by its users. They perform user interviews based studies on how well does the library support different categories of users in writing programs for typical use-cases that involve the library. In this paper we evaluate aspects of the APIs usability by expressing the informal API design guidelines using a unified framework that comprises both the program elements accessible to the API users and a conceptual model of the API's domain.

Concepts location. Locating the implementation of a domain concept in the code (concepts location) and linking a part of the sources to the concepts that it implements (concepts assignment) is one of the most important tasks in program comprehension [2, 10]. The essential role of identifiers as glues that link the program to the domain concepts is presented in [1, 4]. With the extensive use of libraries in software projects, a significant part of the program understanding represents in fact APIs understanding. In this paper we base on our previous work in program understanding [11, 12] to characterize and quantify the reference and representation of concepts in APIs and thereby to improve their usability.

6 Conclusion and Future Work

In this paper we propose a framework to quantify the reference and representation of concepts in APIs by explicitly linking the program elements with the concepts that they implement. The current work represents only a first step in providing methods for analytically expressing and quantifying usability aspects of APIs. We envision two main directions of future work: the first regards an empirical validation of this framework by analyzing more APIs and an extension of our framework with other measurements; the second direction concerns the usage of the experiences distilled from these analyses in the forward engineering in order to timely recognize the API smells and thereby improve the development of new APIs.

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] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE*. IEEE CS, 1993.

[3] S. Clarke. Measuring API usability. *Dr. Dobbs Journal Windows/.NET Supplement*, pages 7–9, 2004.

[4] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3), September 2006.

[5] B. Ellis, J. Stylos, and B. Myers. The factory pattern in api design: A usability evaluation. In *ICSE*, pages 302–312. IEEE CS, 2007.

[6] M. Jacques. API Usability: Guidelines to improve your code ease of use. *The Code Project, http://www.codeproject.com*.

[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] B. Meyer. Lessons from the design of the Eiffel libraries. *Commun. ACM*, 33(9):68–88, 1990.

[9] B. Meyer. *Reusable software: the Base object-oriented component libraries*. Prentice-Hall, Inc., 1994.

[10] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *IWPC*. IEEE CS, 2002.

[11] D. Ratiu and F. Deissenboeck. Programs are knowledge bases. In *ICPC*. IEEE CS Press, 2006.

[12] D. Ratiu and F. Deissenboeck. From reality to programs and (not quite) back again. In *ICPC*. IEEE CS Press, 2007.

[13] J. Stylos and S. Clarke. Usability implications of requiring parameters in objects' constructors. In *ICSE*, 2007.

[14] B. Venners. Think of Objects as Machines. *Artima Developer, http://www.artima.com/*, 2003.