Pontifícia Universidade Católica
do Rio de Janeiro

Luiz Marques Afonso

Communicative Dimensions of Application Programming Interfaces (APIs)

TESE DE DOUTORADO

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio as partial fulfillment of the requirements for the degree of Doutor em Informática.

Advisor: Prof. Clarisse Sieckenius de Souza

Rio de Janeiro
April 2015

Pontifícia Universidade Católica
do Rio de Janeiro

Luiz Marques Afonso

Communicative Dimensions of Application Programming Interfaces (APIs)

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Doutor.

Rio de Janeiro, April 6th, 2015

Luiz Marques Afonso

Luiz Marques Afonso received his bachelor degree in Informatics from the Federal University of Rio de Janeiro (UFRJ) in 1996, and the MSc degree in Informatics from PUC-Rio in 2008. He is a software developer with many years of experience in programming, design, and software security. He works for 3Elos, an IT company specialized in Information Security, where he is also a partner.

Bibliographic Data

Afonso, Luiz Marques

Communicative Dimensions of Application Programming Interfaces (APIs) / Luiz Marques Afonso ; advisor: Clarisse Sieckenius de Souza. — 2015.
153 f. : il. (color.); 30 cm

Tese (doutorado)-Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2015.
Inclui bibliografia

1. Informática – Teses. 2. Design e avaliação de interfaces de programação. 3. Engenharia Semiótica. I. de Souza, Clarisse Sieckenius. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To Elen and Maria.

Acknowledgments

First and foremost, I am grateful to my wife, Elen, for the love and patience during all these years of hard work. Many thanks to my family, especially my mother, for the love, support and encouragement.

I would like to thank my thesis advisor, Prof. Clarisse Sieckenius de Souza, for the dedication, insight, patience and for the always sharp comments, suggestions and questions about my work. I am also grateful to her for willing to advise me when I was already in the course of my PhD years. It has been a privilege and a pleasure to work closely with such an awarded researcher and, as the PhD years showed me, a remarkable person.

I am grateful to Prof. Renato Cerqueira for his special collaboration to this work and for advising me in the first half of the course. I also thank him for inviting me to apply for the PhD at PUC-Rio, after receiving my MSc degree in 2008, when he was my advisor.

Special thanks to my friends at 3Elos, Marcelo Duarte and Marcelo Pettengill, for supporting my academic endeavor.

I would also like to thank the people from the Semiotic Engineering Research Group (SERG), for the many discussions over the course of our workshops and meetings: Bruno Chagas, Cátia Ferreira, Cleyton Slaviero, Eduardo Tolmasquim, Ingrid Monteiro, João Bastos, Luciana Salgado, Marcelle Mota and Priscilla Abreu. Special thanks to Carla Leitão for her precise methodological suggestions and comments, and also to Juliana Ferreira and Rafael Brandão, for the close collaboration.

I would also like to thank PUC-Rio and CAPES for the partial financial support.

Abstract

Afonso, Luiz Marques; de Souza, Clarisse Sieckenius (Advisor). **Communicative Dimensions of Application Programming Interfaces (APIs)**. Rio de Janeiro, 2015. 153p. Doctoral Thesis — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Application programming interfaces (APIs) have a central role in software development, as programmers have to deal with a number of routines and services that range from operating system libraries to large application frameworks. In order to effectively use APIs, programmers should have good comprehension of these software artifacts, making sense of the underlying abstractions and concepts by developing an interpretation that is compatible with the designer's intent. Due to the complexity of today's systems and programming environments, learning and using an API properly can be non-trivial task to many programmers. Traditionally, studies on API design have been developed from a usability standpoint. These studies have provided evidence that bad APIs may affect programmer's productivity and software quality, offering valuable insights to improve the design of new and existing APIs. This thesis proposes a novel approach to investigate and discuss API design, based on a communication perspective under the theoretical guidance of Semiotic Engineering. From this perspective, an API can be viewed as a communication process that takes place between designer and programmer, in which the former encodes a message to the latter about how to communicate back with the system and use the artifact's features, according to its design vision. This approach provides an account of API design space that highlights the pragmatic and cognitive aspects of human communication mediated by this type of software artifact. By means of the collection and qualitative analysis of empirical data from bug repositories and other sources, this research work contributes to a deeper comprehension of the subject, providing an epistemic framework that intends to support the analysis, discussion and evaluation of API design.

Keywords

Application programming interface design and evaluation. Semiotic Engineering.

Resumo

Afonso, Luiz Marques; de Souza, Clarisse Sieckenius. **Dimensões Comunicativas de Interfaces de Programação (APIs)**. Rio de Janeiro, 2015. 153p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Interfaces de programação, ou application programming interfaces (APIs) possuem um papel chave no desenvolvimento de software, uma vez que os programadores precisam lidar com diversas rotinas e serviços, desde bibliotecas de sistemas operacionais a frameworks de aplicação complexos. Para o uso efetivo de uma API, programadores devem ter uma boa compreensão do artefato de software, suas abstrações e conceitos subjacentes, desenvolvendo uma interpretação compatível com a intenção do designer. Devido à complexidade dos sistemas e ambientes de programação atuais, aprender e usar adequadamente uma API pode ser uma tarefa não trivial para muitos programadores. Tradicionalmente, estudos sobre o design de APIs foram desenvolvidos sob uma perspectiva de usabilidade. Esses estudos geraram evidências de que o projeto inadequado de uma API pode ter impacto sobre a produtividade de um programador e sobre a qualidade do software, e colaboraram para incrementar o design de APIs novas ou já existentes. Esta tese propõe uma nova abordagem para investigar e discutir design de APIs, baseada numa perspectiva de comunicação sob a orientação teórica da Engenharia Semiótica. Nessa perspectiva, uma API pode ser vista como um processo de comunicação que ocorre entre o designer e o programador, no qual o primeiro codifica uma mensagem para o segundo sobre como ele deve se comunicar com o sistema e usar as suas funcionalidades, de acordo com a visão de design. Essa abordagem provê uma caracterização do espaço de design de APIs que enfatiza os aspectos pragmáticos e cognitivos da comunicação humana mediada por este tipo de artefato de software. Através da coleta e da análise qualitativa de dados empíricos de repositórios de bugs e outras fontes, essa pesquisa contribui para uma compreensão mais ampla sobre o tema, provendo um framework epistêmico que pode ser usado no apoio à análise, discussão e avaliação do design de APIs.

Palavras–chave

Design e avaliação de interfaces de programação. Engenharia Semiótica.

Contents

List of Figures

List of Tables

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Martin Fowler, *Refactoring: Improving the Design of Existing Code.*

1 Introduction

Abstraction is a key concept in Computer Science (1, 2), especially in the context of software construction and use. It is a cognitive resource that allows us to concentrate on the core properties of a complex object or concept, simplifying things by removing excessive details. Most software systems are composed by a large number of layered abstractions, and current applications contain highly complex and sophisticated abstractions. However, in the lowest level, they are all encoded in a binary representation of simple, executable steps to be interpreted by a computer.

Despite being executed by a machine, software has its creation and use processes deeply based on human interpretation, since most software is written *by* humans and *for* humans. The construction of a software artifact implies a set of design intentions that envision (at least part of) the users' expectations, goals, needs and limitations. Since each user is unique, the meanings extracted from the abstractions contained in the artifact's representation may show great diversity. As such, the design of a software artifact should effectively communicate these underlying intentions in order to conduct the users' interpretation process in the direction of a compatible mental model.

Programmers belong to a specific class of users, which deal with a particular type of software artifacts (compilers, interpreters, libraries, components, etc.). Their goal is to use these artifacts to construct other pieces of software. However, the 'general problem' also applies to this more specific context: programmers should have a good comprehension of the underlying abstractions contained in the design of these artifacts, and be able to articulate them in order to construct their own sets of abstractions.

Therefore, human aspects should be frequently addressed by research work comprehending software construction, its related tools and processes. However, a significant part of scholarly studies in Software Engineering and Programming Languages are completely dissociated from this subject (3).

1.1 Context

Reuse is a common technique used by software engineers to speed development time through the use of preexisting software modules. It may also help to reduce the number of defects by using stable and tested artifacts, especially if we consider the growing complexity of today's systems. In general, these modules can be used via its interfaces, which allow other pieces of software to call the available operations and create new functionalities on top of the existing ones. These interfaces are generally known as application programming interfaces, or APIs.

In this text, we refer to an API as any software interface that provides a set of semantically related operations and data, usually associated with a specific domain. Software components, modules, libraries and frameworks usually provide APIs that expose services to be used by other software elements. This definition is similar to the one adopted by de Souza et al. (4) in their work regarding the study of APIs in the context of cooperative work.

APIs are the 'window' that exposes the services provided by software components, modules, libraries and other types of routines. Currently, they play a central role in the programming scenario, since most popular languages provide an extensive library of data structures and algorithms to professional users. As such, it is not usual to write a program 'from scratch' anymore, even a short one. A programmer's job is, to a large extent, to find adequate modules, components or functions that provide the required functionalities and to combine them appropriately in order to accomplish her task. Also, the ubiquity of Web applications, interoperating in different forms like mashups, blogs, social networks, services, and so on, takes the use of APIs to a much higher number of client users.

Every seasoned programmer has probably faced difficulties when trying to learn how to use a new software module interface, accompanied by a feeling that 'it could have been easier'. Complex APIs may discourage their adoption, since the effort to use it effectively may overcome the benefits. Also, badly designed APIs may hinder programmers' comprehension, leading to all sorts of defects due to misunderstandings. Henning (5) provides an example of how an overly complex API may have a severe impact on its success. In another work (6), the same author emphasizes that the importance of API design is growing "not only because we are designing more APIs, but also because these APIs tend to provide access to much richer and more complex functionality". He also claims that "good APIs are hard", and "bad APIs are easy".

Designers encode their intention behind an API by means of its lexical, syntactical and semantic characteristics, as well as by its runtime behavior and documentation. Programmers need to interpret the concepts underlying the interface design in order to use them effectively. In this process, they construct their own meanings regarding the artifact's features, trying to grasp the designer's 'message'. From a human-centric perspective, this can be viewed as a communication process taking place between these parties (designer and programmer), mediated by the software artifacts involved.

The learning and interpretation processes commonly associated with the use of a new API may impose a considerable amount of cognitive load on the programmer, depending on the abstractions involved and on the design of the artifacts provided. The higher this load is, the higher is the intellectual effort needed, which may increase the error-proneness of programming tasks. The variety of notations involved in the representation of an API and its resources may also influence this cognitive process of learning and using an API and its associated language.

In this context, the design and use of an API are strongly influenced by the characteristics of the underlying programming language. From a design perspective, the programming language determines the expressiveness available for the designer to choose identifiers, types, parameters, returns, exceptions, and so on. The semantics of the language is also important here, as it defines how the chosen representation will behave at runtime, linking design intent with actual results. From the user's side, the language's notation and the choices made by the designer may influence the user's interpretation of the intent behind the artifact, positively or not. The programming culture associated with the language and its community of programmers should also be taken into consideration when analyzing the interpretation process of the design intent behind an API.

There are various resources concerning expert knowledge and guidelines to help the design of an API, e.g. (7, 8, 9). However, each API is unique, having its own particular characteristics and goals. There can be subtleties in the design process that make some decisions critical to its completeness and flexibility. The work by Ierusalimschy et al. (10) provides interesting insights about how the design of the API for embedding Lua code in C programs influenced the design of the Lua language, and vice-versa.

There is a growing demand for software for all kinds of purposes, ranging from large enterprise systems to simple applications to automate a single task. To keep up with this demand, a larger community of programmers is being formed by both professional and end user developers. The increasing number of non-professional programmers add a different perspective to the discussion of API design, since they have less (or no) formal education in Computer Science, and less programming background. However, they have problems to solve and tasks to automate, turning them into users of programming languages and API's. Therefore, the characteristics of the intended audience should be taken into account when designing a programming language and its API's.

In the next section, other aspects in the context of language and API design are discussed in order to provide arguments that motivate this research work.

1.2 Motivation

The study of human aspects related to programming has a long tradition in the research community. According to Myers and Ko (11), in the 1980's, there was a "significant amount of work" under different names like Psychology of Programming, Software Psychology, and Empirical Studies of Programming. Despite being less frequently found in the 1990's, these types of research studies restored their importance among academics in the 2000's, and are now a more popular theme in mainstream conferences. More recently, some researchers have explicitly argued in favor of a stronger emphasis on human factors in research studies related to Software Engineering and Programming Languages (12, 3).

Under the influence of this growing importance of human aspects of programming, many topics related to API design and evaluation have been extensively studied in the past decade (13). Some of these studies have been funded by software companies that publish APIs to a large client base, e.g. (14, 7). Their concern originates from the fact that getting a good API design before publishing is mandatory, because post-release fixes are costly and may break legacy code.

Many studies in API design have been developed under a usability or learning context, with a strong emphasis on the programmer side, e.g. (15, 16, 17, 18, 19). Despite the programmer's importance as being the end user of software interfaces, the role of the designer should also be considered in the scope of studies about effective API design. The designer is responsible for the choices that represent the intention behind the artifact implementation, being an active participant in this communication process.

In an interesting study(17), Robillard and Deline report qualitative findings regarding the most significant API learning obstacles, from the perspective of professional developers. Their study reported inadequate API documentation as the most severe obstacle faced by developers when learning a new API.

From this conclusion, the authors list the main implications derived from their analysis of interviews with developers, focusing on API documentation aspects. Among these implications, they argue that intent documentation should be provided when correct usage is not evident, and that small examples showing API usage patterns are more useful than 'single-call' examples. Examples are also useful to illustrate best-practices regarding the artifact's usage. However, they can also hinder user's comprehension, when there is a gap between the abstraction provided by code examples and the usage context required by the programmer. In addition, they state that matching scenarios to API elements are considered helpful, as they may describe more complex API structures and how they fit into the big picture. Finally, they emphasize the relevance of explicit documentation about performance, error handling and triggering, and all sorts of specific API behavior, corner cases and relevant internal aspects of the implementation.

Still in the context of Robillard and Deline's findings, the authors state that some developers "look for a coherent, linear presentation of the documentation", because "fragmented collections of hyperlinked articles can be overwhelming". Most implications of this study reveal the importance of effective communication between API designers and programmers, especially when developing abstractions and concepts contained in a software component. Documentation is only the most obvious element of this communication, but the pragmatic conditions of software reuse may involve a more complex set of communicative and cognitive factors that influence this interaction.

When developing software, a programmer should have a good mental model of the software artifacts being used to compose the solution to a certain task. Without a good understanding of these abstractions, a programmer can introduce subtle errors in the code that will only appear later in the software life cycle. In his PhD thesis (20) about common software defect causes and characteristics, Hovemeyer claims that "API misuse is the single most prevalent cause" of the bug patterns detected. Despite focusing on the Java language, this work is a good example of how API design may have a strong impact on software quality.

In order to illustrate how API design may hinder programmer comprehension and lead to severe bugs, we refer to a short but compelling example of bugs caused by API misunderstandings. It is based on a study of concrete problems in the use of Secure Sockets Layer (SSL) libraries, focusing on non-browser software dealing with digital certificates and security. The next subsection presents this example in more detail.

1.2.1 "The most dangerous code in the world"

In their work (21), Georgiev et al. presented examples of non-browser applications that use SSL libraries insecurely. One of the problems pointed by the authors is that some applications do not validate adequately the server's digital certificate when establishing an SSL connection, making them vulnerable to 'man-in-the-middle' attacks. This means that an attacker may successfully intercept the communication between client and server, by faking the server's identity. Some of the examples presented are related to security-sensitive operations like, for instance, on line payment.

According to the authors, *"The root cause of most of these vulnerabilities is the terrible design of the APIs to the underlying SSL libraries. (...) As a consequence, developers often use SSL APIs incorrectly, misinterpreting and misunderstanding their manifold parameters, options, side effects, and return values"*. From this description, one possible research question is: what characteristics of these APIs may have determined the high rate of security bugs found in their usage ?

The study mentions problems identified in many software packages using SSL functions in different programming languages, like Java and PHP. This section focuses on one of these libraries as an illustrative example, namely the PHP Curl extension[1]. This module is a PHP wrapper of libcurl[2], a widely used C library that offers URL-based data transfer facilities.

The vulnerabilities caused by the misuse of the PHP Curl API are related to a specific characteristic of its design and implementation. Before executing a transfer operation from a server URL, users may have to set a number of configuration options that define the library's behavior. For example, when connecting to a server using an SSL-based protocol (e.g. HTTPS), the server must present a digital certificate to the client. This allows the client to verify the server's identity, if the server's certificate has been issued by a certification authority trusted by the client.

The basics of operations with the PHP Curl API involves three steps: initialization, configuration and execution of the data transfer. The configuration step consists of calling a generic function (curl_setopt) that handles more than a hundred options. The need to set the various options depends on the operation being attempted. For instance, when dealing with an SSL-based connections during development, it is common to 'relax' the library's default behavior, in order to accept connections to servers presenting untrusted cer-

[1]`http://php.net/manual/en/book.curl.php`
[2]`http://curl.haxx.se/libcurl/`

tificates. The code in listing 1.1 illustrates this situation, by turning off the CURLOPT_SSL_VERIFYPEER option and forcing the client to accept any certificate presented by the server.

Listing 1.1: Example of insecure PHP Curl code

```
$ch = curl_init("https://localhost");
curl_setopt($ch,CURLOPT_SSL_VERIFYPEER, false);
if( curl_exec($ch) ) echo "Request OK";
else echo "Error: ".curl_error($ch);
curl_close();
```

By turning off CURLOPT_SSL_VERIFYPEER, the programmer disables the verification of server certificates, rendering the code vulnerable to man-in-the-middle attacks. This configuration setting has a close relationship to another one: CURLOPT_SSL_VERIFYHOST. It controls how the client checks the host name specification in the server's certificate against the host name used in the connection. This means that this second option only makes sense when the first one is enabled.

Although the CURLOPT_SSL_VERIFYPEER setting has boolean semantics (accepts true or false), the second setting (CURLOPT_SSL_VERIFYHOST) expects an integer value, ranging between 0 and 2. The most secure option is 2 (the default), which means full server certificate check. However, it is quite easy to find code excerpts in the Web where this option is set to 1, which is not the most secure option. Probably, this happens because programmers infer that it also has boolean semantics, by making an analogy with the first setting.

This "perversely bad" (21) interface has a side effect related to this inconsistency that may really impact its usability: as the PHP language is weakly typed, *curl_setopt()* accepts any type as the value to set the option. So, the programmer that calls the function to set CURLOPT_SSL_VERIFYHOST with the boolean value *true* is, in fact, having this value automatically converted to the integer value 1, which is not the most secure choice (the default is 2, full security check).

An interesting fact about this case study is its high incidence. A quick Web search returns many examples of vulnerable code snippets, like bug reports[3], open source code[4], blogs[5], and user comments in PHP documentation[6].

[3]https://support.zabbix.com/browse/ZBX-5924

[4]http://code.google.com/p/a-paypal-express-class/source/browse/trunk/Paypal.class.php?spec=svn2&r=2

[5]http://icfun.blogspot.com.br/2008/04/php-curl-to-fetch-html-from-https.html

[6]http://www.php.net/manual/en/function.curl-setopt.php

This issue of the PHP Curl API is a short but representative example of the relevance of effectively communicating design intent behind a programming artifact. It also shows that dealing with this type of inconsistencies and bad design choices may have a cognitive impact on programmers, possibly leading to coding errors or misunderstandings. Motivated by this type of mismatch between API design and the pragmatic conditions of programming, the next section presents the main objectives for this research work.

1.3 Research Goal

The creation and use of programming artifacts comprehend a series of 'interactions' between programmers, at different layers. From system-level to end user programming, software modules provide new abstractions on top of existing ones, allowing other programmers to call these services through their interfaces. Figure 1.1 shows some possible instances of these relationships between programmers.

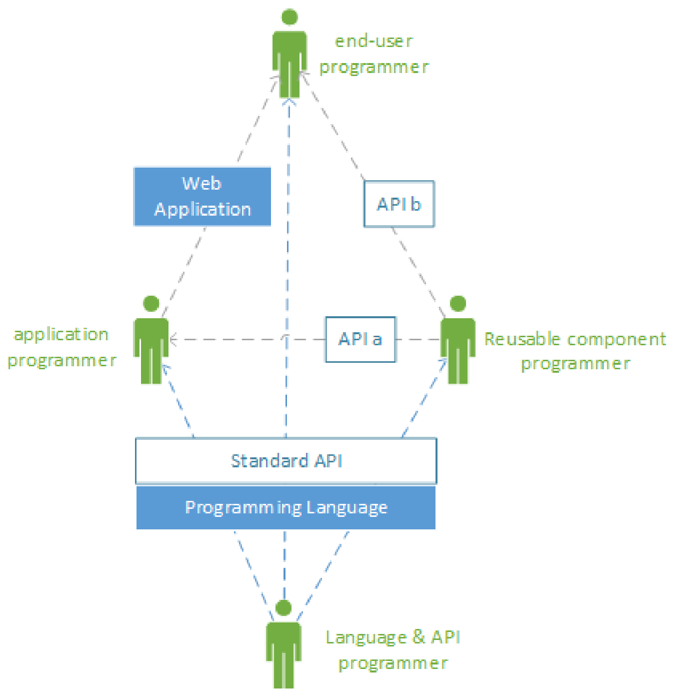


Figure 1.1: Interactions between programmers

In these interactions, software interfaces mediate a communication process between designer (programmer that creates the API) and user (programmer that uses the API). When this process is not effective, there can be breakdowns with different outcomes. For instance, the programmer may develop a wrong mental model of the high-level concepts behind an API's design. This can possibly introduce semantic defects in the code, since the programmer won't be able to articulate properly the API's abstractions to accomplish the intended task. Even worse, the programmer may be unable to write any code by simply not knowing what to do with the API's primitives.

Another possible outcome is the artifact's misuse, when the programmer develops a compatible mental model of the underlying abstractions, but fails to code a correct implementation of the solution. For example, the programmer may write code to call a function passing wrong parameters, caused by not fully understanding the operation's contract.

In order to be effective, software reuse implies a programmer being able to articulate the abstractions contained in the artifacts in a timely (and possibly pleasant) fashion, solving the intended problem within a reasonable limit of difficulties and errors. There is empirical evidence that certain aspects of programming languages and APIs impose a greater difficulty on programmers, leading to the introduction of errors in code, or misinterpretation of what is represented in code. Also, there can be discrepancies between users' and designers' conceptual, cultural and technical stances, which can also be a source of breakdowns in the communication that takes place between them.

Based on the context and motivation presented so far in this text, the main objective of this research is to investigate how the design of APIs and programming languages may influence users' success (or failure) in the attempt of reading and writing code to perform a programming task. This investigation focuses on a human-centric perspective of the communication process mediated by these artifacts, and the cognitive impacts associated with this process.

This objective can be summarized in the following research question: *How communicative and cognitive aspects of the design of programming languages and APIs influence their usage by programmers, leading to errors or difficulties in writing and understanding code ?*

Based on the possible outcomes of the search for an answer for this question, a secondary objective of this work is to provide epistemic tools [7] and a vocabulary to support API designers in the process of constructing these software artifacts. Our goal is to provide a consistent set of aspects of what *should be thought of* in terms API and language design, from a communication and cognitive perspective. One possible contribution of this research study would be to provide insights for designers to make them more aware of the pragmatic condition of the artifacts in use. It is important to emphasize that the objective is not to provide a *complete* set of factors that influence the

[7]According to de Souza (22), epistemic tools are *"not used to yield directly the answer to the problem, but to increase the problem-solver's understanding of the problem itself and the implications it brings about"*.

design process and its outcomes, but rather a significant one, depending on what could be inferred from the qualitative analysis of selected empirical data.

It is important to distinguish between two main classes of APIs: 1) ‘standard’ APIs, those which are produced by the same group or institution which is responsible for a programming language. These APIs usually provide a set of services that extend the basic language’s functionalities; 2) ‘third-party’ APIs, those which are independent of the underlying programming language’s producers, but uses it as the basis for its implementation and use. They usually provide more specific services not contained in the standard APIs.

Concerning the first category of APIs, there is a ‘continuum’ between the programming language and its standard API, since their design should be mutually consistent with respect to the ‘message’ sent to users, considering a communicative perspective. As explained in the next section, the studies carried out in this thesis were based on empirical data associated with Java and PHP’s standard APIs. Therefore, we take into consideration this relationship between the standard API and its programming language when interpreting the evidence contained in the analyzed empirical data. However, the results obtained in this thesis are not restricted to this category of APIs. The difference lies in the ‘communication sender’ involved, which allows us to include aspects of the underlying programming language as part of the object of analysis when evaluating a standard API from a communication perspective.

The next section describes the theoretical and methodological approach elaborated to pursue these research objectives.

1.4
Approach

This section provides an overview of this work’s methodological approach, in the context of the research objectives explained in section 1.3.

In order to investigate the communicative aspects of the interactions between programmers, one of the first steps of the chosen research approach was the collection of empirical evidence that suggested breakdowns in the reception of the designer’s message, from the perspective of the programmer (API or language user). This type of evidence can be found in different sources like programming forums, discussion lists, Internet blogs, source code repositories, bug tracking systems, and so on.

The main source selected for the collection of empirical data was the issue tracking systems for two popular programming languages, namely PHP and Java. In these systems, the usual work flow involves the creation of a new record in the issue database by a member of the community of users (the reporter). The user describes the problem (or suggestion) in the report, usually following an informal protocol that includes a brief explanation, a small code sample to illustrate the situation (if applicable), the expected result, and the actual result. In the next step, a member of the community of developers (the evaluator) analyzes the report, possibly making comments and classifying the issue according to a predefined list of categories. The taxonomy used in these categories varies among different issue tracker implementations, but most systems include a classification entry specified as 'not an issue', 'not a bug' or similar. This category of issue is of special interest for this research.

When a user registers a new entry in an issue tracking system, it may be that she is facing a blocking situation, and is incapable of comprehending and/or using a specific API or language feature. Other possibilities include the user's suggestion of a new feature, or a change in the language/API behavior or implementation. The latter may indicate that the artifact's design does not meet the user's expectations or needs. In any case, when the evaluator classifies an issue as 'not a bug', this usually means that there is no point in the user's complaint, and that there is nothing to be 'fixed'. In other words, it is as if the evaluator meant to say: 'it's not a bug, it's a feature.'. In a number of cases, this situation may imply a breakdown in the user's interpretation of the language's or API's features. This is a primary object of investigation for the objectives of this research.

As previously mentioned, the selected scope for the collection of empirical evidence included the PHP and Java languages, and their respective standard API's. The basic criteria for choosing these languages was the language's popularity (both appear high in the TIOBE language index[8]) and the availability of a large database of registered issues. Also, the existing differences between these languages influenced their choice, as they could add different perspectives to the analysis. One of the main differences between Java and PHP is their typing system (static for Java, dynamic for PHP). This characteristic of the language may influence the kind of issues that users deal with when interacting with an API. Also, Java is a general purpose object-oriented language, while PHP is a procedural/object-oriented language with a clear focus on the creation of Web applications. Another difference between the languages is their intended audience: PHP aims to be an "easy to learn" language for (possibly) novice programmers, and Java is more oriented to advanced programming tasks.

In order to provide a solid theoretical foundation to the systematic analysis of the empirical data collected, the research method is based on the

[8]`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

discipline of Semiotic Engineering (22). Semiotic Engineering views human-computer interaction as a communication process between designers and users, mediated by an interactive system (the designer's proxy). More specifically, an interactive system can be viewed as a 'one-shot message' from designer to user about how the user should communicate back with the system, which characterizes this process as communication about communication (metacommunication).

Traditionally, Semiotic Engineering has been applied in the context of the design and evaluation of interactive computer applications. Differently from graphical user interfaces, a programmer's interaction with an API occurs when a program is written, compiled, executed and debugged. A programmer reads documentation, reads source code, writes code to perform a certain task, evaluates the return codes, handles exceptions, runs the program, analyses the outcome, reads messages, output traces and logs, and so on. This type of interaction inherently limits the designer options to be 'present' at interaction time when compared to more visual and dynamic system interfaces. Also, this type of interaction has a more 'open' and generative nature, as languages and API's can be used to create arbitrary pieces of software.

Despite the longer tradition in HCI-oriented research, the Semiotic Engineering Research Group (SERG) [9] has been working towards the application of the theory and its research methods in the context of Software Engineering research (23, 24, 25), including this research work. The basic idea behind this trend is that the theory provides a consistent ontological and epistemic support to guide researchers in the investigation of the different forms of human communication mediated by software. The group's long term research agenda includes the semiotic tracing of human communication aspects in the various stages and facets of the software development domain, going up to the end user's perspective.

The systematic analysis of the empirical data collected has been performed under the lens of Semiotic Engineering, which supported the communicative perspective of the qualitative research. In spite of the availability of the discipline's own set of scientific methods (26), they have not been directly applied in the context of this research. This is due to the fact that both methods were not well suited to the type of studies that have been carried out. The Semiotic Inspection Method (SIM), as the name suggests, is an inspection method that focuses on the analysis of the communicability profile of an interactive system (from the designer's perspective), and is based on specific scenarios. The Communicability Evaluation Method (CEM) is used in

[9]`http://www.serg.inf.puc-rio.br/`

the evaluation of interactive systems from the user's (receiver) perspective, by analyzing empirical data from sessions involving participants interacting with a system in a specific scenario. Nevertheless, the theoretical foundations of these methods have been extensively used in the studies developed as part of this research, as will be explained in chapter 3.

In the cognitive context, also a subject of investigation of this research, the Cognitive Dimensions of Notations framework (CNDf) (27) provided support to describe the impact of communicability issues on users. As the object of study involves programming languages and API's, notations play a central role in the qualitative analysis, and the CDNf provides a good vocabulary to describe the cognitive aspects involved. It also serves as an epistemic tool to guide the researcher's reasoning.

The use of the CDNf has a long tradition in research studies concerning API design and evaluation (14). However, the investigation of this subject from a communication perspective is, to the best of our knowledge, a novel approach, especially when considering the theoretical support of Semiotic Engineering. The methodological approach selected for this research intends to provide an in-depth account of communicative and cognitive aspects in use of programming languages and API's. Given the intrinsic qualitative nature of the studies performed in this research, there is no intention to generalize any hypothesis or assumption in order to be predictive about successful design of API's or programming languages.

In order to conclude the overview of the methodological approach, figure 1.2 provides a high-level depiction of the steps of the research work's strategy. Chapter 4 provides a more detailed description of the approach.

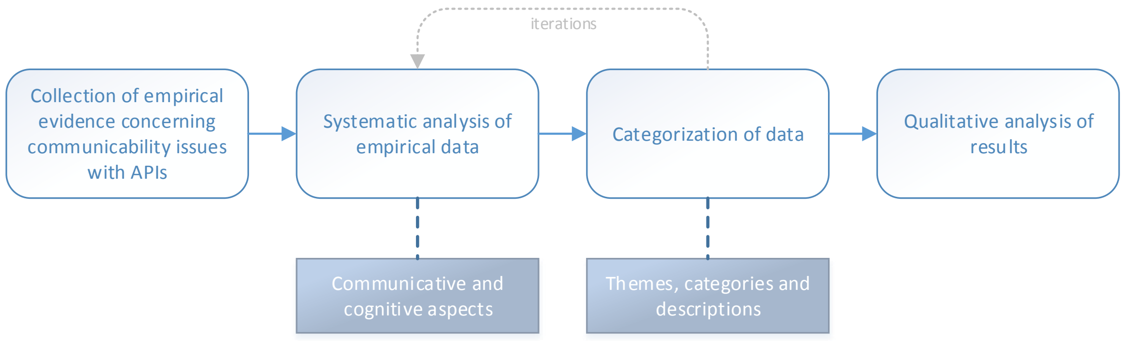

iterations
Collection of empirical evidence concerning communicability issues with APIs
Systematic analysis of empirical data
Categorization of data
Qualitative analysis of results
Communicative and cognitive aspects
Themes, categories and descriptions

Figure 1.2: Research strategy overview

1.5 Results

The research approach described in the previous section provided two kinds of results, to be detailed in chapter 5.

First, the elaboration of a conceptual framework[10] to support API design discussion and evaluation, based on the proposed communicative and cognitive perspective. This framework consists of a set of epistemic tools which contribute to greater awareness of the nature of communicability issues that may occur in the use of APIs.

In addition, this thesis' results include qualitative findings deriving from the elaboration and use of the epistemic framework in the study of empirical data collected from bug reports, as explained in section 1.4.

1.6 Outline

This section provides an overview of the organization of the remaining chapters in this thesis.

Chapter 2 reviews related studies grouped by their main topic, describing their influence on this research work, when applicable.

Chapter 3 describes in detail the theoretical foundations that supported our research approach and methodology, encompassing Semiotic Engineering and the Cognitive Dimensions of Notations framework.

Chapter 4 presents the methodological approach and describes the API studies carried out in the research process.

Chapter 5 analyzes and discusses the research results, organized in two parts: 1) the epistemic tools that compose the conceptual framework elaborated in the research process; and 2) qualitative findings derived from the application of the framework.

Chapter 6 concludes this thesis by providing a final discussion about the research results' implications, main contributions and limitations. It also describes some opportunities for future research.

[10]The meaning of 'framework' in this context is not related to 'software framework', a common term in programming. We refer to 'framework' in its more general sense, as an analytical tool that helps us to reason about a specific domain of problems.

2 Related Work

This chapter contains an overview of research studies which have similar interests or objectives to this work, in various ways. It also provides a more detailed description for those studies which have especially inspired or influenced this research.

As mentioned in chapter 1, to the best of our knowledge, the communication perspective adopted in this research is a novel approach in the literature. For this reason, the studies selected in this chapter are mostly related to API and programming language usability aspects, as well as other API and language research topics in general. The set of studies is not meant to be exhaustive, but a representative sample of previous scientific work in the area.

The following sections propose an organization of the selected studies based on their main contribution or object of analysis. Although these studies could have been categorized in different ways, the main goal is to provide a high-level classification of the chapter contents according to different research approaches.

As such, the selected studies have been organized in the following categories: 1) API and language evaluation; 2) API and language design; 3) API learning and documentation; 4) programming, communication and semiotics; and 5) other studies. This chapter's sections describe related work according to each of these categories, ending with a section with general considerations about the role of some studies in the remaining chapters. As a general rule, studies inside each category follows an approximate chronological order.

2.1 API and Language Evaluation

This section provides an overview of representative work oriented to the evaluation of the quality of languages and APIs, especially from a usability perspective.

Early work on language and API evaluation. The first studies on this subject have been influenced by previous work in the areas of "Empirical Studies of Programming", "Software Psychology" and "Psychology of Programming" (11). Researchers in these areas have been especially active in the 1970's and 1980's. Back then, studies focused on the psychological and cognitive aspects of programmers, and usability was not an usual term.

In 1981, Sheil (28) provided a review of the psychological research on programming at the time, classifying existing work according to the following categories: studies of programming notation (conditionals, control flow, etc.), studies of programming practices (indenting, naming, etc.), and studies of programming tasks (learning, coding, debugging). These categories provide a high-level view of the main focus of researchers at that time. In the review, Sheil criticizes the lack of methodological rigor in some of these studies, arguing that "they appeal to our own 'common sense' model of the cognitive processes involved in programming". This is still a source of concern to some researchers, like Hanenberg (3), who makes similar claims for the use of appropriate research methods in programming language research.

In the 1990's, with the increasing popularity of object-oriented languages and libraries, there was a growing interest in studying the human aspects of programming languages, APIs and reuse. Methods and techniques from the Human-Computer Interaction community had been gradually introduced in studies on programming and human behavior. For instance, Rosson and Carroll reported an empirical study (29) to provide a qualitative characterization of reuse strategies among expert Smalltalk programmers. They observed a systematic use of code from example applications as an implicit specification for reuse, which they called "usage context". The authors referred to this strategy as "reuse of uses": programmers searched for similar code in example applications to guide their own tasks.

The study by McLellan et al. (30), in 1998, is one of the first well-known usability evaluations of an API. The authors collected data from the observation of 20 professional programmers interacting with an actual API related to the oil industry domain. The study included HCI techniques like scenario-based sessions, videotaping, think aloud protocol and interviews. They reported findings about the influence of code examples, which supported some activities among programmers, but hindered others. According to the authors, the code example "*functioned as a way to 'show off' the library's capabilities, allowing the programmers to form hypotheses about the library itself*". However, they also report that the example code sometimes violated "rules of programming discourse" (31), which was a source of confusion among programmers. Their results informed improvements to the API's design, as it was an actual project used in an industrial context. In addition, their report of programmers' behavior and their novel approach influenced other researches in academia to investigate API's from a human perspective.

Cognitive Dimensions of Notations Framework. The first studies on API and language evaluation based on the use of the Cognitive Dimensions of Notations Framework (CDNf) (27) have been performed by Clarke and his colleagues at Microsoft. In the first paper reporting these studies (32), Clarke described how the cognitive dimensions had guided the evaluation of C#, a new programming language at the time. They had a special interest in evaluating the language's new features when comparing to older languages like C++, for example. The study involved laboratory sessions with programmers, asking participants to complete a programming task and then completing a questionnaire. The researcher analyzed their observations and the participants' verbal protocol, together with answers to the questionnaire, under the perspective of the cognitive dimensions. The study provided valuable results that motivated the continuation of their work, especially in the context of library and API usability evaluation (14, 33, 34). Since then, their studies have influenced other researchers' work, including most papers mentioned in this chapter.

Lastly, the work by Maia et al. (35) proposes an adaptation of the CDNf to investigate and compare flexibility aspects of two middleware platforms, characterizing the cognitive effort made by programmers in the adaptation of these systems.

Methods. Following these first studies, other researchers proposed different methods to evaluate APIs. The CDN-based method proposed at Microsoft has influenced the work of other groups, positively or not. For instance, the work by Bore and Bore (36) criticizes the use of the cognitive dimensions to evaluate APIs, arguing that there are too many dimensions, and that "their interpretation is often not obvious, and involves subjective judgment". Instead, they propose a set of "API profile dimensions" that can be "measured through simple mechanical procedures with little subjective judgment". They also mention that the proposed method was tested in the evaluation of an API in the consumer electronics domain.

Other research groups have proposed methods to evaluate various aspects of APIs based on different approaches. As an example, Ratiu and Jurjens propose a formal framework to evaluate how well an API represents the domain concepts involved (37). There is also work proposing methods based on the text analysis of API elements (38), on the use of a "lightweight walkthrough" of the API (39), and on a technique of visual knowledge representation called "Concept Maps" (40).

Another evaluation method that should be mentioned is called "API usability peer reviews" (41). It is an inspection method developed at Microsoft that try to identify usability issues in an API by promoting review sessions including programmers with different areas of expertise, and also uses the cognitive dimensions of notations as a vocabulary to describe the findings and support the discussion.

Recently, Grill et al. proposed a usability evaluation method consisting of three phases based on the combination of HCI techniques: heuristic evaluation, developer workshop and interviews (19).

Case studies. The research group headed by Myers at the Carnegie Mellon University has performed interesting case studies in the evaluation of various API design aspects and their impact on usability. In some of these studies, they adopted a strategy to factor out a single API design aspect in order to investigate its influence on usability, in an attempt to generalize their findings to other APIs. For instance, they have investigated the usability consequences of using the factory pattern to create objects while using an API (42). They also reported on the impact of requiring parameters in an object's constructor, as opposed to the *create-set-call* idiom to instantiate, configure and use an object in Java (15). The influence of method placement inside classes on API learning has also been one of their objects of study (43).

A more recent case study of API usability evaluation has been reported by Piccioni et al. (44), which combines interview questions based on the CDN framework with observations of programmer behavior, using a classification created by the authors and named as "usability tokens". These tokens comprehend a set of five labels that shortly describe an event related to participant's behavior during the programming sessions, as follows: "surprise", "choice", "missed", "incorrect" and "unexpected". This classification has an interesting analogy with our approach to the study of APIs, based on the investigation of communication breakdowns.

It is also worth mentioning the empirical study carried out by Spiza and Hanenberg (45) concerning the influence of using type names on API usability, even in the absence of static type checking. In their findings, the mere use of a type name had a positive influence on the participants' performance, but a single wrong type name had a negative impact on development time.

Metrics. The CDN framework may help in the evaluation of measurable aspects of an API, but it does not have a quantitative nature. Therefore, some researchers proposed methods to provide an objective way to quantify usability aspects of APIs. For instance, studies about metrics based on complexity factors that impact an API's usability have been performed by Doucette (46), de Souza and Bentolila (47), Scheller and Kuhn (48), and Cataldo and de

Souza (49). In a more recent work on metrics, Rama and Kak propose a way of measuring "structural defects" in API method declaration and documentation, considering the recommendations and guidelines given by experts and found in the literature (50).

2.2 API and Language Design

This section describes representative work concerning various aspects of API and language design, including guidelines derived from experienced professionals and researchers, and also studies that investigate factors that influence this type of design.

Design aspects. Researchers have been investigating various design aspects that may impact programmers when interacting with an API. For instance, Stylos has proposed a model of API design decisions (51), identifying the main architectural and language level decisions that designers should consider when creating a set of classes. In spite of being Java-oriented, the model provides an interesting organization of the design space for these decisions that applies to most object-oriented programming languages.

In a different approach, Zibran et al. performed qualitative and quantitative analyses of bug reports, searching for a classification of factors that affected the usability of the APIs studies (52). They identified a set of 22 usability factors like complexity, naming, documentation, consistency, error handling, and so on. Although many factors from their findings were not new, the study provided a quantitative indication of the relative importance of these factors, at least in the bug repositories where data has been collected.

Expert recommendations and guidelines. In 1990, Meyer wrote a paper reporting the experience of designing a programming language standard library, the Eiffel libraries (53). Although the report does not focus on usability issues, it does contain interesting design aspects and lessons learned from this large project.

Other researchers argued for the relevance of human aspects in the design and evaluation of programming languages and APIs, as programmers are frequently left out of this discussion. Arnold suggested the use of HCI techniques and provided interesting examples of how API design can be improved (12). In a similar fashion, Henning provided a reflection about the impact of bad APIs, including compelling examples that support his argument (6).

Finally, Bloch is an often cited researcher and practitioner in the area of API design, for having proposed guidelines and recommendations that strongly influenced scientific and technical work related to this theme (7, 54). Other influential expert guidelines have been collected in books by Cwalina and Abrams (9), from Microsoft, and also by Tulach (8), creator of the NetBeans integrated development environment (IDE).

2.3 API learning and documentation

This section presents selected studies investigating documentation aspects and other factors that may impact programmers' experience when using and learning a new API.

Learning. Robillard reported the results of an empirical study at Microsoft conceived to identify obstacles that professional developers faced when learning a new API (55). The study produced 5 high-level categories of obstacles: resources, structure, background, technical environment and process. The results of this survey served as input to a larger study that also took place at Microsoft (17), which included qualitative interviews and a follow-up survey to confirm the general findings and collect additional data. As previously mentioned in chapter 1, this work provided interesting findings concerning the impact of documentation issues on programmers that try to learn a new API.

A different approach has been adopted by Hou and Li, as they also investigated learning obstacles in using frameworks and APIs, but performed an exploratory study in which they analyzed data from programming newsgroups discussions (56), focusing on the Swing Java Framework.

Other researchers have performed exploratory studies in which they trace and analyze the questions and answers that programmers have to deal with, especially when changing a program or when learning a new API. Examples of this approach can be found in the work by Sillito and Murphy (57) (changing a program), and also by Duala-Ekoko and Robillard (58) (learning a new API).

The need for better support tools for API learning has also motivated the study carried out by Kuhn and DeLine (59). They collected information from sessions with 19 programmers at Microsoft, in order to investigate the requirements and design implications for tools that provide developers with an effective learning experience when using an API.

Documentation. Considering that documentation has a key role in the process of communicating the design intent behind an API, work in this area is also of interest in the context of this thesis. In one type of approach, researchers investigate tool support to enhance documentation usability. For instance, Dekel and Herbsleb created a tool called "eMoose" to address the need of "pushing" information about API directives that have special relevance for the programming task being performed (60). Stylos and Myers proposed and evaluated a tool called "Jadeite" to support users in searching API documentation, displaying frequently used classes and methods in a more noticeable fashion, according to the code being written (61).

The study carried out by Ko and Riche (62) aimed to bridge the gap between API documentation and programmers' conceptual knowledge about the domain to which the programming task refers. In their study, participants were asked to explore the feasibility of implementing two requirements related to a domain they were not familiar with, using a predefined set of APIs. The authors performed an open coding of participants' utterances corresponding to their judgments of the feasibility of the requirements, and arrived at 5 categories: relevance, usability, audience, proximity and metacognitive judgments.

In addition, some empirical studies have been trying to identify and categorize the types of knowledge contained in API documentations, and their distribution. Monperrus et al. created a taxonomy of 23 API directives, which are "natural language statements that make developers aware of constraints and guidelines related to the usage of an API" (63). Following their work, Maalej and Robillard went further and created a taxonomy of 12 API knowledge types, including "directives" as one of the identified types (64). Their taxonomy will be further discussed in chapter 4.

Still in the domain of API documentation, following the work by Spiza and Hanenberg cited in section 2.1, Endrikat et al. performed an experiment and exploratory study to compare the impact of using documentation and a static or dynamic type system on the usability of APIs (65).

2.4 Programming, Communication and Semiotics

This section refers to related studies on programming based on concepts borrowed from communication or semiotic theories.

Programming as communication. Communication-related approaches to programming are not frequently found in the literature. An early work exploring this perspective has been developed by Soloway and Erlich (31), and is frequently cited in studies on programming comprehension. In this paper, they argue that experienced programmers acquire two types of specific knowledge: 1) "programming plans", term describing program fragments that represent patterns of sequences that have common goals and characteristics; and 2) "rules of discourse", which specify common conventions in programming, being analogous to discourse rules in conversation.

Blackwell investigated mental models contained in Java programming libraries, finding some interesting insights about the conceptual metaphors encoded in the Java documentation (66).

In order to investigate how knowledge is shared in a team of programmers, Dubochet developed a study in which he considered programming languages as a medium for human communication, providing insights about how common grounds influence programmers' code comprehension (67).

Orchard discussed programming language design and claimed that "a programming language should improve the four Rs of programs: reading, writing, running, and reasoning". He also stated that the "four Rs" provide a framework for thinking critically about the effectiveness of languages and language features (68).

Semiotics and programming. Zemanek has provided one of the first semiotic perspectives of programming languages (69). In his paper, he discussed how semiotic concepts apply to programming languages, making an analogy with natural languages, with special attention to pragmatics.

An example of a combination of semiotic and cognitive approaches can be found in the work by Kamthan, in which he developed a framework based on concepts from semiotics and on the CDN framework to evaluate formal specifications using the Z language (70).

Lastly, the book by Tanaka-Ishii (71) provides a deep and solid semiotic account of programming paradigms and languages, from a theoretical perspective. Even though this thesis' approach has been guided by the theory of Semiotic Engineering (as opposed to Semiotics in general), Tanaka-Ishii's book has also been a source of inspiration for this work.

2.5 Other studies

This section presents other studies worth mentioning that could not fit into the categories of previous sections.

Computer-supported Collaborative Work. De Souza et al. provided studies that investigate the roles of APIs in the context of collaborative work, and observed that they can be metaphorically regarded as contracts, boundaries or communication mechanisms (4, 72).

Meta studies. Burns et al. conducted a systematic mapping study of the literature on API usability, reviewing a total of 28 papers and categorizing them by research type. They also provided a closer analysis of the papers' evaluations to summarize recommendations (73). Most of the studies analyzed have also been cited in this text.

Another type of meta-study performed by Stefik et al. provides a systematic evaluation of the use of empirical evidence with human users contained in the papers from workshops PPIG, Plateau and ESP. They analyzed the collected data both qualitatively and quantitatively to assess the overall quality of the evidence on this type of study (74).

Security. From a software security standpoint, Wurster and Oorschot claimed that not all developers can be security experts. As such, they argued that programmers that design and develop APIs intended to a large community of users should put more effort into its usability, with a stronger focus on security (75).

2.6 Considerations about related work

This section wraps up the chapter and highlights some aspects of related work described in previous sections that influenced this research, and how they appear in the remaining of this text.

Similarly to studies from Clarke and others, e.g. (32), our research approach uses the CDNf to study APIs. However, differently from these studies, we frame the use of the cognitive dimensions in a communicative context, as will be explained in the remaining chapters.

Maalej and Robillard's taxonomy for 'patterns of knowledge in API documentation' (64) has been used in the studies carried out in this research in order to provide conceptual support to the analysis of bug reports, as will be explained in chapter 4.

Piccioni's usability tokens(44) describe users' reactions during their interaction with APIs. We provide a classification for the effects of communicability on users which shows some similarities to their work. However, our classification is based on a communicative perspective of the phenomenon, and shows a different organization of the categories, as will be detailed in section 5.2.

Next chapter introduces the theoretical foundations that supported the development of this research.

3
Theoretical Foundations

This chapter introduces the theories and methods that inspired and oriented our research work. The text provides an overview of Semiotic Engineering, describing its foundations and main concepts, and explaining why the theory is suited for supporting the research. In addition, an interpretation of some the theory's concepts in the context of API design and evaluation is discussed. In addition, this chapter offers a brief description the Cognitive Dimensions of Notations framework, also discussing its role in the research work.

3.1
Semiotic Engineering

Semiotic Engineering has been developed as a theory for Human-Computer Interaction (HCI), and provides a semiotically-based interpretation of software interface design and interaction as a communication process between designers and users[1]. In this context, a software artifact plays the role of the 'designer's deputy' at interaction time, reproducing the designer's messages and interpreting user's responses. The theory has been supporting a number of studies which provided new perspectives and insights to the HCI community, and more recently has been applied in the context of Software Engineering research. This thesis is an example of the latter case, as it brings a Semiotic Engineering perspective to the study of programming-related topics.

Semiotic Engineering has its roots on Semiotics, the study of signs, signification systems and the process of meaning construction, with strong philosophical and linguistic connections. This section briefly introduces the main concepts required for a general comprehension of the theory and its application in the context of this research.

[1]In Semiotic Engineering theory, the singular terms 'designer' and 'user' are frequently used to refer to two 'groups' of people: one that designs and develops a technology and, on the other side, another group which uses the technology for a variety of purposes in a number of situations. Adopting the term 'designer' instead of 'developer', for example, highlights the importance of communicating the *design logic*, which should be followed in the technology's development.

3.1.1 Signs and semiosis

Charles Sanders Peirce, one of the founders of Semiotics, defined a sign as *"anything that stands for something else, to somebody, in some respect or capacity"* (76). Peirce's theory was strongly influenced by his background as a logician, philosopher and scientist. In his studies, he developed an interest in epistemology, investigating the nature of mind and the process of acquiring knowledge.

Ferdinand de Saussure, another important thinker and contemporary of Peirce (c. 1900), also investigated signs and signification. He was the founder of *Semiology* (77), a theory of signs based on the study of language and linguistics. He studied the organization of signs as arbitrary units used by individuals to communicate in a number of contexts, and proposed the distinction between *langue*, the abstract system of these units of language, and *parole*, the actual use of signs in communication. He proposed a dyadic model of signs, consisting of two parts: the *signifier* and the *signified.*

Peirce proposed a model of sign structure based on three constituents: *object* (referent), *representamen* (representation) and *interpretant* (meaning). One of the consequences of Peirce's triadic model is the need of an interpreter: meaning is associated with a representation only by the mediation of an interpreter's 'mind'. This implies that a representation of an object may trigger different meaning-making processes, depending on the mind involved in signification. This process of sense making is known as *semiosis*, and it may be repeated indefinitely, since the association of meaning to a sign can potentially trigger other signs, in a recursive process that is theoretically unlimited ('unlimited semiosis').

By itself, the perspective of semiosis as an unlimited process is problematic, since it does not explain how stability is reached in the construction of meaning. According to Peirce, this process is halted due to *abductive reasoning.*

3.1.2 Abductive reasoning

Peirce's theory proposed that, in the meaning-making process, an interpreter's mind formulates plausible rules that contribute to explain an observed result and determine the 'meaning' constituent of a sign. If a confirming case is observed, the rule is (at least temporarily) reinforced. Conversely, if a contradicting observation occurs, the rule is revised to provide a new hypothesis to explain it. This process, known as 'abduction' or 'abductive reasoning', may proceed iteratively until reaching a stable rule. Figure 3.1 illustrates this cycle.

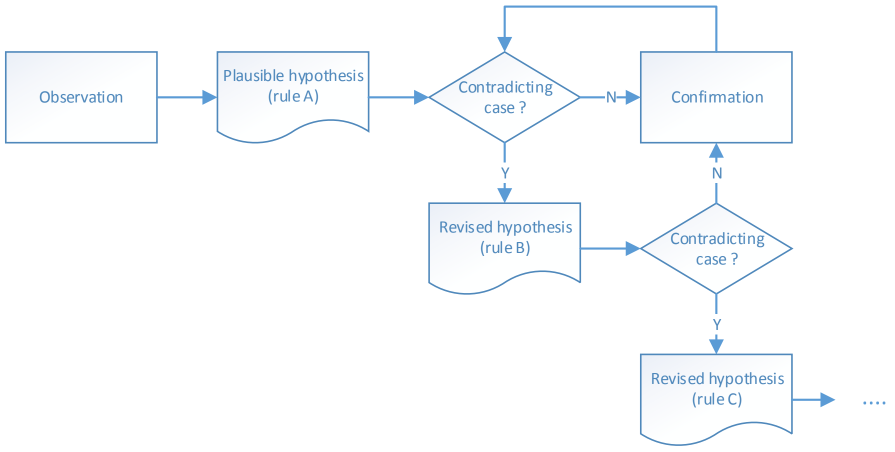

Observation
Plausible hypothesis (rule A)
Contradicting case ?
N
Confirmation
Y
Revised hypothesis (rule B)
N
Contradicting case ?
Y
Revised hypothesis (rule C)
....

Figure 3.1: Abductive reasoning

This type of reasoning contrasts with *deductive reasoning*, which consists of applying a known rule to an observed case, leading to the inference of new results. Abductive reasoning explains how the interpretive process (semiosis) can be halted by finding a plausible hypothesis which explains a pragmatically defined set of similar cases, which are already confirmed and tested. This process may also be interrupted due to the finite and fallible nature of the human mind.

3.1.3 Communication processes

Semiotic Engineering relies on Roman Jakobson's model (78) for the characterization of the communicative design space. This model consists of six elements: sender, receiver, channel, message, code and context. Figure 3.2 illustrates the model and its components.

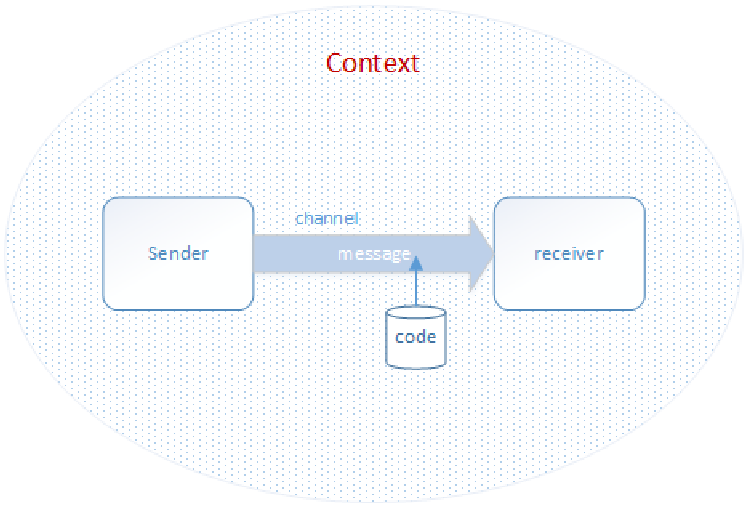


Figure 3.2: Jakobson's model of the communication space

In summary, the model states that a communication process consists of an individual (sender) creating content (message) encoded in a representation (code) and sending it to its peer (receiver) by certain means (channel), in accordance to shared assumptions and knowledge (context).

In addition to Jakobson's model, communication can also be characterized in terms of three dimensions: *intent*, *content* and *expression*. Intent corresponds to sender's goals in communication; expression refers to the use of different code forms to represent the message; and content describes the meaning carried by the sender's message. These concepts will be further discussed in chapter 5.

According to the definition in (22) p. 26, communication is "the process through which, for a variety of purposes, sign producers (i.e., signification system users in this specific role) choose to express intended meanings by exploring the possibilities of existing signification systems or, occasionally, by resorting to non-systematized signs, which they invent or use in unpredicted ways." In this context, a signification system consists of a set of culturally established representations and their associated rules, providing codes for social communication processes.

Semiotic Engineering concerns the study of two classes of communication processes: 1) user-system communication, comprehending user's interaction with the *designer's deputy* (i.e. the interactive software system); and 2) designer-to-user communication, in which designers send a message to users to tell them how to communicate back with the system and, for this reason, it can be regarded as a *metacommunication* process. The concept of metacommunication will be discussed later in subsection 3.1.6.

3.1.4 Software as an intellectual artifact

The term 'artifact' describes physical or mental objects or tools created by humans that results from our ingenuity. An 'intellectual artifact' is a special type of artifact that possesses the following characteristics, as described by de Souza in (22), p.10:

- it encodes a particular understanding or interpretation of a problem situation;
- it also encodes a particular set of solutions for the perceived problem situation;
- the encoding of both the problem situation and the corresponding solutions is fundamentally linguistic (i.e., based on a system of symbols

– verbal, visual, aural, or other – that can be interpreted by consistent semantic rules); and

– the artifact's ultimate purpose can only be completely achieved by its users if they can formulate it within the linguistic system in which the artifact is encoded (i.e., users must be able to understand and use a particular linguistic encoding system in order to explore and effect the solutions enabled through the artifact).

This definition is suitable to describe an interactive piece of software, which also consists of a linguistic encoding of a particular solution to a problem. However, what distinguishes software from other intellectual artifacts is the fact that the solution is encoded in an artificial language. Similarly to natural languages, artificial languages also have lexical, syntactic and semantic rules. However, differently from natural languages, artificial languages do not necessarily have a set of pragmatic rules. These rules provide a refinement of natural languages' semantic rules with respect to the context of communication.

In semiotics, pragmatics accounts for the relation between two parts of a sign: representation and meaning. In a linguistic approach, pragmatics studies how context influences receivers in the assignment of meaning to messages encoding senders' intent. Regarding the intellectual nature of a software artifact and the typical lack of pragmatic rules in artificial languages, one of the goals of Semiotic Engineering is to provide a deep account of the pragmatic aspects of human communication mediated by software in an attempt to provide support to overcome the inherent limitations if this type of communication.

3.1.5 Pragmatic concepts and principles

Speech Act theory (79, 80) defines pragmatic concepts which influenced HCI theories in general, and Semiotic Engineering in particular. It argues for the use of language as a means to do things or to achieve some effect in the world. This perspective promotes a refinement in the use of language, by distinguishing between the language user's intent (the *illocutionary act*) and the effects that result from language use (the *perlocutionary act*).

There are five categories to characterize the nature of speech acts, as described by in (22), p. 60:

– assertive: speech acts that commit the speaker to the truth of what is being said;

- directive: speech acts that aim at causing the hearer to do something;
- declarative: speech acts that change the status of the world by virtue of what is said, by whom, and to whom;
- commissive: speech acts that commit the speaker to taking some particular course of action in the future;
- expressive: speech acts that aim at drawing the hearer's attention to the speaker's psychological state or attitude.

Influenced by speech act theory, philosopher Paul Grice proposed the Cooperative Principle (81), a pragmatic principle that helps to describe the constraints involved in the elaboration of a conversation. The Cooperative Principle defines four 'maxims of conversation', as described in (22), p. 61:

- The maxim of quantity: participants in a conversation should make their contribution as informative as necessary; not more, not less;
- The maxim of quality: participants in a conversation should only say what they honestly believe to be the case; the should acknowledge their doubts about what they don't know for a fact, and never tell a lie;
- The maxim of relation: participants in a conversation should only talk about what is relevant for the ongoing conversation;
- The maxim of manner: participants in a conversation should express their contribution clearly and unambiguously.

Pragmatists' efforts to formulate concepts and principles provided relevant contributions to refine the study of communication. However, cultural factors and individual characteristics of speakers and listeners strongly influence the actual outcomes of communication. For this reason, pragmatics is usually described in terms of principles, not rules, since human reasoning and behavior cannot be always predicted. Therefore, Semiotic Engineering is strongly influenced by pragmatics, due to the intellectual nature of a software artifact and its linguistic encoding of an intended solution to a problem. These concepts and principles will be further referenced in chapter 5.

3.1.6
Metacommunication

Metacommunication can be technically defined as "communication about (aspects of) communication". It is a useful resource to provide 'hints' to receivers about how communication itself should be interpreted.

In HCI terms, this is a commonly explored technique to tell users a message about how they could achieve certain goals by interacting with it. For instance, the presence of a hyperlink or a button in an eletronic document may suggest that something will happen if the user clicks on it.

A particular characteristic of metacommunication in the HCI context is that the designer encodes all contents in a single and immutable message, which makes it a *one-shot message*. This message can be paraphrased by a concept from Semiotic Engineering known as the *metacommunication template*:

> *"Here is my understanding of who you are, what I've learned you want or need to do, in which preferred ways, and why. This is the system that I have therefore designed for you, and this is the way you can or should use it in order to fulfill a range of purposes that fall within this vision".*

The template takes the designer's perspective to communicate the design vision encoded in a software artifact. Metacommunication is only complete if the user interprets the message's signs in a way that generates meanings which are compatible with the designer's intent. Otherwise, user-system communication is unsuccessful, which implies that metacommunication is not achieved either.

3.1.7
Semiotic Engineering ontology

The categories of 'things' that are available for studies in Semiotic Engineering are described by the theory's ontology, which comprehends:

– signification processes: signs and semiosis
– communication processes: direct user-system communication and designer-to-user metacommunication
– interlocutors: designers, systems and users
– design space: senders, receivers, contexts, codes and messages

This ontology, despite its reduced size, encompasses all the elements that compose a communicative view of HCI, including the concepts, actors, processes, and a model that organizes their interaction in the design space.

3.1.8 Communicability

'Communicability' is a key concept in Semiotic Engineering, and it has been evolving over the years along with the theory's refinement, as discussed in (26). One of the first definitions of communicability described it as "*the distinctive quality of interactive computer-based systems that communicate efficiently and effectively to users their underlying design intent and interactive principles*" (82).

In 2005, the concept has been revised to explicitly include the interlocutors involved (designer's deputy and user): "*Communicability can (...) be more technically defined as the designer's deputy capacity to achieve full metacommunication, conveying to users the gist of the original designer's message. (...) Communicability applies to both interpretive and expressive codes that the designer's deputy handles for generating and interpreting messages during situated interaction with users.*" (22)

The experience acquired from a number of studies with Semiotic Engineering methods led to another refinement in the definitions above, especially crafted to avoid misunderstandings concerning the use of terms like 'efficient' and 'effective', which are frequently used in quantitative studies but with different semantics. Therefore, the concept of 'efficient and effective communication' has been defined as "*communication that is organized and resourceful (efficient), and achieves the desired result (effective)*" (83).

3.1.9 Classification of signs

Semiotic Engineering provides a classification of signs that reflects the nature of their representation in designer-to-user metacommunication. The classification is mostly associated with the inspection of software artifacts using the Semiotic Inspection Method (see below), which provides a communicability profile for artifact's design. The classification consists of the following types of signs:

– static: signs whose representation is motionless and persistent when no interaction is taking place;

– dynamic: signs whose representation is in motion regardless of users' actions or whose representation unfolds and transforms itself in response to an interactive turn;

– metalinguistic: signs that represent other static, dynamic, or metalinguistic signs.

Static signs stimulate users to interact with the system, and provide hints about this interaction (e.g. a button with an ‘open’ label). As interaction proceeds, dynamic signs confirm or not users’ expectations, as anticipated by static signs (e.g. the opening of a ‘file open’ dialog after clicking on the ‘open’ button). The meaning of static and dynamic signs is explained and illustrated by metalinguistic signs (e.g. documentation about the behavior of the ‘open’ button).

3.1.10 Scientific methods

Semiotic Engineering provides two methods for scientific investigation: the Semiotic Inspection Method (SIM) and the Communicability Evaluation Method (CEM) (26). These methods have been designed to support a new account of known problems, by using theoretical concepts that help in the formulation of new research questions. They may also contribute to the identification of new solutions to known problems, or even the identification of new problems, theories, concepts or methods.

Semiotic Inspection Method

The goal of SIM is to evaluate the communicability of a software artifact, focusing on the emission of the message. It has been originally developed to explore how interface languages communicate, implicitly or explicitly, the logic and design intent from those who conceived the system. The metacommunication template guides the organization and execution of the method’s steps.

The method consists of a sequence of five steps, where the first three *deconstruct* the designer’s message by performing a segmented analysis of the different classes of signs (static, dynamic, metalinguistic). The last two steps *reconstruct* the message by integrating and interpreting the deconstructed signs: the fourth step compares the results of the segmented analyses performed in the first three steps, and the last step does a final evaluation of the artifact’s communicability. The result is a characterization of the message structure in terms of signs and meanings.

After inspecting a software artifacts’ signs, an analyst should be able to fill out the metacommunication template’s parts, according to her findings. This allows to contrast aspects of the designer’s intended and actual meta-communication, in order to find inconsistencies, incompleteness, ambiguities or other issues that may affect the artifact’s communicability.

Communicability Evaluation Method

Contrasting with SIM, CEM is an evaluation method that focuses on users' reception of metacommunication. The method involves the collection of empirical data associated with users' interaction with a software system, in a specific scenario of use. The main objective in the analysis of collected data is to identify user breakdowns that reveal communicative failures. The evaluation also allows the inference of other aspects of communicability, even positive ones. In order to allow a pragmatic interpretation of the communication between the user and the designer's deputy, this type of evaluation should only refer to situated interaction determined by goal-oriented scenarios.

The classification of failures is based on the use of communicability tags, which provide a diagnostic of the nature of users' difficulties faced during interaction. There are three major types of failures: complete, partial and temporary, described in detail further in this section. Each type corresponds to inconsistencies between intent and effects of communication (illocutionary and perlocutionary acts). They also distinguish between the impact on users' goals (global illocution) or strategies (local illocution).

The diagnostic of failures in CEM occurs by *tagging* users' utterances identified in empirical data. Tagging applies to goal-oriented tasks, and the evaluator needs to do an assessment of the designer's metacommunication message conveyed by the application before the evaluation tests. Then, she should select relevant tasks in the application that should be part of the evaluation tests.

Tagging comprehends thirteen basic communicability utterances to characterize breakdowns in user-system communication (communication between the user and the designer's deputy). Communicability utterances have been used mostly in evaluations of interactive interfaces, but they can be applied to most computer-based techonologies, and its use with respect to APIs will be further discussed in chapter 5.

Complete Failures. Complete failures occur when global illocution is not consistent with global perlocution, which corresponds to situations in which users' goals are not achieved. This type of failure can be further specialized to distinguish between situations where user is aware of failure or not. Table 3.1 presents the tags that represent this type of failure, its description and illustrative symptoms. The contents of tables in this section describing communicative tags have been adapted from CEM method explanation in (26).

The 'looks fine to me' is the most severe breakdown, since it represents a complete failure in which the user thinks that she has achieved her goal, without actually achieving it. For this reason, only the evaluator is able to

Complete failures		
Distinctive feature	**Tag**	**Illustrative symptoms**
User is conscious of failure	"I give up."	The user believes that she cannot achieve her goal and interrupts interaction.
User is unconscious of failure	"Looks fine to me."	The user believes she has achieved her goal, although she has not.

Table 3.1: Complete failures

identify this tag's occurrence, as the user is clearly not in a situation to be able to realize it (or else she would not be mistaken in the first place).

Partial failures. These failures correspond to situations in which local illocution is consistent with local perlocution, but somehow the user fails to do exactly what is expected. Failure subtypes indicate if user understands the designer's deputy illocution or not. This means that users achieve their goals by interacting with the system in unexpected or inefficient ways. Table 3.2 illustrates the tags that correspond to a diagnostic of this type of failure.

Partial failures		
Distinctive feature	**Tag**	**Illustrative symptoms**
User understands the design solution	"Thanks, but no, thanks."	The user deliberately chooses to communicate her intent with unexpected signs, although she has understood what preferential designer's solutions are promoted.
User does not understand the design solution	"I can do otherwise."	The user communicates her intent with unexpected signs because she cannot see or understand what the system is telling her about better solutions to achieve her goal.

Table 3.2: Partial failures

Temporary Failures. Lastly, temporary failures happen when global illocution is consistent with global perlocution, but local illocution is not consistent with local perlocution. This situation indicates problems in the user's strategy to achieve her goal, which needs to be revised. There are three subtypes of temporary failures. First, user's semiosis may be temporarily halted, because she is trying to find a way to express her intended action, could not understand system communication or does not know how to proceed. Table 3.3 illustrate the first group of temporary failures.

The second group of temporary failures describes breakdowns that occur when the user realizes that her illocution is not appropriate for the intended action. This may be caused by the user expressing his intent in the wrong context, using the wrong expression or following a wrong path in conversation. Table 3.4 summarizes these failures.

Temporary failures: (1) user's sense making is temporarily halted		
Distinctive feature	**Tag**	**Illustrative symptoms**
Because she cannot find he appropriate expression for her intended action.	"Where is it?"	The user knows what she is trying to do but cannot find an interface element that will tell the system to do it. She browses menus, opens and closes dialog boxes, etc. looking for the particular sign.
Because she does not see or understand the designer's deputy's communication.	"What happened?"	The user does not understand the system response to what she told it to do. Often, she repeats the operation whose effect is absent or not perceived.
Because she cannot find an appropriate strategy for interaction.	"What now?"	The user does not know what to do next. She wanders around the interface looking for clues to restore productive communication with the sytem. She inspects menus, dialog boxes, etc. without knowing exactly what she wants to find or do. The evaluator should confirm if the user knew what she was searching ("Where is it?"), or not ("What now?").

Table 3.3: Temporary failures: (1) user's semiosis may be momentarily halted

Lastly, the third group of temporary failures corresponds to the user's attempt to understand the designer's deputy's illocution, by exploring metacommunication explicitly or implicitly, or by autonomous sense-making. Table 3.5 lists the third group of temporary failures.

Temporary failures, as suggested by their name, may be in effect only for a short period of time, being circumvented by user's continuous interpretations of the metacommunication message. However, successful occurrences of this type of failure may hinder user's sense-making in ways that compromise interaction, leading to complete failures.

The classification of communicative failure will be further referenced in chapter 5, when discussing the results of the empirical studies.

3.1.11 Semiotic Engineering in the context of APIs

The previous topics in this chapter provided an overview of Semiotic Engineering's foundations and its main concepts . In this subsection, we discuss why the theory is suited to investigate APIs, how it applies to this kind of study and the nature of potential findings. The goal is to provide an understanding of the motivation and benefits of the methodology developed during the research work described in chapter 4.

As previously discussed, software is an intellectual artifact because it provides a linguistic encoding of a particular solution to a perceived problem.

Temporary failures: (2) user realizes her intended interaction is wrong		
Distinctive feature	**Tag**	**Illustrative symptoms**
Because it is uttered in the wrong context.	“Where am I?”	The user is telling things to the system that would be appropriate in another context of communication. She may try to select objects that are not active or to interact with signs that are output only.
Because her expression is wrong.	“Oops?”	The user makes an instant mistake but immediately corrects it. The “Undo” operation is a typical example of this tag.
Because a many-step conversation has not caused the desired effects.	“I can’t do it this way?”	The user is involved in a long sequence of operations, but suddenly realizes that this is not the right one. Thus, she abandons that sequence and tries another one. This tag involves a long sequence of actions while ”Oops!” characterizes a single action.

Table 3.4: Temporary failures: (2) user realizes that she must reformulate illocution

It also represents the designer’s understanding of both the problem and its solution. A user, in order to benefit from this solution, has to engage in a ‘linguistic contract’ with the designer. However, as already mentioned, this communication involves the use of artificial languages, pragmatically more limited than natural languages. In addition, one of the interlocutors – the designer’s deputy – has limited semiotic capabilities, since it is equivalent to a *snapshot* of the designer’s semiosis, encoded in a *one-shot* metacommunication message. Therefore, these communication processes (designer-to-user metacommunication and user-system communication) are complex phenomena posing important pragmatic challenges to their investigation, which are objects of study for Semiotic Engineering.

Traditionally, Semiotic Engineering has focused on the study of interactive interface languages in HCI contexts. Recently, the theory has been used in a larger context to encompass other types of language (e.g. programming) which, just like interface languages, carry the logic and design intent of those who created the language, API or program. Programming interfaces, when compared to visual interactive interfaces, are composed by elements of lower abstraction levels, but with the potential to be combined in multiple and complex ways.

From a semiotic perspective, visual interface languages consist of a rich signification system, providing a great variety of signs that represent potentially complex abstractions to be combined in the form of an interactive interface. In contrast, programming languages and APIs usually consist of a smaller set of signs, but offering great flexibility and allowing for complex combinations to provide an unlimited number of new abstractions and signs

Temporary failures: (3) user seeks to clarify the designer's deputy's intended signification		
Distinctive feature	**Tag**	**Illustrative symptoms**
Through implicit metacommunication.	"What's this?"	The user does not understand an interface sign and looks for clarification by reading a tool tip or by examining the behavior of a sign.
Through explicit metacommunication.	"Help!"	The user explicitly asks for help by accessing "online help," searching system documentations, or even by calling the evaluator as a "personal helper."
Through autonomous sense making.	"Why doesn't it?"	The user insists on repeating an operation that does not produce the expected effects. She perceives that the effects are not produced, but he strongly believes that what she is doing should be the right thing to do. In fact, she does not understand why the interaction is not right.

Table 3.5: Temporary failures: (3) user studies the designer's deputy's illocution

(including the ones that compose interactive interfaces).

According to Tanaka-Ishii (71), there are four kinds of signs in computer programs: literals, operators, reserved words and identifiers. The first three types belong to the language system definition, and programmers only utilize them. Conversely, identifiers are mostly defined by the programmer, who articulates them in a variety of ways to represent in code the various entities that compose her solution to a specific problem. To the 'mechanic' interlocutor (the computer), the meanings encoded in these identifiers' are not relevant, since they only check the syntactic and semantic of identifiers with respect to the program structure, not considering the vocabulary from natural languages. In contrast, these identifiers are of great value to the human interlocutors' (the programmer included), since they are a powerful resource to signify the intent encoded in the program.

In her semiotic approach to programming, Tanaka-Ishii also proposes three levels of semantics for the interpretation of identifiers:

– Computer hardware level: an identifier represents a memory address and a pattern of bits of associated information stored in this address;

– Programming language level: identifiers at this interpretation level may represent the definition and use of entities in a program. At this level, there are two additional layers of interpretation: a) layer of type, defining the kind of data it may be associated with; and b) layer of address, in a sense that is analogous to the hardware level;

– Natural language level: identifiers usually represent lexical elements from natural languages, and as such, use the same vocabulary. This allows programmers to encode meanings in the program to be interpreted by other programmers, herself included. According to the author, "since identifiers are borrowed from natural language, they are considered subject to normal semiotic analysis of terms in natural language."

Identifiers play a key role in the API programming context, since they are a powerful resource for API designers to signify their intent behind the interface of a software artifact. Assuming that in most cases API users don't read an API's source code in order to use its affordances (because they can't, don't want or need to it), the interface specification represents the 'window' that enables users to communicate with the API. The syntactic specification of API operations using the underlying programming language is the only artifact that is surely available to users, and the identifiers involved (names, types, parameters, etc.) are key resources to communicate design intent to users.

In addition to the kinds of signs in programming languages mentioned earlier, the classification of signs from Semiotic Engineering may also be interpreted in the context of APIs. For instance, static signs comprehend mostly the same kinds of signs listed by Tanaka-Ishii: literals, operators, reserved words and identifiers. Their representation is persistent even when no 'interaction' occurs. They encompass signs from the API and the programming language.

Dynamic signs can be interpreted as the ones that result from 'user interaction', i.e. execution of the API's operations. They consist of return values, error messages, exceptions, side effects, and other observed behavior that derives from using the API (the absence of signs may also be considered a dynamic sign itself).

Metalinguistic signs, just like in interactive interface languages, represent other static, dynamic or metalinguistic signs. The most evident example in the case of APIs and programming languages is documentation. However, it may also include other instances like comments, metalevel language elements (e.g. Java annotations), contract specifications, computational reflection resources, and so on.

The design of interfaces (programming included), as seen, involves the articulation of a number of signs to encode the intent that motivates its construction (in the problem and solution spaces). One of the envisioned advantages of a semiotic view of interface languages is the fact that the signs composing an artifact are available to be interpreted, independently of the sign producer's intent. Therefore, if a researcher knows the actual intent behind these signs, and is able to collect evidence of the consumer's interpretation of the signs, then she may identify existing mismatches in this communication process.

When compared with visual interfaces, programming interfaces show differences in the nature of users' interaction. API 'interaction' consists of writing code in the corresponding programming language to call an API's services, passing required parameters and satisfying its preconditions. However, there is a 'gap' in the time between this preliminary interaction with its actual effects: results only occur after executing the program, which may also involve a compilation step. Abstracting from the details of supporting tools like interpreters, compilers and runtime environments, we may view API interaction as the iterative cycle of writing code that calls API's operations and executing the program to obtain results. Dynamic signs occur in the second part of the interactive cycle, as a resulting effect of calling the operations, which may include exceptions, returned values, side effects, and so on.

Designer-to-user metacommunication, in the API context, occurs through the articulation of the static, dynamic and metalinguistic signs, as described. Therefore, in order to investigate the pragmatic aspects that influence this type of communication process, it is imperative to adopt a 'holistic' approach to APIs, encompassing the different signification systems and kinds of signs used in this conversation. This means that the design of APIs should convey coordinated and consistent messages in the different codes and channels involved, in order to achieve effective metacommunication. Conversely, the evaluation of API communicability should also be based on a comprehensive analysis of its various components: formal language specification, natural language documentations, models, runtime behavior, and other potential sources of signs.

Semiotic Engineering offers to designers the necessary support to study and analyze users and their reactions with respect to the proposed artifact. In this context, it also provides the opportunity for designers to study, analyze and make decisions about their own communicative behavior and strategies, which makes it an essentially reflective theory. As a consequence, the theory places designers in its ontology in a position as important as the users'.

The reflective nature of Semiotic Engineering helps designers in the development of a deeper understanding of the problem and the solution to be encoded in the design of an artifact. As a consequence, the theory may be used to produce 'epistemic tools', which are "not used to yield directly the answer to the problem, but to increase the problem-solver's understanding of the problem itself and the implications it brings about" (22). This type of tool addresses the problem's space and nature, and also the restrictions to potential solutions. This is an envisioned benefit of using Semiotic Engineering as the theoretical foundation for investigating APIs, since supporting designers to develop greater awareness and comprehension about the complexities of this phenomena may contribute to more effective API design and use.

In contrast, the inherently non-predictive nature of Semiotic Engineering theory constrains the possibility of a positivist approach to investigating APIs, which would pursue cause-effect relations in the context of API design and use. Therefore, adopting the theory as the ‘lens’ to investigate API design and evaluation does not allow us to expect findings with predictive nature.

3.2 Cognitive Dimensions of Notations framework

The Cognitive Dimensions of Notations framework (CDNf) (27) was conceived to help in the assessment of cognitive aspects associated with representations like, for instance, programming languages and APIs. A brief definition for the framework is “*a broad-brush evaluation technique for interactive devices and for non-interactive notations*” (84). It also provides a vocabulary to describe and discuss cognitive impact associated with the use of an artifact, offering a comprehensible framework that, according to the authors, can be used by non-specialists. In addition, one of its goals is to organize the design space from a cognitive perspective, and expose the potential consequences of certain design decisions in terms of trade-offs among dimensions.

The cognitive dimensions (CDs) usually apply to the analysis of scenario-based tasks, associated with various types of activities categorized as follows: incrementation, transcription, modification, exploratory design, searching and exploratory understanding. Among these categories, incrementation, exploratory design and understanding frequently occur in a programming context.

The CDNf has been successfully applied to evaluate the design of artifacts in many different contexts. For instance, the CDs motivated studies to evaluate the usability of programming languages (32), visual languages (84), middleware (35), and APIs (14, 41, 44).

Despite its suitability to evaluate a variety of notations and devices, we refer to the CDNf in this section with respect to its application in the context of APIs and programming languages.

3.2.1 List of dimensions

The core set of cognitive dimensions includes 14 of them. Some of these dimensions are self-explanatory, and others deserve further discussion. Table

3.6 summarizes each of the core dimensions (84, 85).

Cognitive dimension	**Description**
Abstraction	Types and availability of abstraction mechanisms
Closeness of mapping	Closeness of representation to the domain
Consistency	Similar semantics are expressed in similar syntactic forms
Diffuseness	Verbosity of language
Error-proneness	The notation invites mistakes and the system gives little protection
Hard mental operations	High demand on cognitive resources
Hidden dependencies	Important links between entities are not visible
Premature commitment	Constraints on the order of doing things
Progressive evaluation	Work-to-date can be checked at any time
Provisionality	Degree of commitment to actions or marks
Role expressiveness	The purpose of an entity is readily inferred
Secondary notation	Extra information in means other than formal syntax
Viscosity	Resistance to change
Visibility	Ability to view entities easily

Table 3.6: Cognitive dimensions of notations

The *abstraction* dimension refers to the minimum and maximum levels of abstraction that are available or required from a notation. Regarding programming languages, for instance, they can be characterized in terms of the abstract dimension as: 1) 'abstraction-hungry', if the user is required to provide a number of abstractions in order to be able to use it at a starting level; 2) 'abstraction-tolerant', if it allows the creation of new abstractions by users; and 3) 'abstraction-hating', if it does not allow to create new abstractions. From this classification, we can see that this dimension may contribute positively or not to a language's usability. In addition, the creation of new abstractions may affect negatively other dimensions like hidden dependencies or premature commitment, as discussed later when describing these dimensions.

The *closeness of mapping* dimension describes the distance between a 'problem' and its 'solution' as provided by a notation. For instance, a language that offers sophisticated mathematical primitives provides more closeness of mapping to solve mathematical problems than languages that do not have such features. This dimension is also influenced by abstraction, since higher level abstractions may provide a more direct mapping to a domain, increasing the language's closeness of mapping to accomplish tasks related to a specific domain.

Consistency is a somewhat intuitive dimension: when using a consistent language, the user should be able to infer some of its features by knowing other parts of the language. This avoids 'surprises' when dealing with a language, possibly reducing the language's error-proneness.

Diffuseness describes situations in which the user needs to write more than necessary or expected to achieve a certain goal while using a language or notation. For instance, Cobol may be regarded as an example of a diffuse programming language, since it requires lots of text writing in order to accomplish basic tasks.

Error-proneness is associated with a high probability of errors when using a language, because it does not offer mechanisms to protect the user from mistakes or lapses. An example of error-proneness in a programming language is the possibility of inadvertent use of semicolons after *for* or *while* statements in C, which is a common source of errors among novice programmers.

The *hard mental operations* dimension describes language features which impose a high cognitive load on users in order to use them effectively. To illustrate this dimension, we may refer to languages that allow the programmer to iterate over an array, for example, by using statements with a 'foreach' semantics, returning each element of the collection at a time, without having to control each aspect of the iteration. This involves less cognitive resources to express when compared to traditional 'for' and 'loop' statements, which require that users specify a counter to index the array, initialize and increment its value, while checking its value against the array bounds. The 'foreach'-like language construction contributes to reduce the 'hard mental operations' dimension in most cases.

The *hidden dependencies* dimension refers to cases in which two or more entities in a program are mutually dependent, but these dependencies are not explicit. A well-known example of this dimension occurs in spreadsheet cells: it is not always clear that a cell is a dependency to other cells, and that changing its value causes changes to these other cells' values. In the case of programming languages, allowing to share state between independent units of execution, for instance, may be the source of hidden dependencies in code. It should be also noted that the creation of new abstractions may increase hidden dependencies between the 'abstracted' entities, since higher level features usually depend on lower level entities that may be somehow related.

Premature commitment occurs when a user has to make persistent decisions before all required information is available. For instance, if a UML modeling application strictly enforces the underlying model consistency rules all the time, it does not allow users to 'sketch' a drawing of a model without satisfying all the mandatory conditions. Therefore, this application imposes a situation of premature commitment to its users. In the context of programming languages, an example of premature commitment occurs when a user needs to create an object from a class whose constructor requires a list of parameters which are not always known or available at the moment of object creation. The creation of a new abstraction may increase premature commitment when it forces the user to make early decisions in order to be able to use it.

The *progressive evaluation* dimension describes the ability to check the current state of work, even if only partially complete. An illustrative example of this dimension occurs with respect to dynamically versus statically typed programming languages: in general, it is easier to evaluate the current state of an incomplete program when using a dynamically typed language, which contributes positively to this cognitive dimension.

Provisionality refers to the notations support to 'sketching' or recording design options, even if there are constraints on the order of doing things (premature commitment). This dimension may also describe the possibility expressing parts of an object that is not fully defined yet. In object-oriented programming, for instance, an example of the provisionality dimension is the ability to create an abstract method in a class that will only be defined when creating a concrete subclass that implements it.

Role expressiveness refers to how easy it is to infer an entity's features and goals. For instance, PHP provides a function named *dl()*, a poor naming choice which hinders its role expressiveness[2]. The secondary notation dimension may contribute to increase a notation's role expressiveness, as does a higher abstraction level. Closeness of mapping may also affect positively this dimension, and vice-versa.

Secondary notation denotes the ability to express information using syntax that escapes the formal notation. Examples of this dimension includes the use of comments or indentation to add meaning to the representation. In the case of visual languages, the spatial layout is also a form of secondary notation commonly explored by expert users to represent additional information.

Viscosity metaphorically refers to the physical properties of fluids that determine its 'resistance to change'. In the context of programming languages or applications, for example, it means the amount of work needed to accomplish a certain modification task. Text editors, for instance, usually offer a search-and-replace abstraction that allows users to perform a number of changes with less work. This example also illustrates how the abstraction dimension may contribute to reduce viscosity. In addition, aspect-oriented programming may be regarded as a technique with the potential to reduce viscosity, since cross-cutting aspects are implemented in a centralized form. Therefore, a program's features implemented as aspects may be considered less 'viscous'.

Lastly, the 'visibility' dimension corresponds to how difficult it is to identify or access an object or piece of information. For instance, if a program's feature is only accessible by going deeply into a sequence of screens or options,

[2]The dl() function dynamically loads a PHP extension.

it has low visibility. In addition, if a language or API 'hides' information about a feature in documentation pages that are accessible only after navigating through various intermediate steps, it is also an example of low visibility.

3.2.2
Cognitive dimensions and APIs

Differently from Semiotic Engineering, the CDNf has a long tradition of supporting usability evaluation of APIs. The first well-known studies using the framework to evaluate programming languages and APIs have been developed at Microsoft by Steven Clarke and his group(32, 14). These studies introduced a new approach to investigating API design and provided interesting insights about the potential cognitive impact caused by specific API features.

In their studies, the group at Microsoft developed their own set of cognitive dimensions, mostly based on the original dimensions with some adaptations. Their proposed adaptation included the following list of dimensions:

- Abstraction Level: what are the minimum and maximum levels of abstraction exposed by the API, and what are the minimum and maximum levels usable by a targeted developer;
- Learning Style: what are the learning requirements posed by the API, and what are the learning styles available to a targeted developer;
- Working Framework: what is the size of the conceptual chunk needed to work effectively;
- Work-Step Unit: how much of a programming task must/can be completed in a single step;
- Progressive Evaluation: to what extent can partially completed code be executed to obtain feedback on code behavior?
- Premature Commitment: to what extent does a developer have to make decisions before all the needed information is available?
- Penetrability: how does the API facilitate exploration, analysis, and understanding of its components, and how does a targeted developer go about retrieving what is needed;
- API Elaboration: to what extent must the API be adapted to meet the needs of a targeted developer?
- API Viscosity: what are the barriers to change inherent in the API, and how much effort does a targeted developer need to expend to make a change;

- Consistency: once part of an API is learned, how much of the rest of it can be inferred?
- Role Expressiveness: how apparent is the relationship between each component and the program as a whole?
- Domain Correspondence: how clearly do the API components map to the domain? Are there any special tricks?

From the above list, we can see that some of the original dimensions have been kept (e.g. premature commitment), others have been renamed (e.g. ‘closeness of mapping’ to ‘domain correspondence’), and others have been created (e.g. ‘working framework’). According to the authors, these changes have been proposed to better accommodate their requirements at Microsoft, and also to adopt a vocabulary that would be more easily interpreted by their team. In this research, we refer to their classification as an example of extension and reinterpretation of the cognitive dimensions, but keep with the original terminology and conception of the CD’s as proposed by Green.

Despite the successful use of the CDNf in the evaluation of programming languages, APIs and other kinds of notations and devices, the framework also has its shortcomings. For instance, as Petre states: “*our biggest struggle centred on scope: where does the notation end, and how much does a CDs analysis include? Many of the big unresolved issues concern not just dimensions of notation, but also cognitive issues of notations in use, and of their context of use*” (86). In her work, she proposes the use of facet analysis to compensate for some of these issues.

One of the envisioned advantages of using the CDNf under a Semiotic Engineering perspective is that, together, they enable the analysis of a notation in a more completely specified context of use, because it includes explicit intentionality and communication aspects. This provides a richer situated view of the notation with respect to the cognitive issues involved, as opposed to a generic unsituated analysis of the notation by itself. Therefore, a combined semiotic and cognitive approach has the potential to circumvent the limitations of an isolated adoption of the CDs, as identified by Petre.

Next chapter illustrates how the CDNf has been used together with Semiotic Engineering in the definition of the research method to perform the empirical studies. The results of the research studies are detailed in chapter 5.

4
Methodology and Empirical Studies

This chapter provides an overview of this thesis' research design and methodological approach, based on the concepts and theoretical support described in chapter 3. It also discusses the evolution of the research and its main phases, enabling a more detailed comprehension of the steps that conducted to the results to be introduced and analyzed in chapter 5.

4.1
Research approach

In order to discuss the research approach, we begin by recalling the research question introduced in chapter 1: *How communicative and cognitive aspects of the design of programming languages and APIs influence their usage by programmers, leading to errors or difficulties in writing and understanding code ?*

This is a research question of an 'open' nature, since it addresses the inquiry of general knowledge about an object of study, aiming to achieve a deeper comprehension of a class of problems. Due to its nature, the proposed research question is suited to a qualitative approach. It contrasts with 'closed questions', those which aim to test a specific hypothesis, searching for answers regarding the effectiveness or the efficiency of a method or technique to solve a specific problem, for example. Closed questions are typically addressed by a quantitative approach to research, based on statistic studies over large samples of collected data, in order to confirm an hypothesis and allow predictive results from studies.

Qualitative research methods, in contrast, search for a deeper understanding of observed phenomena. Researchers are an integral part of qualitative inquiry, since they confer meaning to data collected and analyzed. As such, subjective interpretation is not only allowed, but also expected. Researchers bring their background, biases and reflections to the investigation process, as qualitative research inherently deals with the meanings that a person (or a group of people) extract from a human perspective of the investigation.

In qualitative research, theory commonly works as the 'lens' that provides ontological and epistemic support to the collection and analysis of data. It provides guidance to researchers in determining what should be analyzed, how the analysis should be performed and the nature of the meanings that may be extracted from data. In general, the researcher performs an inductive analysis of data, going from particular details from observed evidence to more general themes and findings, in a 'bottom-up' approach. Therefore, it usually involves raw data collection from the 'observed' environment, like, for instance, interviews, observations, documents, recordings, images, and so on.

Due to the openness of qualitative research questions, this type of investigation commonly involves the selection and combination of existing qualitative methods. Adaptation of selected methods to better fit the problem in question is also a common practice. The combination and adaptation of methods can also be regarded as the generation of new methods, that can be possibly used in other contexts of investigation.

A qualitative research project usually involves an iterative process of construction and refinement of the method utilized by the researcher. The initially selected method (or combination of methods) determines the early stages of data collection and analysis and, as the researcher develops a deeper comprehension of the observed phenomena, new procedures or techniques may be added to the process. These cycles are usually associated with stages or phases of the investigation, which represent the evolution of the researcher's interpretation process in the direction of providing consolidated findings resulting from the analysis.

4.2
Research Design

In this section, we discuss the general principles that conducted the design of the research steps and procedures. The description of the research design strategy follows a qualitative approach, as explained in the previous section. As such, the discussion here is restricted to principles and procedures for this type of approach.

The design of a research project encompasses the definition of its main components: questions, goals, methods, theoretical framing, procedures, resources and so on. Among the procedures, the researcher has to determine the steps for data collection, recording, analysis and interpretation. In addition, the research strategy should also involve activities regarding its reliability and validity.

According to Creswell (87), qualitative validity involves employing procedures for checking for the accuracy of the findings, like triangulation, member checking, rich descriptions, use of negative or discrepant information, among others. Reliability procedures aim to check mostly for consistency among a group of researchers or across different research projects. Some examples of reliability procedures include cross-checking of transcripts and analysis (coding) among researchers, coordinated communication between team members, and precise documentation of research steps.

In the context of this thesis' research design, the main procedure for data collection was a systematic search of bug reports, as mentioned in chapter 1. The goal of this procedure was to collect empirical data that could possibly offer evidence of communicative breakdowns. In order to increase the probability of finding bug reports with significant breakdowns, the collection strategy defined was to search for bug reports classified by evaluators as 'not a bug' or 'not an issue'. The classification of a reported issue as 'not a bug' is a potential symptom of a mismatch between the designer's intent behind the software artifact and its interpretation by users.

As to data recording, the nature of empirical data collected facilitated this procedure, since it consisted mostly of textual data from Web sites. The languages selected as sources of bug reports (Java and PHP) offer public Web search engines for their bug tracking system database. The initial collection and recording of raw data involved the inspection of bug reports' contents. if the bug report looked interesting from the perspective of the research, part of its content was copied a text file, as a quick method to mark it for further analysis in the next iteration over data. A second step of analysis involved a more structured analysis and organization of data in spreadsheets, as will be detailed in section 4.3.

Data analysis and interpretation procedures are the 'core' of qualitative inquiry, since it usually involves a number of iterations in which the researcher reflects on collected data and partial results. Creswell suggests a bottom-up approach for data analysis in qualitative research which consists of seven 'layers', as depicted in figure 4.1. Despite the fact that the figure suggests sequential procedures, the author states that they are interrelated and not always visited in the order presented.

Data analysis procedures carried out in this research followed a similar approach to Creswell's model. They permeated the research process's iterations, mixing various activities with different levels of intensity over the course of the studies.

In order to provide an overview of the organization of research activities, we refer to a representation proposed by Brandão in the C&A4Q model (88), a theoretical Capture & Access model targeted at supporting the registration of

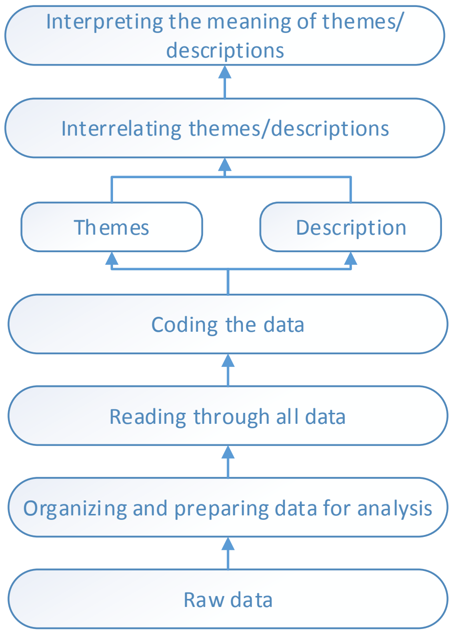

Interpreting the meaning of themes/ descriptions
Interrelating themes/descriptions
Themes
Description
Coding the data
Reading through all data
Organizing and preparing data for analysis
Raw data

Figure 4.1: Qualitative data analysis procedures (Creswell)

qualitative research procedures applied to HCI. This model proposes the characterization of qualitative procedures in two different contexts: the researcher's context, in which 'internal' activities are carried out by the researcher in the actual development of the research; and the scientific community's context, where part of the qualitative reliability and validity procedures may take place. Figure 4.2 depicts the top-level organization of the procedures carried out in the research.

The left part of figure 4.2 shows the researcher's context and the internal procedures of qualitative analysis and validation. The inner box shows the procedures of data collection and analysis, organized in iterative phases that follow the evolution of the method over the course of the research. In the right side of the researcher's context, we can see the validity procedures carried out along with the phases of collection and analysis: triangulation and negative/discrepant case analysis. The 'scientific community context' box in the figure shows the validity procedures carried out with the participation of other members from the Semiotic Engineering Research Group (SERG): peer debriefing and member checking. These procedures will be explained in section 4.3.

The first phase in the iteration chain of data collection and analysis consisted of the initial selection, collection and quick analysis of bug reports. They have been collected from bug tracking systems and their basic information was organized in text files for further analysis in the second phase.

In the sequence, phase 2 involved the first systematic classification of

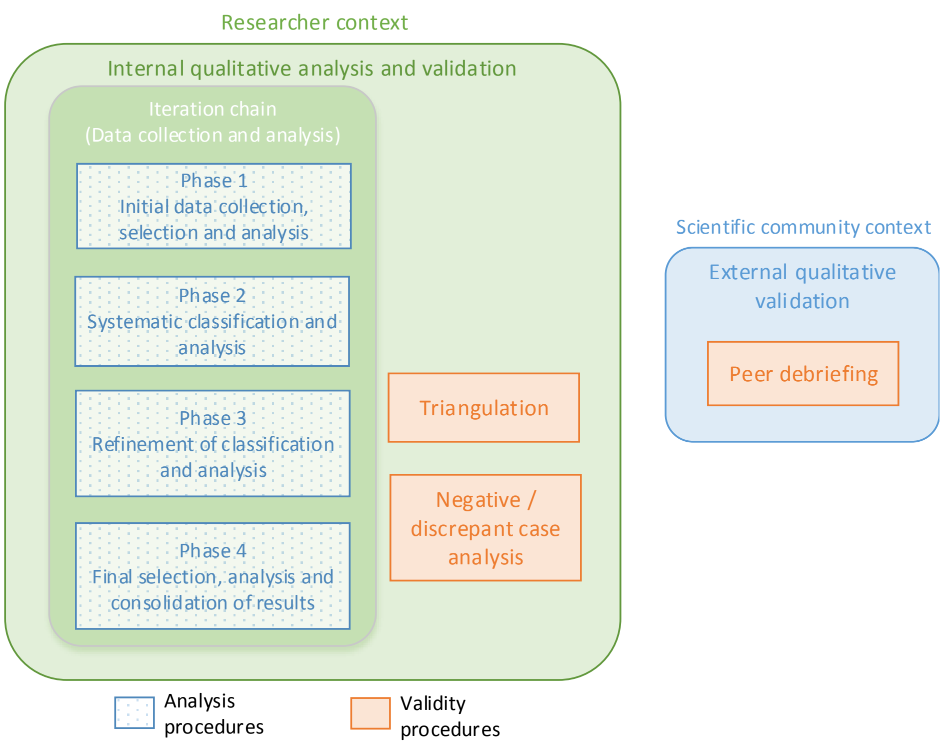

Researcher context
Internal qualitative analysis and validation
Iteration chain
(Data collection and analysis)
Phase 1
Initial data collection, selection and analysis
Phase 2
Systematic classification and analysis
Phase 3
Refinement of classification and analysis
Phase 4
Final selection, analysis and consolidation of results
Triangulation
Negative / discrepant case analysis
Scientific community context
External qualitative validation
Peer debriefing
Analysis procedures
Validity procedures

Figure 4.2: Overview of research procedures

bug reports, and also a refinement of the initial reports' selection in order to eliminate those that did not provide evidence as required by the investigation criteria.

After completing the first systematic classification, phase 3 addressed the refinement of the initial classification, adding new aspects to the categorization process and changing some interpretation aspects of existing criteria.

The final stage of the research comprehended further selection of bug reports for deeper analysis and classification of a more representative dataset. The goal was to focus on a more significant dataset and identify existing relations and themes among these reports. New categories have been added to the classification, and a final consolidation of results has been carried out to provide the findings described in chapter 5.

We discuss the research phases and procedures in greater detail in the next section.

4.3 Empirical studies

This section provides details about the empirical studies performed during the research project, its main phases and activities. It also describes the procedures carried out with the goal to collect and analyze data, leading to the results detailed in chapter 5. As introduced in the previous section, the research work comprehended four phases of iterations consisting of data collection, analysis and registration of results. We discuss details about each phase's activities, and then conclude with considerations about the conducted research in section 4.4.

4.3.1 Phase 1 – Initial data collection, selection and analysis

Phase one started with a 'broad-brush' inspection of a large number of bug reports from the PHP[1] and Java[2][3] bug tracking systems. As previously mentioned in chapter 1, these languages have been chosen as the primary source of empirical data due to their popularity and the availability of a large number of bugs registered in their issue tracking systems, including almost two decades of each language's history. Additionally, the availability of source code in both cases contributed to the decision of choosing these languages, since it would allow a deeper investigation of the APIs' issues and evolution over time, if necessary.

The initial search for common issues regarding the languages and their APIs encompassed the inspection of their 'change log' between versions, especially looking for 'deprecated features', which could provide indication of potential topics of interest for the research. In this context, the website for PHP's request for comments (RFCs)[4] is an interesting source of information about major changes to the language and its APIs. These RFCs' discussions provide evidence of the rationale behind changes that occurred over the course of the language's evolution. These changes frequently resulted in the 'deprecation' of features with negative impact on users, either by causing misunderstandings or by being rejected by the community of programmers.

The collection of raw data involved a number of sessions for the inspection of bug reports. Systematic searches have been performed in each language's issue tracking systems, filtering by reports classified as 'not a bug' or 'not an issue', depending on the adopted terminology. In both cases, search results have been sorted by bug ID in ascending order, which also implied a chronological order of their creation. As previously explained, the filter's goal was to obtain evidence of mismatches between design intent and users' interpretations and/or expectations, which could be associated with communicative issues.

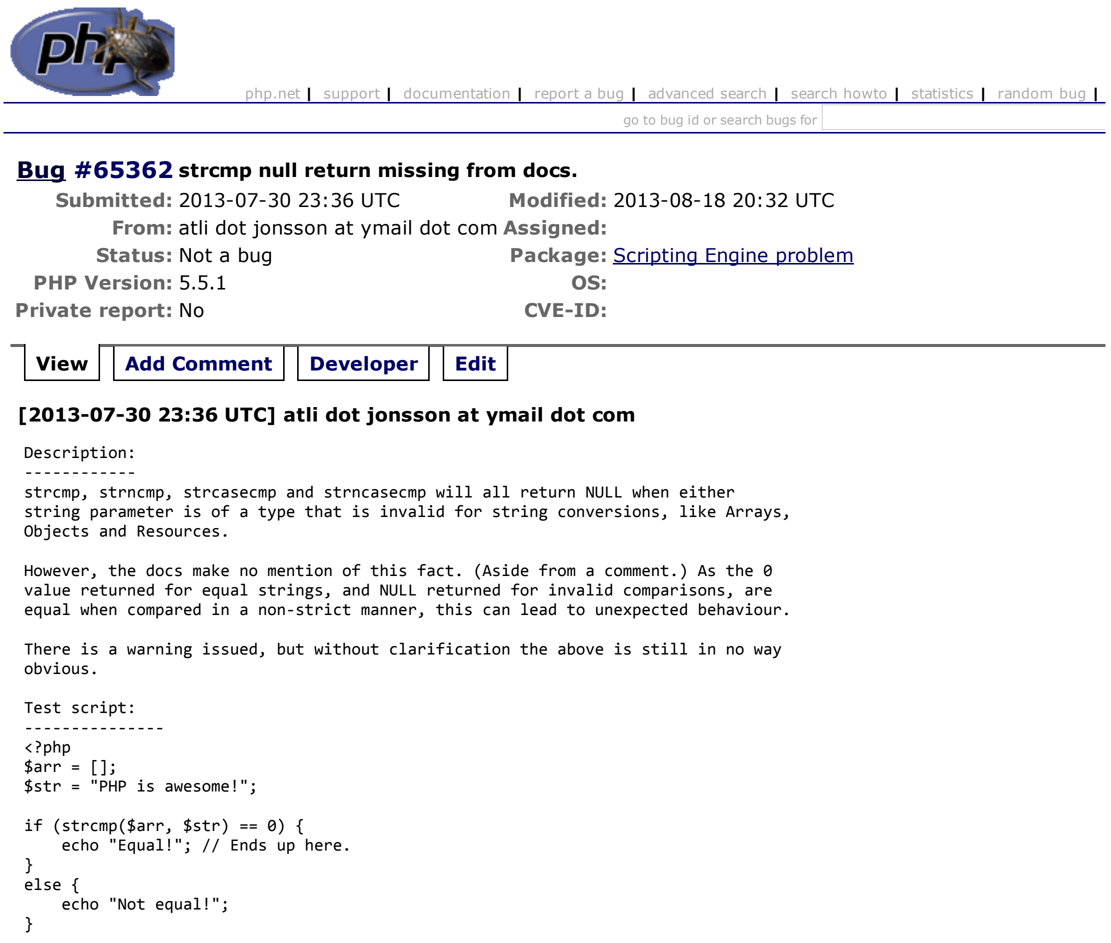
[1]`https://bugs.php.net`

[2]`https://bugs.openjdk.java.net`

[3]The 'openjdk' bug database provides the complete history of bugs for the official Java distribution, including the period preceding its change into an open-source project, formerly maintained by Sun Microsystems and currently by Oracle Corporation.

[4]`https://wiki.php.net/rfc`

The collection process encompassed the inspection of 6k bug reports for each language, divided in two groups: the oldest 3k bug reports, followed by the 3k newest bug reports (as of mid-2014), resulting in the scanning of 12k bug reports ('not a {bug,issue}'). The goal for the division in groups of oldest and newest bugs was to identify possible variations of common issues over the course of language evolution. In order to illustrate the nature of data inspected, figures 4.3 and 4.4 depict bug reports from PHP[5] and Java[6], respectively.


php.net | support | documentation | report a bug | advanced search | search howto | statistics | random bug |
go to bug id or search bugs for
View
Add Comment
Developer
Edit

Bug #65362 strcmp null return missing from docs.

Submitted:	2013-07-30 23:36 UTC	**Modified:**	2013-08-18 20:32 UTC
From:	atli dot jonsson at ymail dot com	**Assigned:**	
Status:	Not a bug	**Package:**	Scripting Engine problem
PHP Version:	5.5.1	**OS:**	
Private report:	No	**CVE-ID:**	

[2013-07-30 23:36 UTC] atli dot jonsson at ymail dot com

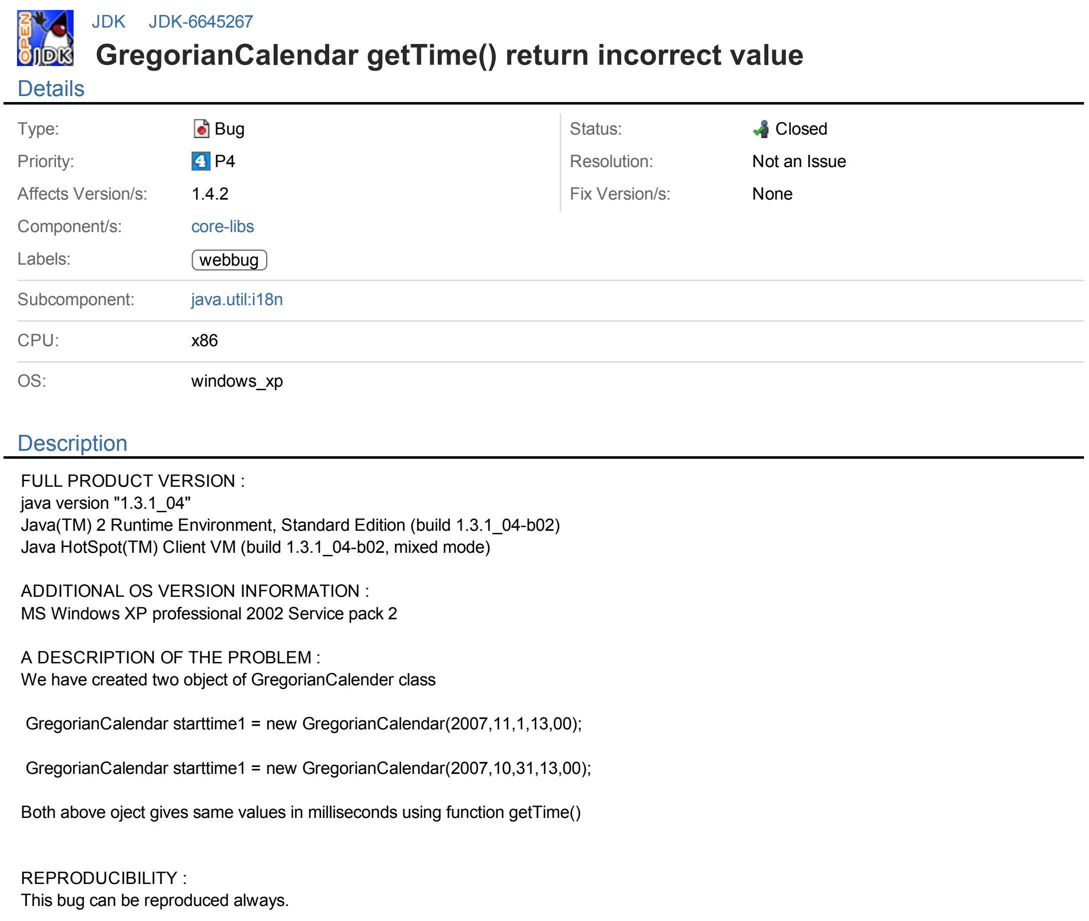
```
Description:
------------
strcmp, strncmp, strcasecmp and strncasecmp will all return NULL when either
string parameter is of a type that is invalid for string conversions, like Arrays,
Objects and Resources.

However, the docs make no mention of this fact. (Aside from a comment.) As the 0
value returned for equal strings, and NULL returned for invalid comparisons, are
equal when compared in a non-strict manner, this can lead to unexpected behaviour.

There is a warning issued, but without clarification the above is still in no way
obvious.

Test script:
---------------
<?php
$arr = [];
$str = "PHP is awesome!";

if (strcmp($arr, $str) == 0) {
    echo "Equal!"; // Ends up here.
}
else {
    echo "Not equal!";
}
```

Figure 4.3: Example of PHP bug report

Despite the large number of bug reports inspected, only a small percentage of the initial set has been selected, since the goal was to analyze in deep detail only bug reports that could provide quality evidence for the study, considering the communicative perspective of the approach. Therefore, the following criteria has been adopted to discard most items during the initial bug report database scan:

- bug reports containing incomplete or inadequate information (invalid reports);

[5]`https://bugs.php.net/bug.php?id=65362`
[6]`https://bugs.openjdk.java.net/browse/JDK-6645267`


JDK JDK-6645267

GregorianCalendar getTime() return incorrect value

Details

Type:	Bug	Status:	Closed
Priority:	P4	Resolution:	Not an Issue
Affects Version/s:	1.4.2	Fix Version/s:	None
Component/s:	core-libs		
Labels:	webbug		
Subcomponent:	java.util:i18n		
CPU:	x86		
OS:	windows_xp		

Description

```
FULL PRODUCT VERSION :
java version "1.3.1_04"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_04-b02)
Java HotSpot(TM) Client VM (build 1.3.1_04-b02, mixed mode)

ADDITIONAL OS VERSION INFORMATION :
MS Windows XP professional 2002 Service pack 2

A DESCRIPTION OF THE PROBLEM :
We have created two object of GregorianCalender class

 GregorianCalendar starttime1 = new GregorianCalendar(2007,11,1,13,00);

 GregorianCalendar starttime1 = new GregorianCalendar(2007,10,31,13,00);

Both above oject gives same values in milliseconds using function getTime()


REPRODUCIBILITY :
This bug can be reproduced always.
```

Figure 4.4: Example of Java bug report

– bug reports that clearly revealed user's lack of knowledge about programming or other computer science concepts (e.g. floating point representation and operations);

– 'old' bugs that had already been fixed in versions of the language available at the time of the report;

– issues associated with the language's tools and runtime environment;

– issues associated with language features which are too specific (e.g. Java GUI classes);

– issues associated with related technologies or applications (e.g. Apache web server);

– users' mistakes when reporting the bug, in which the error was soon acknowledged by further user comments;

– reports related to inherent limits of the technology;

– issues closed without justification by the evaluator;

– performance-related issues;

– confusing bug reports that could not be clearly interpreted.

During the first inspection of raw data, information about selected bug reports has been copied into a text file, including bug ID, date and title, for further analysis. In addition, interesting excerpts and annotations were added to the registration of the bug report in the text file. The first data collection resulted in 393 PHP bugs and 289 Java bugs selected for further investigation and initial classification.

4.3.2
Phase 2 – Systematic classification and analysis

In this phase, the resulting set of bug reports selected from phase 1 went through the first iteration of systematic analysis. The annotated text files (one for each language) served as the basis for a new inspection of each bug selected. This time, information about bug reports was stored in a more structured way by using a spreadsheet application.

For each bug report selected in the first phase, the report's content was revisited, which also resulted in a new selection process. The second round of analysis, influenced by a refinement in the selection criteria, resulted in the elimination of a number of bugs from the first set. As such, the resulting spreadsheets at the end of this phase contained 304 PHP bugs and 161 Java bugs.

The initial systematic analysis of bug reports comprehended the identification of users' breakdowns, in the attempt to classify the type of failure associated with the report (as seen in chapter 3). The first approach involved the classification of bug reports with respect to existing categories from Semiotic Engineering, including the type of communicative failures and the metacommunication template components. In addition, a categorization of topics (coding) has been carried out to identify recurrent themes in the reports (as illustrated in the approach depicted by figure 4.1). The goal was to identify communication failures associated to users' reception of the designer's message. After the classification of failures, the next step was the identification of the metacommunication template components associated with the failures, describing which parts of the designer's intent encoded in the artifact's design could have affected the user's interpretation of its meanings. Chapter 5 provides a more detailed explanation about this classification procedure.

The first systematic classification and analysis of collected data resulted in storing the following information in spreadsheets, for each bug selected:

– bug ID;

- creation date;
- title;
- short description highlighting the main aspects of the situation involved;
- selected excerpts from users' or evaluators' discourses;
- related links (other bug reports, related documentation, user groups' discussions, tutorials, etc.);
- tags: initial coding of main themes involved in the reports;
- API/language: flag indicating if the issue is mostly related to the language core or its API/library;
- communication failure classification, based on CEM method's tags detailed in 3.1.10;
- metacommunication template components (see 3.1.6);
- cognitive dimensions associated with the issue (see 3.2.1);
- programming concepts associated with the report;
- language-specific features involved in the report;
- API knowledge types: patterns of knowledge for API documentation, as defined in (64) (see below).

In their work, Maalej and Robillard (64) provide a taxonomy of 12 'patterns of knowledge' contained in API reference documentation. According to the authors, the taxonomy "*can be used to help practitioners evaluate the content of their API documentation, better organize their documentation, and limit the amount of low-value content. They also provide a vocabulary that can help structure and facilitate discussions about the content of APIs.*" Table 4.1 summarizes the taxonomy proposed by the authors.

In the process of bug report analysis and classification, this taxonomy for API knowledge types provided conceptual and epistemic support to the framing of issues described in reports. It also served the purpose of internal triangulation of the concepts and themes identified during the analysis, providing an additional element to be considered in the interpretive and reflective process of qualitative research.

Over the course of the analysis, an additional procedure adopted was the creation of short programs, in order to check if the language or API feature worked as described by the user's report. The code usually consisted of the example provided by the user, sometimes with additional tests. This also worked as a validity procedure, since it double checked the quality of the collected evidence, especially when the report raised doubts about its

API Knowledge type	**Description**
Functionality and behavior	Describes what the API's functionality or features.
Concepts	Explains the meaning of terms or domain concepts used by the API.
Abstraction	Types and availability of abstraction mechanisms.
Directives	Directives are clear contracts.
Purpose and rationale	Explains the purpose of providing an element or the rationale of a certain design decision.
Quality attributes and internal aspects	Describes quality attributes of the API, also known as non-functional requirements, for example, the performance implications.
Control-flow	For example, describes what events cause a certain callback to be triggered.
Structure	Describes the internal organization of a compound element (e.g. important classes, fields, or methods) or how elements are related to each other.
Patterns	Describes how to accomplish specific outcomes with the API, for example, how to implement a certain scenario, how the behavior of an element can be customized, etc.
Code examples	Provides code examples of how to use and combine elements to implement certain functionality or design outcomes.
Environment	Describes aspects not related to the API directly (e.g., compatibility issues, differences between versions, etc.).
References	Pointers to external documents (hyperlinks, 'see also' reference, standards, manuals, etc.)

Table 4.1: Patterns of API knowledge type

correctness. It also helped to check if the issue had already been resolved by the language team. In summary, 49 short PHP programs were written to check issues from bug reports. As to Java, 13 programs were created, including tests for 32 cases collected from bug reports. Listing 4.1 illustrates one of these short programs.

Listing 4.1: Example of a PHP short program to check a bug report

```
<?php
//bug 60650
try {
    $date = new DateTime('2011-02-31');
    print $date->format('Y-m-d') . "\n";
    $date = new DateTime('2011-02-32');
    print $date->format('Y-m-d');
} catch( Exception $e ) {
    print $e;
}
```

During the classification performed in this phase, some issues described in bug reports have been checked against other sources like, for instance, the Stack Overflow (SO) Web site[7]. SO is a public forum for technical questions and answers, mostly about programming topics. Users can vote for best questions

[7]`http://www.stackoverflow.com`

and answers, and popular items can be identified by their score and number of views. As users vote for an answer, it goes ‘up’ in page results.

Actually, the check for topics identified in a bug report usually involved Google searches for the main terms describing the issue. However, the top ranked search results frequently included SO’s questions and answers, due to their popularity and significance to the subject. The main goal of this type of check was to verify if a bug report’s issue was representative of a relevant class of problems, concerning the API’s features involved. In the case of Stack Overflow, a high number of views and ‘upvotes’ for a question or answer distinguishes ‘popular’ topics from others of less interest.

4.3.3 Phase 3 – Refinement of classification and analysis

The third stage of the research corresponded to refinements in the classification of bug reports, by revising each bug’s details according to changes in the categorization criteria. The main goal was to double check the classification’s consistency and reflect on the evolution of the method and its findings, deriving from insights acquired during the process of investigation.

The procedures carried out during this phase can be summarized as follows:

- classification of the nature of the main issue involved (e.g. date and time operations, etc.);
- separation of excerpts containing user and evaluator discourse, in order to facilitate the analysis of each part’s arguments and reasoning;
- references and comments about documentation related to the main issues associated with the report, to support further reasoning;
- creation of a new field to include a more detailed analysis of the report, with comments about the nature of the problem behind the issue and its interpretation;
- separation of communicative failures in subcategories (complete, partial, temporary), and refinement of applied criteria;
- refinements in the classification of cognitive dimensions and metacommunication template components.

In addition, this phase involved the recording of a two-hour session of bug report analysis, through the use of a capture and access system (CAS). This was part of a cooperation with another PhD research project being carried out at the time (88), also from a member of the Semiotic Engineering Research

Group (SERG). Brandão's research goal was the investigation of a model to apply capture and access technology to support qualitative research.

The use of the CAS system allowed the simultaneous recording of researchers' audio and video, also including the screen capture for user interaction with the various applications involved (spreadsheet, Web browser, text file, operating system console and so on.). The system also allowed the researcher to apply 'tags' to the procedures that corresponded to the main steps of the analysis method. This mechanism enabled easy tracing and recovery of the most relevant parts from the research session.

The final product resulting from this capturing session was a multimedia document that served a twofold purpose: 1) it has been used as the basis for discussion and analysis among SERG members, which generated another CAS document itself, and worked as a 'peer debriefing' validity procedure; 2) it allowed a reflection about the research process and its procedures, providing greater awareness about its nature and possible improvements.

4.3.4
Phase 4 – Final selection, analysis and consolidation of results

The final phase consisted of a new selection of bug reports and a more detailed categorization of their characteristics. A new category has been added to the analysis in order to characterize the main effects on users that resulted from API metacommunication, as described later in chapter 5.

In addition, we decided over the course of the investigation to concentrate on issues more clearly associated with the languages' APIs, as opposed to problems deriving from the languages' basic features. This new perspective resulted from regarding basic language knowledge a precondition for users to be able to interpret API metacommunication properly, eliminating bug reports associated with lack of user's knowledge about basic use of programming language. This does not mean that programming languages cannot (or should not) be the object of a communicative approach to their investigation, but we decided to focus on the metacommunication aspects of APIs, assuming that the programming language works as a 'common ground' between interlocutors.

Therefore, a new selection of bug reports concentrated on keeping bug reports clearly related to API issues, resulting on 152 PHP reports and 146 Java reports. A new round of analysis and validation procedures also investigated the significance of each report's main issues, by triangulating with external sources like Stack Overflow, programming forums, technical references and code checking tools, which are relevant sources of information about the 'popularity' of certain issues. Therefore, bug reports involving topics that could possibly reflect isolated cases have been discarded. The final configuration of selected bug reports comprehended 104 from PHP and 96 from Java (200 reports).

The final refinement in the analysis and classification of bugs has been supported by the use of an additional tool called *Elastic Search*[8], an open source document store with full-text search capabilities and sophisticated aggregation features. This tool allowed more elaborated searches and the use of data visualization techniques that supported new perspectives on the nature of collected evidence. This procedure offered new insights and contributed to improvements in the research work. It also enabled the use of queries to check the consistency of manually generated data stored in the spreadsheets (e.g. categorization and tagging). A number of inconsistencies have been identified and adjusted by using this mechanism, which also worked as a validity procedure for qualitative research.

This search and analysis process involved the creation of various scripts to automate the main tasks that were intensively repeated as changes occurred. Python scripts have been created to index data from spreadsheets (exported as CSV files) into the Elastic Search (ES) engine. Queries on indexed data have been performed by using ES's JSON[9]-based query language, which also returns results in JSON format by default. An ES plugin offers a Web interface that allows easy query execution and analysis of results.

In order to provide better visualization and a more detailed investigation of results, other Python scripts have been developed to automate ES queries and data conversion to other formats. For instance, query results have been converted from JSON to CSV in order to generate charts (illustrated in chapter 5). Other conversions allowed the use of data in more dynamic visualization mecanisms, based on the D3 (Data-Driven Documents) Javascript library[10]. Figure 4.5 illustrates an example of data visualization using D3, which allowed dynamic navigation of aggregated data to support better reasoning and data analysis. The picture shows only the 'top level' visualization, which allowed zooming into inner categories by clicking in the corresponding circles. In spite of being an interesting resource for data visualization, this type of 'dynamic chart' is not appropriate for static visualization in a textual document. For this reason, chapter 5 provides other types of visualization to support the analysis of results.

Additionally, shell scripts were used to automate a number of the tasks, mostly calling Python scripts to index data, perform queries and convert

[8]`http://www.elasticsearch.org`
[9]`http://json.org/`
[10]`http://d3js.org/`

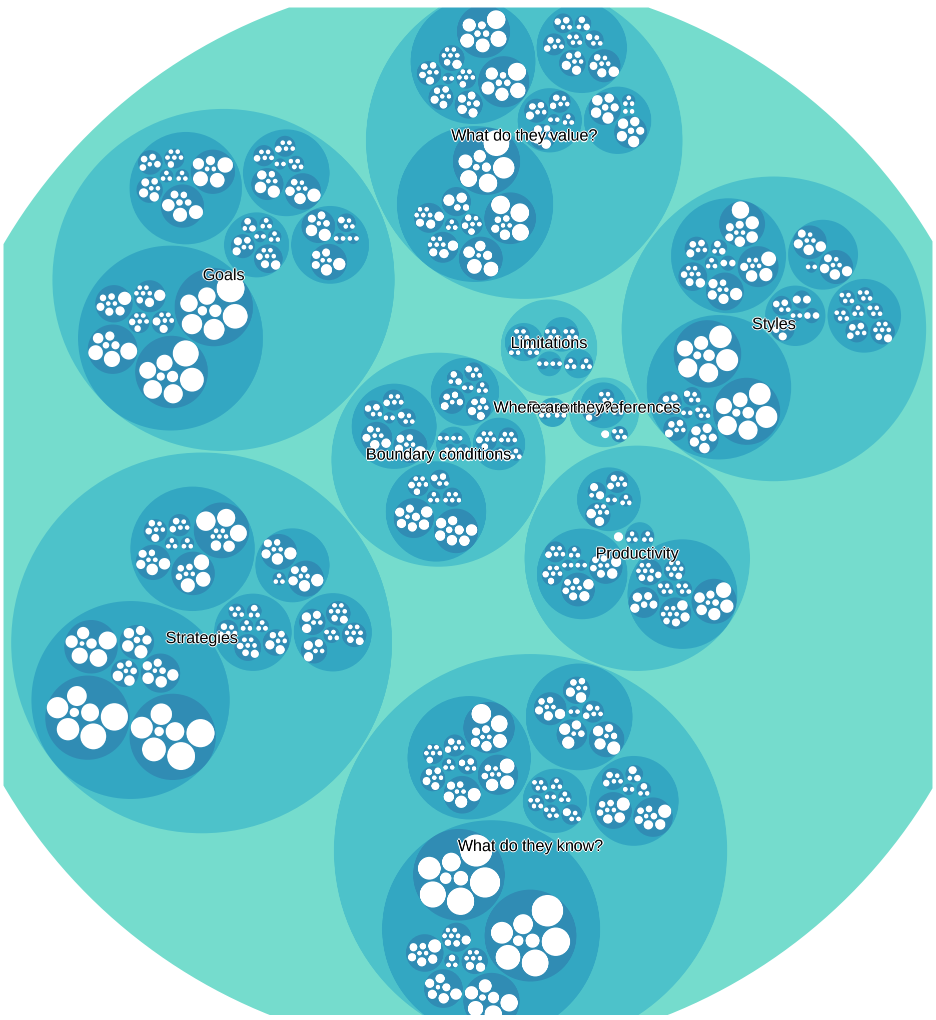


Figure 4.5: Example of dynamic data visualization

results. The scripts were a helpful resource to facilitate the execution of many cycles of tests and adjustments of data and scripts over the course of this research phase.

4.4
Considerations about the research

The research approach described in this chapter has a typical qualitative nature, in which the object of investigation is gradually constructed and refined. In this process, collected data influences the refinement of methods used in the research, and inspires new hypothesized meanings to be potentially extracted from them. In turn, as methods and theoretical tools are refined, they lead to new insights and perspectives over the collected data. This iterative nature of the research process serves the acquired knowledge from one iteration as input to the next one, and previous findings and interpretations end up being under constant revision.

The qualitative researcher is also part of the process, as an additional instrument of research. Differently from a quantitative researcher, who applies well defined and rigorous statistical, mathematical or computational techniques to analyze data, the qualitative researcher usually refines the object of research and methods as part of the investigation process. In hindsight, it is clear that the initial analytical tools and methods in the research presented in this volume have been adapted under the influence of the knowledge acquired from the results of data inspection and analysis. Conversely, the refined methods have provided new perspectives and helped the extraction of new meanings from empirical data.

The perception of this iterative and incremental nature of the qualitative research process helps to explain the next chapter's contents. Section 5.2 describes the epistemic tools for API design and analysis, which have been formulated over the course of the research, based on the theoretical support of Semiotic Engineering and the Cognitive Dimensions of Notations. The tools' initial conception helped to organize the analysis of collected data, but the process of analysis also influenced the shaping and refinement of these tools. Therefore, the tools are part of the applied method and, at the same time, a contribution of the research.

Chapter 5 analyzes the results of the research, encompassing the epistemic tools mentioned above and the qualitative findings from the studies described in this chapter.

5
Analysis of Results and Findings

This chapter presents analytical results from the qualitative studies described in the previous chapter. These results include the components of an epistemic framework to support API design and analysis from a communication perspective, and the findings that contribute to a deeper comprehension of communicability and cognitive aspects that influence API design and use.

Section 5.2 describes the epistemic framework components organized along three major communicative dimensions: intent, effects and failures. In the sequence, section 5.3 presents qualitative findings of the studies and discusses their implications in the context of API design.

Before analyzing the studies' results, section 5.1 introduces concepts that contribute to a better comprehension of the discussion promoted in the remaining sections of this chapter.

5.1
Considerations about the 'active programmer' and abductive reasoning

In the 1980's, Carroll and Rosson introduced a concept named "the paradox of the active user" (89). They concluded from their studies that users frequently show conflicting behavior when learning a new technology, summarized by two paradoxes: motivational and cognitive.

The motivational paradox corresponds to the "production bias" users show when learning a new technology: they try to be productive in using the techonology as quickly as possible, and this reduces their motivation to spend time just learning about it. Therefore, they try to learn the minimum necessary for them to use the technology, and keep on with this acquired basic knowledge.

The cognitive paradox is associated with the "assimilation bias", which states that people apply previous knowledge and experience to interpret new situations that occur when learning about new technology. In some cases, this bias may contribute to misleading comparisons between 'old' and 'new' knowledge, which may hinder users' learning process.

The main implication of these paradoxes is that, in general, users tend to learn a new technology by a trial-and-error approach, rather than spending time 'reading the manual' and taking similar actions with special focus on learning. Considering that programmers are a special class of users that need to learn a number of technologies and tools (new APIs included), this general principle also applies to them.

User behavior as described by Carroll and Rosson can also be viewed in terms of *abductive reasoning*, a concept used by Semiotic Engineering (and Semiotics in general) to model users' interpretive process when assigning meaning to signs, as explained in (90). Abductive reasoning, as opposed to deductive reasoning, consists of the formulation of plausible hypotheses in users' process of sense making (91). When positive evidence confirms the hypothesized rules, the interpretation process may be interrupted. Otherwise, when there is no confirmation of the initial hypothesis, a new hypothesis may be raised, and the process continues.

Based on this model, we can think of programmers as 'active users' that continuously formulate hypotheses that conduct their interpretation of signs when learning a new API, based on their previous experience, knowledge, biases and cultural influences. After obtaining evidence that confirms their hypothesis, they stabilize the interpretation of API's meanings, and keep with them until counter-evidence arises.

Learning a new API involves using a programming language and the extensions that the API provides on top of the language. Therefore, users formulate their hypothetical rules about API's behavior based on lexical, syntactical and semantic aspects of the API language. Most of these aspects are defined by the underlying programming language, which provides common ground for designer-user communication (as long as user has enough knowledge about the programming language).

However, an API adds new lexical and semantic elements on top of an existing language. These elements encode the designer's intent, which encompasses the pragmatic aspects of designer-to-user API metacommunication, as discussed earlier in this text. For this reason, designers should evaluate their decisions in terms of leading the users' abductive reasoning towards the formulation of productive hypotheses, preferably compatible with the aforementioned pragmatic aspects. To accomplish this goal, designers should make use of static, dynamic and metalinguistic signs in a coordinated fashion, considering the structure of API metacommunication space presented in section 5.2.

Figure 5.1 illustrates the elements that contribute to a user's abductive reasoning process when learning an API. Users inspect lexical, syntactical and semantic aspects of programming language and API, formulating hypotheses that add pragmatic aspects to their meaning construction process. Due to their lack of pragmatic component, programming languages depend on natural languages to complement their expressiveness in order to achieve full metacommunication. In the API context, natural language usually appears in the form of documentation, identifiers and runtime messages.

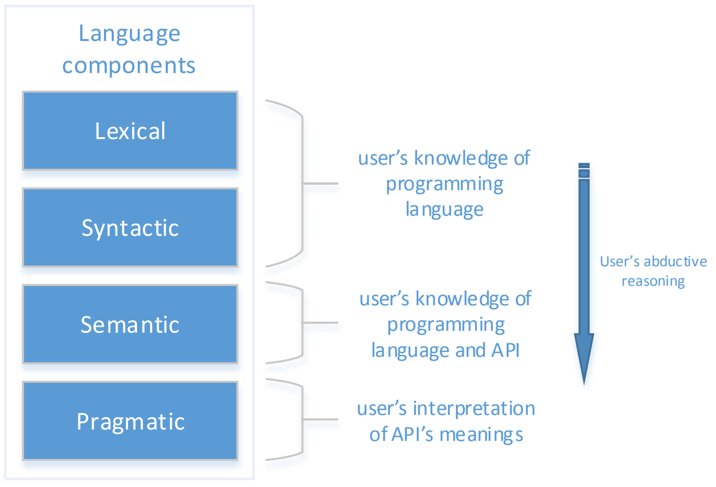

Language components
Lexical
Syntactic
Semantic
Pragmatic
user's knowledge of programming language
user's knowledge of programming language and API
user's interpretation of API's meanings
User's abductive reasoning

Figure 5.1: User's interpretation of API language

This communicative view of the active user paradox is not, by itself, a direct result from this research, since it has been previously described in (90). However, it is a central concept that permeates the study of APIs as human communication mediated by software. It contributes to a change of perspective, in which we may view API design as an attempt to conduct user's abductive reasoning to an interpretation that is aligned with the API's pragmatics. It is a useful concept that helps us to understand the remaining topics in this chapter.

5.2 Communicative dimensions and epistemic tools for API design and analysis

This section introduces the epistemic tools[1] for API design and analysis as part of the research results. As explained in section 4.4, these tools have been developed to carry out the analysis of empirical data from bug reports, and subsequently evolved in the process of the dataset analysis itself. Despite the tools' sequential presentation in this section, it is important to note that

[1]As described in 3.1.11, de Souza defines an epistemic tool as "*one that is not used to yield directly the answer to the problem, but to increase the problem-solver's understanding of the problem itself and the implications it brings about. (...) epistemic design tools are those that will not necessarily address the problem solution, but the problem's space and nature, and the constraints on candidate solutions for it.*" – (22) p. 33

their final configuration is the result of many iterations of investigation, with constant feedback from data to analytical tools and vice-versa.

The tools' presentation follows an organization of the communicative space around dimensions that were relevant to discuss API design, namely *intent*, *effects* and *failures*. This organization is inspired by speech act theory model (80), in which the speaker's intent (illocutionary act) may result in a range of effects on the listener (perlocutionary act), as explained in chapter 3. When a mismatch occurs between the sender's intent and the actual resulting effect, the diagnostic of the failures that may have taken place helps us to develop a better comprehension of the communication breakdowns and their causes, which are objects of primary interest in this research.

It should also be noted that there has been an effort to create these tools with special attention to their soundness and completeness. However, they reflect the subjective (albeit systematic) interpretation of collected data over a delimited set of empirical evidence. Yet, we believe that they provide a novel perspective on the research and development of APIs, as illustrated by the remaining sections in this chapter. The relevance, nature and quality of new knowledge resulting from the application of these tools will determine their future improvement and consolidation.

5.2.1 Intent

In general, an API can be viewed as a 'shortcut' that allows a programmer to accomplish a set of goals with less effort and faster. However, the artifact's design encodes intentional aspects that may be sometimes incompatible with programmers' goals, needs, preferences, and so on. This is a major motivation to frame the analysis of API design under a communication perspective, with especial emphasis on a pragmatic view of the subject. The conceptual distinction between intent and resulting effect allows us to examine this communication process in greater detail with respect to the factors that influence its outcomes.

Two conceptual tools compose the intent dimension of the API epistemic framework: the API metacommunication template and the API metacommunication elements.

API metacommunication template

As discussed in chapter 3, the metacommunication template is a key concept in Semiotic Engineering theory, and is defined as the following sentences ((22) pp. 96–97):

Here is my understanding of who you are, what I've learned you want or need to do, in which preferred ways, and why. This is the system that I have therefore designed for you, and this is the way you can or should use it in order to fulfill a range of purposes that fall within this vision.

The metacommunication template freely paraphrases the designer's intent encoded in a software artifact, and serves a twofold purpose. Firstly, it promotes the designer's reflection while analyzing the options and trade-offs that may exist in the design and construction of an interactive system, contributing to the organization of the design space. In addition, it guides the profiling of an interactive system's communicability, through the application of the Semiotic Inspection Method (SIM) (26). In this method, an independent researcher uses the template as a tool to evaluate inconsistencies and other communicability issues in the actual (as opposed to the intended) designer's discourse.

This research uses the metacommunication template apart from SIM. The template guided the communicative investigation of API design, and was instantiated to better suit the study of software interfaces. It also contributes to increase the designer's awareness of the aspects that may influence API design and use.

The template may be divided in parts that map the space of factors that influence designers' choices and decisions in the process of constructing a software artifact. These major components of the template have been initially defined to guide the analysis of empirical data. Over the course of the research, the refinement of these categories reflected the insertion of various elements that have been observed in the analysis of data, or that could be potentially observed.

The design space paraphrased by the template can be represented by four main questions, further divided into subcomponents that help the process of answering these questions and filling out the template.

Template part	**Components**
Who are the users?	What do they know? Where are they? In what conditions? What do they value?
What do they need or want to do?	Goals Boundary conditions
In which preferred ways?	Strategies Styles
Why?	Limitations Personal motivation Productivity

Table 5.1: Metacommunication template components

The subcategories of the template's components provide high-level guidelines for answering the main questions. They are quite general and can be applied to the more traditional HCI approach of Semiotic Engineering, as well as to the investigation of APIs. In this research we instantiated the main aspects that influence the answer to these questions in the context of API design. These aspects derived from the analysis and interpretation of empirical data, where instances of communication breakdowns provided insights to enumerate these aspects. Figure 5.2 shows the further instantiation of the metacommunication template and its adaptation to the API context.

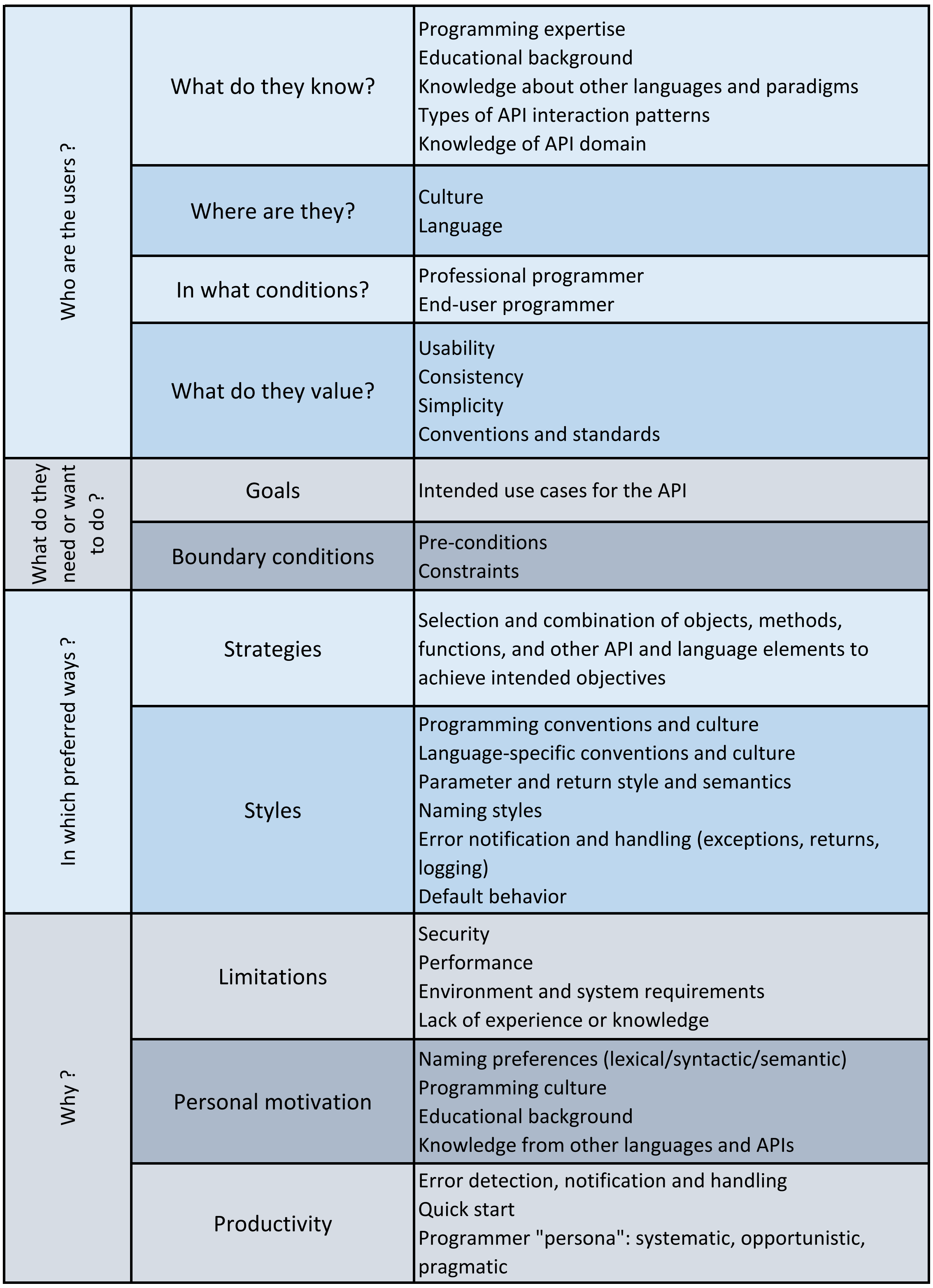
The following paragraphs describe each template component adapted to the study of APIs.

What do they know? In general, an API is closely associated with a specific programming language, except in cases of interfaces that support multiple languages (e.g. CORBA, Web services). In most situations, where the API has a specific underlying language, there is usually a typical profile of the target audience. For instance, the PHP language claims to be "*extremely simple for a newcomer, but offers many advanced features for a professional programmer*"[2]. From this statement, we can infer that novice programmers may be part of the language's typical audience. This aspect should be taken into account when designing an API for a PHP library, for example. Typical users' level of experience in programming, as well as exposition to previous languages and knowledge of the API's domain are relevant aspects to be considered. In PHP, for example, previous knowledge of C or Perl has been a factor of influence in the design of the language's features and libraries. The same applies to Java with respect to previous C++ knowledge. References to unusual or complex concepts that escape the intended audience profile and background knowledge should be object of careful thought in the design process.

It should be noted that the considerations about users' knowledge spans beyond the limits of programming concepts and language expertise. APIs frequently use concepts from 'ordinary' domains that most people are familiar with (at least, in the higher-level sense). For instance, many APIs include concepts from common domains like date and time, which is, by itself, a major source of problems related to their use, as will be discussed further in this chapter. This type of 'basic' user knowledge requirements should not be overlooked, at the risk of possible breakdowns in programmer's interpretation of the designer's message.

Where are they? Programming tools and techniques have a long

[2]`http://php.net/manual/en/intro-whatis.php`



Who are the users ?	What do they know?	Programming expertise Educational background Knowledge about other languages and paradigms Types of API interaction patterns Knowledge of API domain
	Where are they?	Culture Language
	In what conditions?	Professional programmer End-user programmer
	What do they value?	Usability Consistency Simplicity Conventions and standards
What do they need or want to do ?	Goals	Intended use cases for the API
	Boundary conditions	Pre-conditions Constraints
In which preferred ways ?	Strategies	Selection and combination of objects, methods, functions, and other API and language elements to achieve intended objectives
	Styles	Programming conventions and culture Language-specific conventions and culture Parameter and return style and semantics Naming styles Error notification and handling (exceptions, returns, logging) Default behavior
Why ?	Limitations	Security Performance Environment and system requirements Lack of experience or knowledge
	Personal motivation	Naming preferences (lexical/syntactic/semantic) Programming culture Educational background Knowledge from other languages and APIs
	Productivity	Error detection, notification and handling Quick start Programmer "persona": systematic, opportunistic, pragmatic

Figure 5.2: API metacommunication template

tradition of being developed in English speaking countries, especially in the United States. As a consequence, the English language is the dominant language in the software development 'world', and the cultural aspects that surrounds the language frequently show up in the design of software artifacts.

In addition to the inherent difficulties of programming, users which are not native English speakers face an extra barrier to use languages and APIs written in English. While the understanding of technical jargon in its more basic sense is a requirement for all programmers, adoption of more specific vocabulary or figures of speech in the design of APIs can raise the difficulty barrier even higher.

Designer's awareness about this aspect of API construction can avoid certain situations like the one found in the PHP API: the function that searches for the occurrence of a substring inside another (*strstr*) requires two parameters: *haystack* and *needle*. Although familiar to most people, this metaphorical use of identifiers is probably a poor choice when considering non English-speaking programmers, as it requires more vocabulary knowledge and introduces an extra level of signification in the programmer's semiosis.

Other aspects related to cultural and language differences may arise, for example, when dealing with accented characters, number separators and formats, local conventions and so on. Currently, most languages provide a good support for internationalization features, and many problems that may have been more frequent in the past tend to be less common. However, this is still a frequently overlooked aspect of API design, especially in the sense of not considering the pragmatic consequences of using figurative language with cultural references.

In what conditions? API users always have higher level goals that motivate the selection and use of these artifacts. However, the conditions in which the need for a certain API arises may vary. A possible categorization of these conditions is related to the type of activity in which the user is involved. For instance, professional programmers usually have formal education in computer science concepts. Even if that is not completely true, it is quite reasonable to assume that they are more familiar with programming concepts than non-professional programmers. On the other hand, end-user programmers or 'occasional' programmers form a growing community of API users, as the need of extending or customizing existing systems' functionalities increases. Therefore, this is an aspect that designers may consider while encoding their intent in the software artifact.

Is should be noted that this template component was not frequently associated with the problems identified in the bug reports analyzed. Due to the nature of the collected data in the empirical studies, there was little evidence of the conditions in which API use occurred. In addition, the languages involved in the studies (PHP and Java) are commonly used by professional programmers, which means that the empirical data collected probably contains more professional programmers than end-user programmers.

What do they value? This is a difficult aspect to generalize in an anticipated view of the intended audience of an API. It may be influenced by factors like personality, experience, preferences, and so on. However, designers can always base their decisions on common sense, or refer to the programming language community's culture to find a reasonable trade-off among the factors that influence this decision. For instance, consistency and simplicity are generally quality attributes that are valued by users. When complex domains are involved, simplicity can be a difficult goal to achieve, and can be a secondary objective when compared to completeness. Also, proper compliance to well-known standards and conventions can also influence users's acceptance and understanding of the design decisions behind an API.

The categorization of 'programmer personas' (92) can help the definition of a prototypical user profile, which may contribute to more informed decisions about the trade-offs to be made in the design, or even to the design of different 'facets' of the same API, each one intended to a variety of programmer persona.

Goals. This is one of the most relevant components in the design space of API metacommunication. Programmers commonly use APIs as a step to solve larger problems, and these are the goals that drive their motivation to learn and call an API's services. Among the possible outcomes when using an API, the most favorable one happens when programmers are able to effectively use it and achieve their objectives. Another possibility is when the software artifact does not fit the user's intended purposes, which can result in the search for an alternative. A third outcome occurs when programmers face a number of difficulties in attempting to understand and use an API, possibly 'stumbling' on it until giving up. This is probably the worst case, and as such should be prevented from happening. Therefore, designers may think of API design in terms of goals to be achieved, considering the envisioned scenarios in which it is intended to be useful to a large audience. Also, these scenarios should be explicitly communicated to users, so that they can easily identify the main purposes that the API is intended to serve, and even decide if the API is appropriate to the tasks to be accomplished.

Understanding and identifying common user goals that motivate the need for an API can be an important step to achieve a more successful design, as it may provide adequate support to a wider range of scenarios. However, more than enabling common strategies in the use of an API, designers should clearly introduce these strategies to users, and put them in the context of larger goals.

Boundary conditions. This template component is related to users' needs to achieve their goals, capturing the possible aspects that may constrain, limit or prevent users from successfully accomplishing their tasks. For instance, these include corner cases and exceptional conditions of API operations and pre-conditions that are common sources of surprise and unexpected outcomes that may prevent effective use.

This item is closely related to the previous one, as the identification of boundary conditions considers the possible barriers in the path of achieving users' goals. Also, not explicitly stating these boundary conditions may lead to the formulation of wrong hypotheses in user's minds about API's behavior, which is a common source of trouble.

Strategies. As already mentioned, programmers refer to APIs as a shortcut to achieve a higher goal in the context of a software project, be it a small application or a complex system. In both cases, programmers develop strategies to use API's services and combine them with other pieces of code, which usually include other API calls.

The development of programming strategies involves selecting APIs, their operations and features, as well as language constructs, to combine them in ways that provide the intended solution. Designers may provide scenarios to promote common strategies, making their vision about intended use cases explicit, and creating 'patterns' of use that make it easier for programmers to use the API effectively.

Styles. Programming languages with a large community of users commonly develop its own 'programming culture', in which some practices are usually seen as recommended patterns or styles. Common styling aspects involve identifier naming and semantics, error notification mechanisms, patterns or sequences of calls, and so on. Designers should take these preferences into account to support commonly adopted styles in the programming cultural environment in which the API will be introduced. Familiar styles may be easily recognized by users, contributing to reduce their cognitive load and conduct their sense making process to an interpretation which is compatible with the design intent.

Limitations. A programmer's goals, preferences and styles may be justified by constraints and restrictions that limit her choices, or impose requirements that must be met by the software artifact. For instance, programmers may choose to use an API in a non-standard way for security or performance reasons. System or environment restrictions may also limit the use of certain

API features. Also, programmers' lack of experience or knowledge can influence the way they use an API's features. For example, a novice programmer may choose to use only a subset of an API's operations that provides an easier interface, exposing only the most simple operations.

Personal motivation. Users' motivation to adopt certain strategies and styles can also be justified by personal preferences, guided by their previous knowledge, cultural aspects, educational background, and so on. Individual preferences may be difficult to support, as they may vary indefinitely among users. However, in some cases, certain patterns may arise from a community of users, which can also influence newcomers in their learning process. The observation of these patterns can serve as an indication of a possibly 'popular' preference, as people usually learn by examples, and tend to adopt styles and solutions they are familiar with.

For instance, many APIs that support regular expressions adopt Perl's syntax in their implementation, as it is probably the most popular syntax for this type of pattern specification. In this case, there is an easy choice for designing APIs that make use of regular expression syntax. However, in other domains the decision may not be so obvious, which requires some reflection and research to make an informed decision about trade-offs that may influence an API's popularity.

Differently from the HCI context, in which studies about users' profile and motivations are extensively carried out, in the API design domain this type of study is not common, and frequently not feasible. Therefore, API designers may 'anticipate' users' motivations based on thoughtful reasoning guided by the template's components, rather than just guessing.

Productivity. Aspects concerning programming productivity can be a source of motivation to justify user's choices and requirements. For instance, there is evidence from the empirical data analyzed showing that many users prefer to be notified in case of error conditions or inconsistencies that may lead to unexpected API behaviors. This can be considered a productivity matter, as programmers would like to be informed of any potential problems that may occur later in the software development cycle. A choice of a lenient API behavior can be harmful to productivity, as it may 'hide' unexpected conditions that can cause further side effects. This subject will be further discussed in section 5.3. Another productivity aspect to be considered by API designers includes common scenarios that enable a 'quick start' for most use cases. This can help users in their attempt to make effective use of basic API features. Programmer's personas can also be a factor of influence on this topic, since "opportunistic programmers" (92) show a clear preference for quick and simple solutions to solve their problem, even if they don't fully understand it at first.

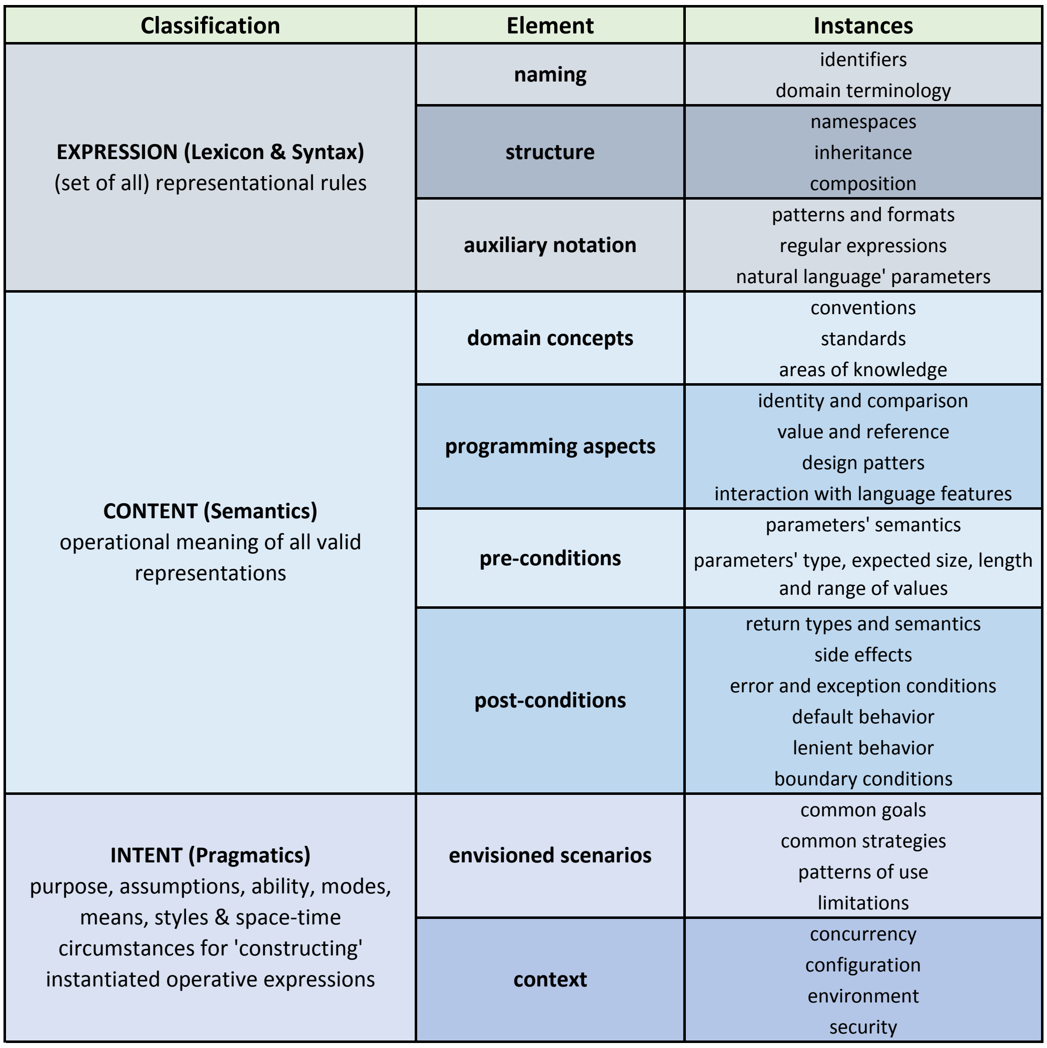
Metacommunication elements

The API metacommunication template organizes the space of the designer's reflection about the artifact that is to be constructed: the audience profile, the problems it is supposed to help solving, the strategies it supports, and so on. An API, in its role as the designer's deputy at interaction time, should effectively communicate its underlying design vision. In order to accomplish this objective, designers can use a set of elements that support this communication process. Among these elements, we have programming language features that allow API specification and implementation, and natural language descriptions (documentation). This subsection provides an overall description of elements that enable API metacommunication, organized by dimensions of human communication which they are mainly related to. These dimensions are *expression*, *content* and *intent*.

Expression refers to the code forms from signification systems used in communication to achieve its goals, i.e. sender's intent. From a linguistic perspective, it is usually associated with the lexical and syntactic aspects of language. Considering communication through software artifacts, designers' expression corresponds to the use of formal language's vocabulary and syntax. Written documentation is also a form of expression in API metacommunication, which extends the formal language with richer vocabulary and syntax from natural language, with less rigor. It provides more expressiveness for designers to convey their message to users, which is in line with our 'holistic' perspective of API metacommunication explained in 3.1.11. In the classification of API metacommunication elements, we take expression to mean the *representational rules* specified by the underlying programming language, i.e. the set of all rules that affect syntactical and lexical choices available to programmers.

Content refers to the actual meaning carried by the sender's expression. In general, it corresponds to a language's semantic aspects. In natural language communication, different forms of expression may convey the same content, which is not usual in the case of formal languages. In the context of API metacommunication elements, content defines the *operational meaning* of all valid representations (i.e. expression).

Intent deals with sender's goals in communication, and is usually associated with pragmatic aspects of language. Semiotic Engineering handles intent by means of speech act theory, which distinguishes sender's objectives (illocutionary act) from the actual effect on receiver (perlocutionary act).



Classification	Element	Instances
EXPRESSION (Lexicon & Syntax) (set of all) representational rules	**naming**	identifiers domain terminology
	structure	namespaces inheritance composition
	auxiliary notation	patterns and formats regular expressions natural language' parameters
CONTENT (Semantics) operational meaning of all valid representations	**domain concepts**	conventions standards areas of knowledge
	programming aspects	identity and comparison value and reference design patters interaction with language features
	pre-conditions	parameters' semantics parameters' type, expected size, length and range of values
	post-conditions	return types and semantics side effects error and exception conditions default behavior lenient behavior boundary conditions
INTENT (Pragmatics) purpose, assumptions, ability, modes, means, styles & space-time circumstances for 'constructing' instantiated operative expressions	**envisioned scenarios**	common goals common strategies patterns of use limitations
	context	concurrency configuration environment security

Figure 5.3: API metacommunication elements

Effective communication occurs when perlocution matches sender's illocution. With respect to API design, intent elements convey the pragmatic aspects of API use: purpose, assumptions, ability, modes, means, styles & space-time circumstances for 'constructing' instantiated operative expressions.

Figure 5.3 illustrates the classification of API metacommunication elements and provides some instances for each type. These elements can be combined to handle API metacommunication, working as 'building blocks' that help the encoding of the template parts in a software artifact.

A brief description of these elements follows.

– naming: choice of the various names that compose an API, like parameters, functions, classes, and so on, using vocabulary that includes programming and domain concepts;

– structure: language features that allow structural organization of an API, like name spaces, inheritance and composition.

– auxiliary notation: secondary representation systems that extend the primary language, like patterns for regular expressions and date parsing or formatting. There are also APIs that use a ‘natural language’ style to specify certain operations like, for instance, relative periods of time.

– domain concepts: APIs usually solve problems of various domains, so there is a need to make clear which concepts from the related domain(s) are used, and in which ways.

– programming aspects: some concepts from programming languages are critically important for proper language use, and yet, they are still source of misunderstandings from programmers, also affecting the use of APIs. For instance, issues concerning object identity and comparison, values vs. references, patterns of use and mutability are some examples of elements that may affect the effectiveness of API design communication.

– pre-conditions: conditions that should be met by users before calling an API operation, like parameter types and accepted range of values, and the meaning of these values. These are user ‘obligations’ in the semantic contract of an API.

– post-conditions: specify API commitments in the semantic contract, i.e. results obtained by users after calling API operations (user ‘rights’). These results usually depend on the evaluation of pre-conditions, as violating them usually result in errors. Designers specify post-conditions to communicate clearly what should be expected in normal operating conditions, and what should happen in case of error. Furthermore, default and lenient behavior are relevant aspects of this metacommunication element that are frequently omitted, frequently appearing as cause of trouble.

– envisioned scenarios: every API is designed to support a set of use scenarios, which may be explicitly defined or not. These scenarios present which goals an API is supposed to achieve, and what strategies can support these goals. This is a metacommunication element of primary relevance, as it provides the pragmatic conditions under which the software artifact is intended to offer its services effectively. Thoughtful scenarios may help programmers identify and understand an API’s common situated uses, providing appropriate patterns that can be adapted to achieve a range of purposes. Scenarios can also be a useful tool to illustrate API’s limitations and corner cases.

– context: API operations may have different meanings and behaviors when operating under specific contexts. For instance, concurrent operations can cause non-deterministic API behavior. Also, configuration, environment and security issues may also affect API operations. These are important aspects to be communicated to API users which usually cannot be fully expressed at the semantic level of programming languages.

By analyzing the description of each element classification shown in figure 5.3, we can see that the definition of the intent dimension is more complex when compared to the definitions of the expression and content dimensions. This may be explained by the fact that, at the pragmatic level, designers have the chance to communicate patterns, conditions and contextual information that language's semantic aspects usually cannot express in isolation. This perspective shows us that, from a communication standpoint, much is left behind if we only specify the lexical, syntactical and semantic aspects of APIs. The findings from the qualitative analysis of empirical data also supports this conclusion, as discussed further in this chapter.

Quality attributes for elements. Metacommunication elements provide an organized view of the resources available to compose the designer's message to users. In addition to identifying *what* makes these elements of discourse, we should also think in terms of their quality attributes to characterize their communicability. Concepts that apply to communication in general can also be used to qualify these elements, like consistency, redundancy, ambiguity and completeness. Depending on the context, these attributes may indicate either positive or negative characteristics of an element. For instance, redundancy is frequently used in communication to highlight certain aspects of a message which may affect positively its effectiveness, whereas in formal languages it is usually considered bad practice.

The cooperative principle, defined by Grice's four maxims (see chapter 3), provides theoretical guidance to the formulation of these attributes. The following list presents the proposed attributes and their description, as well as their relation to the maxims, when applicable.

- completeness: corresponds to the maxim of quantity, which states that "participants in a conversation should make their contribution as informative as necessary; not more not less" (22).
- conciseness: it is the dual of the previous attribute, and refers to the "not more than necessary" part of the maxim of quantity.
- ambiguity: derives from the maxim of manner, and affects negatively the quality of communication.

- correctness: the maxim of quality states that participants in a conversation should not "tell a lie". In the context of API design, this maps to including incorrect information in metacommunication elements.
- relevance: the maxim of relation recommends that participants observe the relevance of their communication, and it also applies to the context of API metacommunication.
- consistency: similar intent and content elements should be expressed using similar code forms. This can be used as a communication strategy to help reduce cognitive impact on users, as described in the cognitive dimensions framework by a dimension named the same way. This dimension is further discussed in section 5.2.2.
- grounding: the maxim of quality also states that participants should observe their knowledge and doubts in conversation. This maps to the need to make sure that participants in communication share a common ground, like domain-specific of language knowledge. Elements used to construct the designer's message should compensate for possibly inadequate grounding, guided by the metacommunication template components.
- visibility: interacting with APIs relies on textual languages, both formal and natural. This means that the structural organization of communication may contribute to (or not) the availability of relevant information. For example, API documentation should emphasize the most relevant information for its appropriate use. This may involve text formatting strategies, as well as lexical, syntactic and semantic choices to achieve better visibility for critical information.
- use of metaphors: choosing appropriate identifiers in API design is critical for signifying its intended use. However, the trade-off between expressiveness and conciseness sometimes make it a difficult decision. Metaphors are a powerful resource to extend the potential meanings extracted from a piece of communication, relying on its pragmatic aspects. Therefore, good metaphors can contribute to better understanding of API's features. On the other hand, bad choices may hinder user's interpretation of its meanings.
- use of metonymies: similarly to metaphors, metonymies are frequently explored in the construction of software artifacts, extending the expressiveness of language by exploring associations between things or concepts (containment, contiguity, product/process, and so on). Just like metaphors, using metonymies can contribute (or not) to improve communic-

ability, depending on factors that influence its interpretation like culture, contextual information or programming experience.

– redundancy: redundant messages provide special emphasis on communication aspects that deserve special attention, especially between different contexts of use. When it comes to API documentation, for example, inserting critical information in different but strategic places may prevent users from bypassing critical details. In the context of formal representation of APIs, however, redundant use of language elements may affect negatively its interpretation.

In spite of being orthogonal to metacommunication elements, not all quality attributes may be adequate to characterize some of these elements. For instance, consistency is probably closer to the characterization of expression elements than content or intent elements. Other attributes like completeness, correctness and relevance can properly qualify any element. In the case of metaphors and metonymies, one can think in terms of proper (or improper) use of metaphors and metonymies, which apply mostly to expression elements (naming). The discussion of findings in section 5.3 illustrates some examples of how these attributes may be used to describe the quality of metacommunication elements.

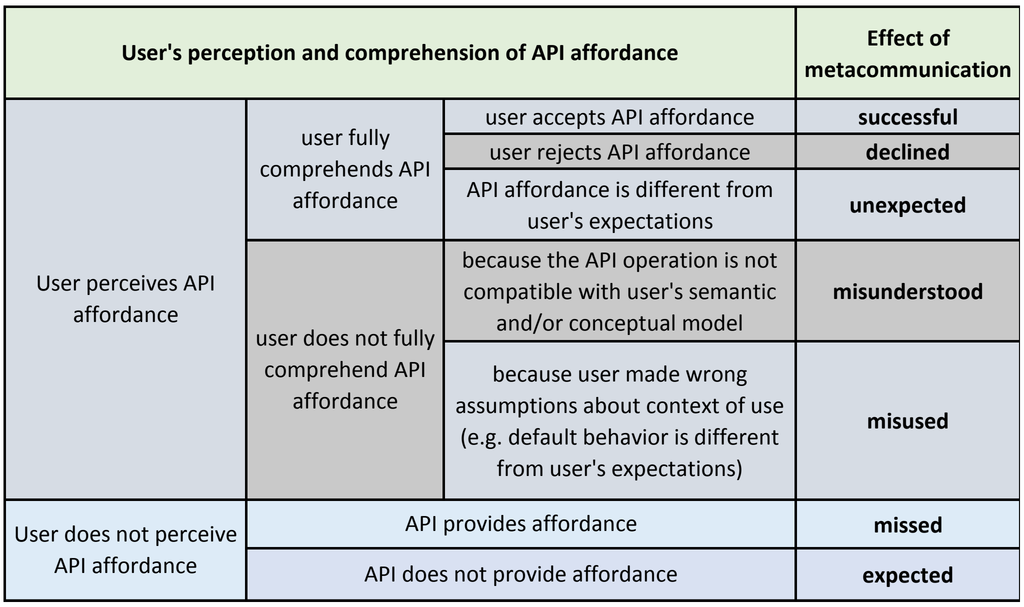
5.2.2 Effects

The second dimension in the proposed communicative perspective of API design accounts for possible effects of API metacommunication on users. As previously mentioned, this classification is based on the distinction between the illocutionary act (intent) and the perlocutionary act (effect), proposed by speech act theory. This subsection introduces a classification of metacommunication effects on users in the context of API use. It also discusses the application of the cognitive dimensions of notations framework to characterize the communication effects in terms of cognitive impact on API users.

Metacommunication effects

The effects of API metacommunication on users encompasses the different ways in which users may perceive and understand the software artifact's available features and operations. We refer to the generally available features and operations of APIs as *affordances*, a term borrowed from Psychology (93) and introduced in HCI by Norman (94) to describe the possible actions that users may perceive when interacting with an artifact. This term is commonly used to distinguish between *real* and *perceived* affordances, making clear the separation between available actions and their actual interpretation by users.

In order to systematically analyze metacommunication effects, we provide a taxonomy for these effects that considers users' perception and understanding of API affordances and their meaning, as well as their actual existence. Figure 5.4 illustrates the proposed classification of metacommunication effects.



User's perception and comprehension of API affordance			Effect of metacommunication
User perceives API affordance	user fully comprehends API affordance	user accepts API affordance	**successful**
		user rejects API affordance	**declined**
		API affordance is different from user's expectations	**unexpected**
	user does not fully comprehend API affordance	because the API operation is not compatible with user's semantic and/or conceptual model	**misunderstood**
		because user made wrong assumptions about context of use (e.g. default behavior is different from user's expectations)	**misused**
User does not perceive API affordance	API provides affordance		**missed**
	API does not provide affordance		**expected**

Figure 5.4: API metacommunication effects

The first level classifies the effect with respect to users' perception of an affordance. When trying to use an API, programmers look for features that help them achieve a certain goal. Depending on metacommunication effectiveness, users may face different levels of difficulty to realize that API's features correspond (or not) to what they are looking for.

Once identifying the existing affordance, a user tries to understand the details of its operation and use it. Depending on his interpretation of API's features, different outcomes may occur. If the user understands and accepts the design of the API affordance, the effect of metacommunication is *successful*. Alternatively, if user fully understands but rejects the chosen design, the effect can be classified as *declined*. Lastly, if a user understands an affordance, but finds out that it works differently from his initial assumptions, this type of effect can be classified as *unexpected*.

Still in the context where a user identifies the existence of an affordance, it may be the case that she does not fully grasp its goals or details of operation. This may happen when user's conceptual or semantic model of the API's operation does not correspond to its actual implementation. It means that the feature's overall goal was understood, but not all of its details. In this case, we refer to this type of metacommunication effect as *misunderstood*. Additionally, there is a subtle variation of the misunderstood effect, when user actually comprehends the concepts and semantics of API operations, but makes wrong assumptions about implicit aspects of its behavior in specific contexts. For instance, user expects a default behavior from the API that does not correspond to the actual implementation. This type of effect can be regarded as a 'partial' misunderstanding of an API affordance, and we refer to it as *misused*. When compared to the effects that characterize API misunderstanding (misunderstood and misused), the *unexpected* effect can be characterized as a temporary breakdown in user's comprehension, which means that a user expects an affordance to work in ways that she quickly discovers that are different from the actual behavior.

When users do not perceive an affordance as available, we divide the effect taxonomy between *missed* and *expected*. The 'missed' effect describes the scenario where the affordance is actually available, but the user does not realize its existence. Its counterpart is the 'expected' effect, which occurs when the user looks for features that the API does not provide.

In the proposed classification, the effect named 'successful' refers to the absence of problems in metacommunication, which is the main objective of API design. Therefore, the remaining effects are of prime interest in the context of this research, as they indicate potential communicability issues.

It should be noted that, in spite of being a possible effect according to the criteria used in the proposed taxonomy, 'unexpected' was not observed in the analysis of empirical data. The nature of the collected empirical data explains the absence of this effect in its analysis and classification. The 'unexpected' effect characterizes a situation where a user perceives an API affordance but temporarily interprets it in a different way. However, in this case the user rapidly restores his interpretation of the affordance, and understands that it is different from her expectations. In the classification of communication failures, this is close to the 'oops' tag, where user immediately realizes and recovers from an error situation, as discussed in subsection 5.2.3.

In addition to the characterization of users' perception and understanding of API affordances, we may also describe the effect dimension in terms of the cognitive aspects that possibly affect users in the process of receiving and unfolding the designer's message encoded in API's artifacts.

Cognitive characterization of effects

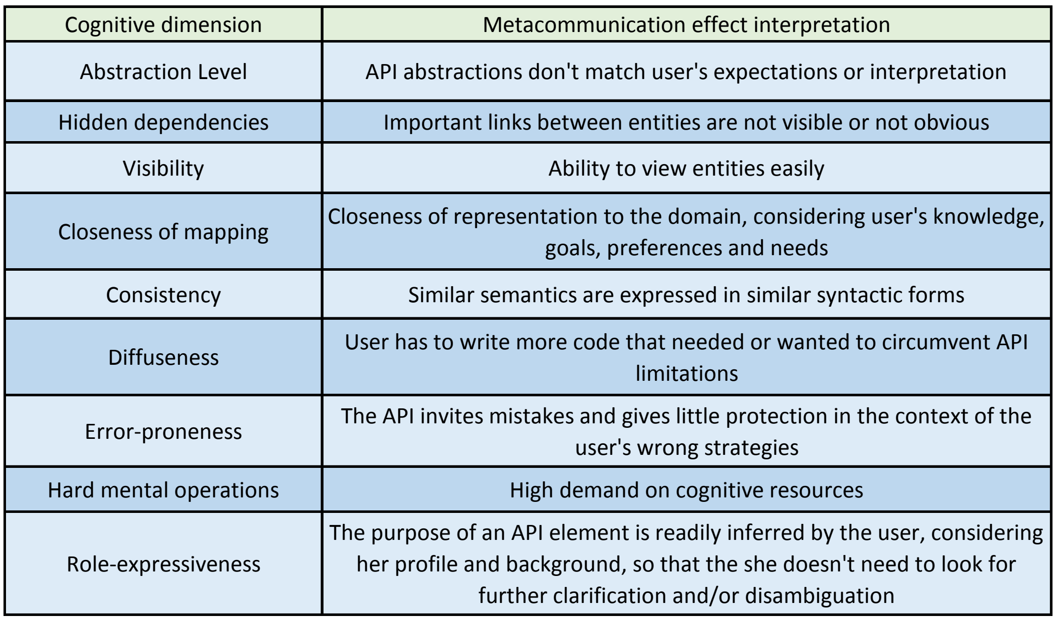
As previously mentioned in chapter 3, the Cognitive Dimensions of Notations framework has been used as part of the methodological tools to analyze the empirical data collected from bug reports. Over the course of the research, the CDs have been used as a vocabulary to describe the cognitive aspects explicitly or potentially associated with the communicability issues identified in the bug reports. Also, the CDs promote an organization of the cognitive space to be analyzed when dealing with notations like programming languages and APIs, much like this research intends to contribute to the organization of API design space from a communication perspective.

Despite the adoption of a designer-programmer metacommunication perspective, this research work uses the CDs to characterize the cognitive impact on programmers only. This is an important distinction to make from the alternative use of CD's, which would describe designers' own cognitive issues when dealing with a programming language to construct APIs. During the research iterations, the role of the CDs converged to the description of how API metacommunication issues may affect programmers cognitively, in association with the possible communicative effects. The analysis of cognitive aspects from the perspective of API designers should probably offer novel and interesting insights, but is beyond the scope of the current work.

Traditionally, CDs have been used to evaluate the usability of notations in scenarios that define a certain task to be accomplished. However, these evaluations usually take notations in isolation. One of the envisioned advantages of using the CDs inside a communicability perspective is that, together, they allow us to analyze a notation in a more completely specified context of use (because it includes explicit intentionality and communication aspects). The metacommunication analysis frames the API notation to which we apply the CDs. This provides a richer situated view of the cognitive aspects involved, as opposed to a generic unsituated analysis of the notation by itself.

In the process of refining the use of CDs to describe cognitive effects of metacommunication, the initial set of dimensions was reduced and reinterpreted to a more specific instantiation of some dimensions. Differently from the adaptation of the dimensions proposed in API usability studies carried out at Microsoft (14), the original names of the dimensions have been kept.

Some of the dimensions were not selected from the original set, not because they don't apply to the evaluation of APIs, but for being closer to the characterization of how users interact with an API to accomplish a series of goals (user-system communication). As we are dealing mostly with the effects of designer-user metacommunication, the selected set of dimensions are better suited to describe the interpretation of API's affordances. For this reason, dimensions like 'viscosity' or 'premature commitment', for example, do not show up in the final list of dimensions, as they are not suited to describe the cognitive impact of metacommunication effects on users. Figure 5.5 illustrates the selected set of dimensions to characterize cognitive aspects associated with API metacommunication effects.



Cognitive dimension	Metacommunication effect interpretation
Abstraction Level	API abstractions don't match user's expectations or interpretation
Hidden dependencies	Important links between entities are not visible or not obvious
Visibility	Ability to view entities easily
Closeness of mapping	Closeness of representation to the domain, considering user's knowledge, goals, preferences and needs
Consistency	Similar semantics are expressed in similar syntactic forms
Diffuseness	User has to write more code that needed or wanted to circumvent API limitations
Error-proneness	The API invites mistakes and gives little protection in the context of the user's wrong strategies
Hard mental operations	High demand on cognitive resources
Role-expressiveness	The purpose of an API element is readily inferred by the user, considering her profile and background, so that the she doesn't need to look for further clarification and/or disambiguation

Figure 5.5: Cognitive dimensions to characterize API metacommunication effects

The application of the CDs to enrich the description of metacommunication effects with cognitive aspects will be further illustrated in section 5.3.

The characterization of API metacommunication effects in terms of users' perception, understanding and cognitive impact enables designers' awareness about the outcomes that may result from the inherent trade-offs of design activities. Next subsection introduces the classification of failures that help in the diagnostic of communicative issues that lead to these effects.

5.2.3 Failures

The third dimension used to structure the communicative space of API design provides a diagnostic of the mismatches between designer's intent (illocutionary act) and its effect on users (perlocutionary act). These mismatches can be characterized in terms of the classification of communicative failures proposed by Semiotic Engineering, presented in chapter 3. In this chapter, we introduce an adapted classification of failures, with minor changes and reinterpretations of their original meaning in order to better suit the context of API metacommunication. There are three main classes of failures as listed below, extracted from (22):

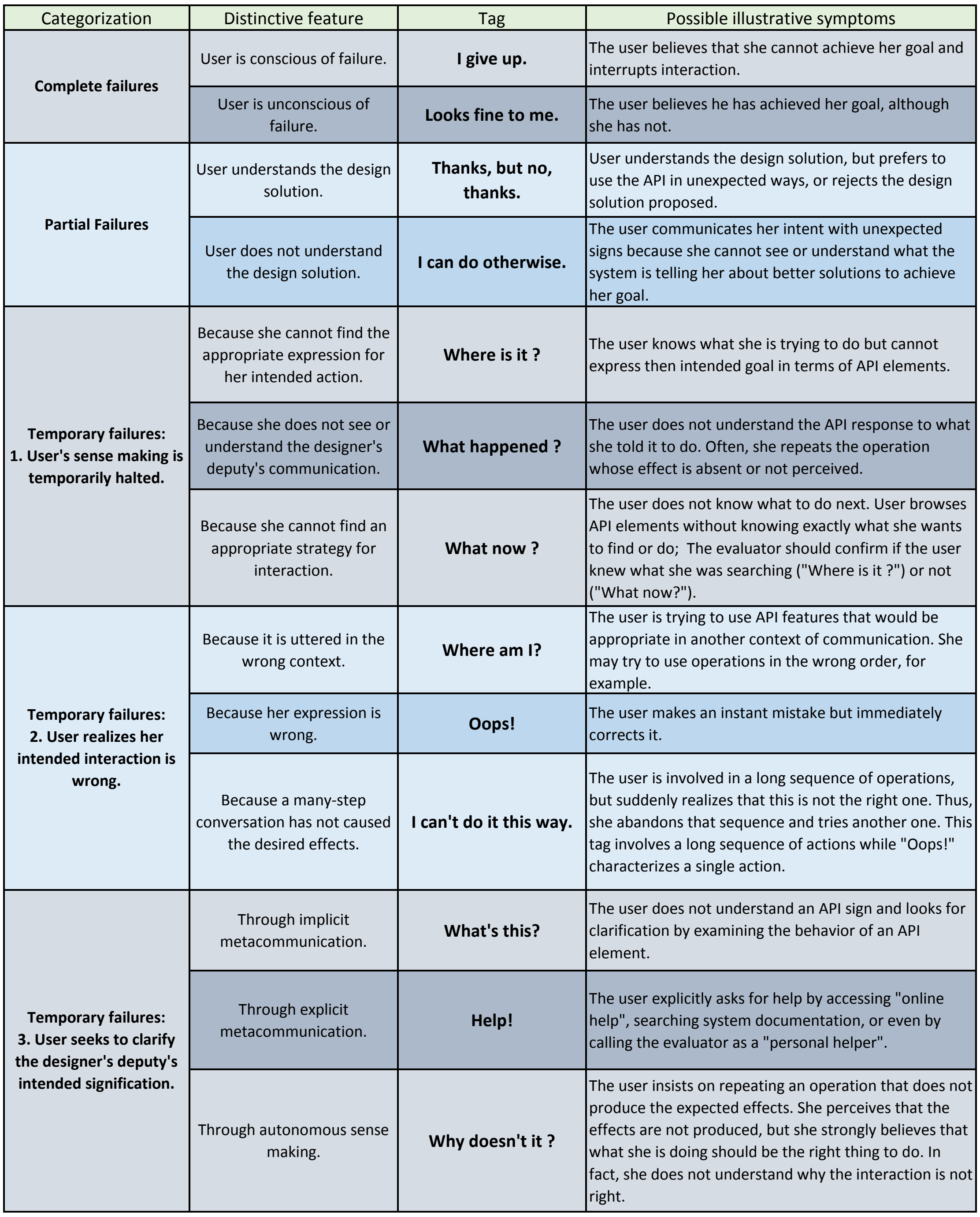
- Complete failures: occur when global illocution is not consistent with global perlocution (strategic failure).
- Temporary failures: occur when global illocution is consistent with global perlocution, but local illocution is not consistent with local perlocution (operational failure).
- Partial failures: if local illocution is consistent with local perlocution, the failure is categorized as partial with:
  - potential residual problems for the user because she does not understand the designer's deputy's illocution, and this somehow fails to do exactly what is expected.
  - no residual problems for the user because she fully understands the designer's deputy illocution, but somehow fails to do exactly what is expected.

The top level classification of failures can be further analyzed at a finer granularity level by using a set of *tags* that characterize communicative breakdowns from the perspective of the receiver. These tags are part of the Communicability Evaluation Method (CEM) (26), and their main goal is to 'put words into user's mouth' when applied to HCI studies involving participants. In the context of this research, these tags describe the breakdowns that programmers may experience as receivers of API designer's metacommunication. Figure 5.6 corresponds to an adaptation of a table found in page 43 from (26), and shows minor changes in the 'illustrative symptoms' to provide examples of possible tag application in the context of API communicability evaluation.

Almost all tags represent utterances that users may explicitly or implicitly emit while experiencing failures in the interaction with a system during an evaluation session. The only exception is the 'looks fine to me' tag, since only the evaluator is in the epistemic position to identify a situation where the user has not accomplished her goal, despite the fact that she believes to have suceeded.

The reinterpretation of some tags in the context of the analysis of bug reports can be summarized as follows:

- 'I give up': this tag applies to the reports where user is conscious of failure in achieving her goal with the API. However, the fact that the user filed a bug report means that she thinks that the API should provide some



Categorization	Distinctive feature	Tag	Possible illustrative symptoms
Complete failures	User is conscious of failure.	**I give up.**	The user believes that she cannot achieve her goal and interrupts interaction.
	User is unconscious of failure.	**Looks fine to me.**	The user believes he has achieved her goal, although she has not.
Partial Failures	User understands the design solution.	**Thanks, but no, thanks.**	User understands the design solution, but prefers to use the API in unexpected ways, or rejects the design solution proposed.
	User does not understand the design solution.	**I can do otherwise.**	The user communicates her intent with unexpected signs because she cannot see or understand what the system is telling her about better solutions to achieve her goal.
Temporary failures: 1. User's sense making is temporarily halted.	Because she cannot find the appropriate expression for her intended action.	**Where is it ?**	The user knows what she is trying to do but cannot express then intended goal in terms of API elements.
	Because she does not see or understand the designer's deputy's communication.	**What happened ?**	The user does not understand the API response to what she told it to do. Often, she repeats the operation whose effect is absent or not perceived.
	Because she cannot find an appropriate strategy for interaction.	**What now ?**	The user does not know what to do next. User browses API elements without knowing exactly what she wants to find or do; The evaluator should confirm if the user knew what she was searching ("Where is it ?") or not ("What now?").
Temporary failures: 2. User realizes her intended interaction is wrong.	Because it is uttered in the wrong context.	**Where am I?**	The user is trying to use API features that would be appropriate in another context of communication. She may try to use operations in the wrong order, for example.
	Because her expression is wrong.	**Oops!**	The user makes an instant mistake but immediately corrects it.
	Because a many-step conversation has not caused the desired effects.	**I can't do it this way.**	The user is involved in a long sequence of operations, but suddenly realizes that this is not the right one. Thus, she abandons that sequence and tries another one. This tag involves a long sequence of actions while "Oops!" characterizes a single action.
Temporary failures: 3. User seeks to clarify the designer's deputy's intended signification.	Through implicit metacommunication.	**What's this?**	The user does not understand an API sign and looks for clarification by examining the behavior of an API element.
	Through explicit metacommunication.	**Help!**	The user explicitly asks for help by accessing "online help", searching system documentation, or even by calling the evaluator as a "personal helper".
	Through autonomous sense making.	**Why doesn't it ?**	The user insists on repeating an operation that does not produce the expected effects. She perceives that the effects are not produced, but she strongly believes that what she is doing should be the right thing to do. In fact, she does not understand why the interaction is not right.

Figure 5.6: API metacommunication failures

affordance that is absent or works differently from expected. Therefore, this tag applies more frequently to situations where the effect on users are classified as ‘missed’ or ‘misunderstood’.

- ‘Looks fine to me’: when user adds examples to the bug report, it usually represents a piece of source code that illustrates the actual user intepretation of API’s affordances. Frequently, this code means that the user believes she is doing the right thing, which is not true in many cases. This tag is of great relevance, since it represents a global failure (user could not achieve goal), although she thinks that the code *should have accomplished the intended goal.* Therefore, there is a subtle distinction in the reinterpretation of this tag when compared to its original sense, since the user knows that there is actually a problem in the interaction with the API (otherwise, there would be no reason to file a report in the first place). The main difference rests in user’s partial awareness of failure: in the bug report context, user thinks the goal was not achieved because there is a problem in the API, since in user’s view the code sample ‘looks fine to me’.
- ‘Thanks, but no, thanks’: user explicitly rejects some characteristic of the artifact’s design. Mostly, this tag is associated with the ‘declined’ effect, but sometimes users reject the design without perceiving the API’s affordance or comprehending its details. In this case, it may correspond to effects ‘missed’ or ‘misunderstood’, respectively.
- ‘I can do otherwise’: bug report systems usually request from users that they inform possible ‘workarounds’ to the identified bug. This, from a communicative failure perspective, can be interpreted as a programmer using alternative or unexpected ways of achieving a certain goal. Usually, this way of doing things is wrong or suboptimal.
- ‘What happened’: when programmers write and execute code that uses some API feature, this tag describes the breakdown that may occur when users face difficulties to interpret its results. It should be noted that the absence of signs is, by itself, a ‘sign’ that can also be associated with this tag.
- ‘Why doesn’t it?’: this tag characterizes a temporary failure that occurs during user’s attempt to test the API feature in discussion by writing sample code. As the code does not work as expected, the user tries to execute it a number of times, possibly with minor changes, in order to understand why it does show the expected behavior. This temporary failure may precede a complete or partial failure, as it represents user’s

attempt to interpret the artifact's behavior and restore productive communication (which sometimes fail to occur).

Due to the nature of empirical data analyzed, some of the communicability tags described in 5.6 could not be observed. Namely, tags 'what now?', 'where am I?', 'oops!', 'I can't do it this way' and 'what's this'. This can be explained by the fact that these tags depend on live observation of user's interaction with the system, since they have a dynamic nature that cannot be inferred from the analysis of bug reports. However, it does not mean that they don't apply to the context of API evaluation, and for this reason they have been kept in the list of tags presented here. Some of these tags also had their meaning reinterpreted to adapt to the API context.

The analysis of failures by tagging user communicative breakdowns was one of the first steps of the method used in the research. The objective was to characterize these failures and try to identify aspects of designer's metacommunication that might have influenced user's perception and interpretation of API affordances. The nature of collected data (bug reports) allowed the analysis of users' discourse which, in many cases, offered clear evidence of the failures involved. Gradually, the analysis of data conducted to the reinterpretation of communicability tags as described above.

It should be noted that most bug reports in the selected dataset can be associated with two kinds of situations. The first occurs when a user fails to accomplish an intended goal with the API, but thinks that her code is right and the problem is on the API's side (user does not fully understand the nature of the problem). The second situation happens when a user understands what is happening during interaction with API, but thinks that its behavior is wrong or inadequate (user rejects the design). In the first case, users are not conscious that they are not using the API correctly, which can be described by the tag 'looks fine to me' (a complete failure). In the second situation, user consciously rejects the API feature's design, which is a mismatch between designer's intent and user's expectations or needs. This is the case for tag 'thanks, but no, thanks'.

Despite the finer granularity of the low-level tags, they are mainly oriented to the diagnostic of failures users experience as receivers of the metacommunication message. However, the top level classification of failures is also of great interest, as it represents the more abstract nature of the breakdown (complete, temporary or partial). Complete failures are the most severe because they mean that users could not achieve their goal. Partial failures, although undesirable, imply that users are not satisfied with API's affordances, or use them in unexpected ways. Temporary failures, as they name suggest, are only temporary, sometimes with minor consequences. However, they are frequently the ‘path’ to a major failure, either complete or partial.

The remaining of this chapter discusses qualitative findings, and the concepts introduced in this section will be applied in the analysis of these findings.

5.3 Qualitative findings

This section presents qualitative findings derived from research work described in chapter 4. These findings are the result of observations made in the analysis of empirical data, based on the communicative approach summarized in section 5.2.

The following subsections describe these findings, organized in major categories of topics which resulted from the qualitative analysis of data carried out in the research studies. This does not mean that these findings should be taken in isolation, since they are frequently interrelated. Whenever possible, intersections are mentioned in the text to make them more explicit. In addition, the implications of these findings are discussed with respect to the research goals.

5.3.1 Narrow protocol and default behavior

Frequently, we find APIs that implement an extensive set of features, which map to a wide variety of possible behaviors. These differences in behavior are usually controlled by methods or functions that work as ‘configuration switches’, setting an entity’s internal state that determines its functionalities.

However, there are cases in which the interface that allows users to specify API behavior is ‘narrow’ when compared to the extensive set of different configurations it maps to. In these cases, a common strategy is to provide a number of default settings in the API to allow its use without having to specify all the parameters that rule its behavior. This is equivalent to having a default value for each parameter or switch that was not specified by programmers. This approach is an attempt to avoid affecting API usability by forcing a user to specify a large number of options or settings every time she calls an API operation.

When designers resort to implementing many default aspects in API behavior, there are pragmatic consequences to API metacommunication that may affect users’ interpretation of its meanings. Sometimes, it is the ‘unspoken’ or less visible aspects of its behavior that mislead users to interpret API metacommunication in ways other than intended ones. Users engage in abductive reasoning to formulate plausible hypothesized rules, according to their previous experience, knowledge and cultural influences.

Default behavior may be specified in terms of semantic elements of API metacommunication, mostly related to operations' post-conditions. Depending on the decisions concerning default values, they may affect the pragmatic aspects of API use, since there can be consequences to the envisioned scenarios of use. The metacommunication template can guide designers in the evaluation of users' knowledge, preferences and needs, so that the choices of default values may contribute to the proper formulation of users' hypotheses about API behavior.

Ambiguous or less visible API default behavior may result in users misunderstanding it, or even declining to its design approach. Users may also be surprised by unexpected API behavior, and accept the proposed design instead of declining it.

From a cognitive view, designers usually take the 'closeness of mapping' dimension into consideration when choosing default values, even not being really aware of it. What is to be noted is that communication processes involves at least two human minds, and what is close to designer's view of the domain may not be the same as the user's. Therefore, as discussed earlier, we take the user's view of the closeness of mapping dimension to characterize the cognitive effects of API metacommunication.

The definition of default values may follow a variety of criteria, and this discussion is based on their usability: sparing users from defining a number of parameters facilitates their work. By making an analogy with Grice's Cooperative Principle (see 3.1.5), a default value can be defined by following the maxims of quality (it is usually *true* for the user) and quantity (it can be omitted, since mentioning it would be unnecessarily redundant). This criterion is based on the pragmatic dimension of language, as it takes into account the context of use and sign production in the metacommunication process involving API designer and user.

In addition to usability, default values may also be defined with respect to 'algorithmic efficiency' criteria. In contrast, this type of criterion is completely dissociated from communicative or cognitive aspects, since it focuses on computational performance. Every variable should be initialized with a value, thus it may be interesting, from a performance standpoint, to use a specific default value that does not necessarily match usability criteria. This potential conflict may lead to decisions based on a trade-off that occurs when designers have to choose which 'interlocutor' to serve with priority: the user or the computer. This topic is further discussed in chapter 6.

Another observed consequence of having a narrow protocol to inform complex API behavior is the adoption of a lenient approach, in which the API tolerates some inconsistent or erroneous condition by automatically 'correcting' the artifact's state. This subject is further discussed in the next subsection.

The Java bug report 8017133[3] presents a curious example of the current topic. The Java *SimpleDateFormat* class enables parsing date and time information from a wide variety of formats, specified by a character mask parameter. When users do not specify all the date and time components for parsing, the API is forced to adopt default values for the non-specified parts. In this bug report, a user tries to parse the date '02/29 08:15' using the format 'MM/dd HH:mm' (i.e. 'month/day hour:minute'). As the year is not specified, a default value is adopted in the parsing process.

In his abductive reasoning, the user inferred that the default year, when not specified, is the current year. However, Java adopts the Unix tradition of representing dates as the number of seconds since 1970-01-01 00:00:00 GMT. Therefore, the default value used for an 'empty' year value is 1970.

The user complains in the bug report about the unexpected API behavior, because it raises an exception to indicate parsing error. From the user's perspective, this should be a legal date. The interesting detail about this report is that the exception only occurred because the date to be parsed was Feb 29th, which is valid only in leap years. The report was created in 2012, a leap year, which made the user think that it should be parsed into a valid date (29 Feb 2012). If the user tried to parse a more ordinary date (for instance, Feb 28th), no exception would have been raised, and the date would be 'silently' parsed as '28 Feb 1970 08:15:00'. This is also an example of lenient behavior associated with the need for default values (see subsection 5.3.2).

Java classes that deal with date and time representation and operations are notorious sources of trouble to many programmers, as evidenced by collected data from bug reports. Articles from programming forums like Stack Overflow, for example, also illustrate the difficulties users face when dealing with date and time in Java. A question about string to date conversion in Java[4] has been viewed more than 530k times[5].

Concerning default values for date and time parsing, an interesting approach has been adopted by an alternative implementation of date and time classes for Java, a library called 'Joda Time'[6]. Due to the variety of problems

[3]`https://bugs.openjdk.java.net/browse/JDK-8017133`
[4]`http://stackoverflow.com/questions/4216745`
[5]All links to web sites in this chapter were visited as of Feb. 2015
[6]`http://www.joda.org/joda-time/`

associated with Java's native date classes, this library gradually became a popular substitute for Java's original API. Joda Time's class for date format specification allows users to query and change the default year used in parsing. Besides allowing the customization of default values, this kind of approach turns the default value a first-class concept in the API, and not an 'almost missed' aspect of its behavior.

It should be noted that the adoption of default values to allow 'underspecification' of API behavior is not, by itself, a problem. However, it is important that designers be aware of the pragmatic consequences of using and choosing proper default values and behavior in situations like the one described in the example above. It is the nature of epistemic tools like the ones proposed in section 5.2 to raise designers' awareness about the trade-offs and possible consequences of certain design decisions, allowing them to develop a deeper understanding of the problem at hand and make choices based on a more informed reasoning.

5.3.2
Lenient behavior

As noted in the previous subsection, a possible consequence of having a narrow protocol to interact with API's features is the adoption of a lenient behavior. This means that the API is more tolerant with respect to inconsistent or erroneous situations, by making adjustments as needed to restore its state.

In order to restore the artifact's consistency, designers adopt some policy to circumvent error states and 'correct' improper use of API's services. The main consequence of this approach is that, by not telling users about the erroneous or inconsistent state, the fact that something has been changed or 'silently corrected' may be the source of user's misunderstandings about the post-conditions of API's operations. This is a recurrent reason for users' complaint in bug reports.

Once again, date APIs are a common source of communicability issues associated with lenient behavior. Both Java and PHP libraries show lenient behavior that is a frequent cause of misunderstandings, misuse or declining effects on programmers. Java *Date* class even allows turning the lenient mode off, but as the default mode is 'on', this is also an often missed affordance in the interface.

PHP bug 66201[7] is an interesting evidence of this issue, with an eloquent user statement about his expectations when trying to create a *DateTime* object from an invalid date:

[7]`https://bugs.php.net/bug.php?id=66201`

"*Actual result: Date is ok. (converted to 2015-03-25 10:57:26 which is weird but ok. can swallow that although it doesn't make any sense to calculate a "real" date from 25.27.2013 input to begin with).*"

PHP bug 54524[8] provides more evidence of how leniency influences API metacommunication, this time related to the conversion of string to integer values (function *intval()*). This function takes a string as the single parameter, and returns an integer corresponding to its numerical conversion. However, when the string representation contains a number that is greater than PHP's maximum integer value, the function returns this maximum value. User complains about not being notified of the error condition:

"*In my opinion, throwing a Warning would be more intelligent than returning 2 ˆ 31-1. It is an Error because it offends Mathematics. A function with different arguments, in this case should return different results, BUT UP TO NOW, it may return always 2ˆ31-1. Can't anyone else see this ? The way it is today turns PHP into a 'hiden bugs' language.*"

A question in Stack Overflow also illustrates the difficulties associated with the use of the intval function: "PHP: intval() equivalent for numbers >= 2147483647"[9].

Lenient API behavior's influence on metacommunication should be considered with the template's components in mind, in order to promote a reflection about its corresponding effects on users' preferences and needs. Allowing a choice between lenient or strict behavior is a viable alternative, letting users select the most adequate behavior to their context. However, either way, API behavior should be clearly stated by a careful combination of metacommunication elements to prevent users from misinterpreting API's signs.

5.3.3 Implicit or ambiguous metacommunication

When creating an API, designers make a number of decisions to signify their intent using the various metacommunication elements discussed in subsection 5.2.1. Depending on these choices, metacommunication can cause 'side effects' in user's interpretation, by carrying unintended or implicit meanings represented by expression elements.

As an example to illustrate this concept, Java provides a class named *Properties*[10] that implements a basic mechanism to store configuration items in the form of key-value strings. A closer examination of the *Properties* class shows us that it derives from the *HashTable* class. In object-oriented

[8] `https://bugs.php.net/bug.php?id=54524`
[9] `http://stackoverflow.com/questions/990406`
[10] `http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html`

programming, inheritance is defined as a 'is-a' relationship between classes, i.e., if class *B* is a subclass of class *A*, this *should* mean that every object of type 'B' can be used where an 'A' is expected.

Therefore, by specifying that *Properties* is a subclass of *HashTable*, designers adopted a strategy that privileged convenience of implementation (by inheriting features and methods). However, the implicit (and unintended) message that may be received by users is that 'every *Properties* object is also a *HashTable* object', which enables the use of class *HashTable*'s methods.

There are Java bug reports (e.g. 4176094[11], 4212280[12], 8043219[13]) classified as 'not an issue' that contain evidence illustrating the above explanation. In bug report 8043219, the evaluator's response shows the effect of this 'implicit' metacommunication:

"*Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not Strings. The setProperty method should be used instead.*"

Lastly, we may characterize this Java example as an ambiguous use of the structure metacommunication element, since API designers created a potentially ambiguous interpretation by using inheritance to specify the Properties class. This is an aspect of API design that, if overlooked, may affect negatively users' interpretation of its intended purposes.

Another example of the influence of implicit or ambiguous metacommunication on API use comes from a type of problem frequently found in PHP bugs. Some array-based functions expect single dimension arrays as parameters to perform their work. However, these functions do not work well when called with multidimensional array arguments, and the documentation is sometimes silent about this fact. Two bug reports illustrate quite well the effects of not being explicit about intended behavior. In bug 11893[14], the evaluator replies: "*array_intersect isn't supposed to handle multi-dimmed arrays. Where have you read that? Not in the manual in any case.*"

In bug 52336[15] ("You can't use multidimensional arrays in functions which support arrays"), user complains about documentation omitting this behavior:

"*The documentation for functions like str_replace says, that you can use an array as subject. Multidimensional arrays are not excluded, but if you use*

[11]`https://bugs.openjdk.java.net/browse/JDK-4176094`
[12]`https://bugs.openjdk.java.net/browse/JDK-4212280`
[13]`https://bugs.openjdk.java.net/browse/JDK-8043219`
[14]`https://bugs.php.net/bug.php?id=11893`
[15]`https://bugs.php.net/bug.php?id=52336`

them nothing happens. This bug is reproducable with ALL functions which should support the use of array. If this is wanted behaviour you should edit the documentation."

While evaluator replies:

"*This isn't a bug. The documentation says you can pass an array of needles and an array of replacements. It's just common-sense that these needles/replacements would be scalar types (not an array).*"

This evaluator's statement is an evidence of how implicit metacommunication and ambiguity may affect users: what is common sense to some people may not be obvious to others. This topic can also be analyzed from the common ground perspective, as discussed later in subsection 5.3.7.

5.3.4 Use of figurative speech: metaphors and metonymies

Figures of speech like metaphors and metonymies can be a powerful resource to enhance communicability, especially if taking into account that formal representation of APIs generally consists of a concise 'message'. Even not being aware of it, API designers frequently use this communicative resource to express API's features. However, as mentioned in subsection 5.2.1, the use of metaphors and metonymies may also affect users' interpretation negatively.

A simple example of a dubious choice of name is the Java *File* class[16]. At first, one can think that it metaphorically represents a physical file stored in the file system. However, as defined in the documentation, it is "an abstract representation of file and directory pathnames". This means that it is not possible to read or write a *File* object in Java. This 'broken metaphor' is really a metonymical representation of a file name, not the actual object. However, this abstract representation of a file path is also somewhat broken, since different representations of the same file path are considered different objects (e.g. absolute and relative representations of the same file path are not equal in the *File* class abstraction). This can be a problematic use of a metonymical representation, since references to the same conceptual object are considered different objects. The effect on users caused by this type of communication issue can be described in terms of the 'abstraction level' and 'role expressiveness' cognitive dimensions (see section 3.2).

Java bug id 4094022[17] illustrates a problem of this metonymical representation of a file path name. In this report, the user complains about the use of method *File.renameTo()*. After calling the method, the user expects the phys-

[16]`http://docs.oracle.com/javase/7/docs/api/java/io/File.html`
[17]`https://bugs.openjdk.java.net/browse/JDK-4094022`

ical file to be renamed, as well as the *File* object's internal path representation. However, only the physical file name changes. The bug report evaluator answer follows below, confirming that the metonymy does not fully hold:

"*Not a bug. An instance of the File class represents a file name, not the file that it names.*"

Another example comes from PHP bug id 67272[18]: user complains that class Pathinfo does not state failure information. The evaluator's reply says[19]:

"*This function has nothing to do with filesystem. It just parses string in order to extract path-specific elemens. Path doesn't have to be existing one, as it is treated just like a string. Maybe we should make it more clear on doc page...*"

In these bug reports, users' interpretation of API's affordances is based on a contiguity relation between the object instance and the physical object. As this metonymical relation does not exist, this ambiguous definition of the API should be resolved by changes in the metacommunication elements that compose the designer's message.

Lastly, Java bug 8014238[20] also provides evidence of a bad choice of metaphorical representation. In this report, a user complains about an error in the documentation of class *ProcessBuilder*. Documentation states that, in order to get the subprocess's standard output stream, one should call method *Process.getInputStream()*. The user suggests that the correct text would refer to method *Process.getOutputStream()*. However, the evaluator replies:

"*The Java documentation describes the world correctly. There is the difficulties with the naming of streams. The naming is right from the parent process point of view, so the the [Process.getInputStream] in the parent end of pipe that can be used for reading of child std output*".

A question in Stack Overflow ("Java Process getInputStream vs. getOutputStream")[21] about this very issue contains interesting users' comments that enriches the discussion of communicability aspects involved:

[answer] "*You are using names that make sense in the context of the spawned process. But the API names make sense in the context of the parent process.*"

[comment] "*I find it very confusing to, not to say flawed. Isn't it a core Object-Oriented design-pattern to name methods in the context of the object that offers this method?*"

[18]`https://bugs.php.net/bug.php?id=67272`
[19]original mistakes kept in text
[20]`https://bugs.openjdk.java.net/browse/JDK-8014238`
[21]`http://stackoverflow.com/questions/4228853`

Considering these findings in the context of the research question, the use of figurative speech in API language representation is clearly a powerful resource to designers. However, it should be carefully analyzed from a communicative standpoint to prevent it from affecting users' interpretation negatively.

5.3.5 Documentation as API metacommunication

Documentation is a fundamental part of most APIs, despite being considered a secondary source of information for 'active programmers'. Some API usability studies regard documentation as a separate entity, focusing on software-only artifacts that compose the API (mostly source code), e.g. (51).

A perspective on APIs that brings designers and programmers as interlocutors in a communication process cannot leave documentation out of the scope of investigation. Documentation is a key resource designers have at hand to extend the API's expressiveness through a complementary signification system which references the API's signification system (which constitutes the API code and its interfaces). This does not mean that documentation should be *the* way to compensate for poor choices in lexical, syntactic and semantic levels of the formal language. These levels deserve a good deal of attention of their own. However, the pragmatics of APIs relies mostly on documentation, as it combines natural and formal languages to provide users with the envisioned scenarios, proper contexts and patterns of use, limitations, and so on. It is the opportunity designers have to 'fill in the blank' left behind by formal language discourse.

In addition, more than just being available as a secondary source of information, documentation should help to eliminate ambiguities and provide completeness to the overall API metacommunication. The full set of quality attributes discussed in subsection 5.2.1 can be observed to contribute to the design of good documentation content.

Evidence from empirical data collected reinforces the relevance of good documentation for programmers, especially when it comes to code examples. The Java API, for instance, has adopted a strategy to create two types of documentation: tutorials and reference. The general rule is that only tutorials contain code examples. However, tutorials are more suitable for learning general API concepts, while regular programming relies mostly on reference documentation. A number of Java bug reports show that programmers would like to have short examples in reference documentation (e.g. [22] [23] [24] [25]). In bug 4090313 ("Add code examples for methods in the API docs"), the reporter provides an interesting motivation for his request:

"*As a former Visual Basic developer, I found the sample code in online help an invaluable tool for learning how to use the language. Please consider adding this feature to your documents.*"

Users' preference for having code examples along with reference documentation has both communicability and usability advantages. First, it contributes to a more effective metacommunication, since code examples convey 'authorized' patterns of API use, and help to prevent ambiguous interpretations. From the usability side, having short code examples near to reference documentation contributes to better visibility, in terms of the cognitive dimensions framework. It also favors role expressiveness, since examples show concrete uses of an API entity.

An additional finding related to the role of documentation in metacommunication can be observed in evidence that comes from many PHP bug reports. PHP reference documentation presents each API function in a standard format, which describes its parameters names and expected types. It also describes return types for successful and error conditions. However, when a program makes a function call with an invalid parameter, there is a consistency layer responsible for parameter checking in the API that returns NULL every time it finds a problem with a parameter. This is mentioned in a single page in the language documentation[26], but not in the documentation of each and every function affected by this check, which would be not only useful but necessary, as evidence suggests. According to many user reports, this is not an easily perceived 'affordance' in the API[27] [28] [29] [30]. Evidence from these reports make it clear that redundancy, in this case, should be part of the documentation strategy to avoid this 'missed effect' on users. For instance, bug 65986 has been submitted by a *language contributor* (email address in php.net domain), and contains the following excerpt:

"*Never noted that NULL behavior as a generality like that, so that's good to know. I had been writing test cases for error handling code and "oci_fetch_all(null)" seemed to be the simplest manner to get the expected*

[22]`https://bugs.openjdk.java.net/browse/JDK-4090313`
[23]`https://bugs.openjdk.java.net/browse/JDK-4143455`
[24]`https://bugs.openjdk.java.net/browse/JDK-4148276`
[25]`https://bugs.openjdk.java.net/browse/JDK-4213311`
[26]`http://php.net/manual/en/functions.internal.php`
[27]`https://bugs.php.net/bug.php?id=60391`
[28]`https://bugs.php.net/bug.php?id=65362`
[29]`https://bugs.php.net/bug.php?id=65986`
[30]`https://bugs.php.net/bug.php?id=67038`

FALSE return (...) so I presumed this different behavior by oci_fetch_all() was a bug. Thanks for the explanation."

There are also interesting replies from users in bug reports 60391 and 65362:

"*Oh, that's cute. I've been developing apps with PHP for 5+ years and now this is first hearing of such 'built in' feature for me. Really nice. Good docs. Very intuitive. I think I'm starting to think about giving up PHP and move to something more life-ready (read - 'enterprise-ready').*"

"*Yes, I realize this is common behaviour, and not as such a bug in PHP. That is why I originally categorized this as a documentation bug. The problem isn't PHP's behaviour, it's that the documentation on the functions I mentioned is misleading about the return type. There is no mention of the possibility of a null return, and while you may get away with not mentioning this fact for most functions, in the case of these particular functions, it can easily lead to unpredictable and hard to find bugs.*"

Many studies highlight the key role of good documentation and code examples in users' process of learning an API. In 1998, McLellan et al. (30) concluded in their studies that code examples were a good way to demonstrate a library's capabilities, "allowing the programmers to form hypotheses about the library itself as well as the code example". This is in accordance with the abductive reasoning description of user's signification process, as discussed earlier. Also, Robillard and DeLine (17) concluded in their study about API learning obstacles that documentation of intent, code examples and matching APIs with scenarios are among the most relevant aspects of API documentation. Therefore, findings presented in this section corroborate with related work, providing new perspectives and deeper comprehension of existing knowledge.

5.3.6 Envisioned scenarios

Programmers usually search for APIs as a 'shortcut' to accomplish a higher goal. As such, learning inner details of APIs are not part of 'the goal', just a step towards it. Thinking APIs in terms of scenarios help designers to envision common uses for the software artifact under construction. In this context, the metacommunication template provides a structured view of the components that influence these scenarios.

In order to illustrate the role of scenarios to help users accomplish their tasks, a simple question "How do I check if a file exists in Java?" has more than

337k views in Stack Overflow [31]. This is an example that even basic tasks may not be trivially inferred from documentation, and that users frequently search for 'how to's' that match their goals. However, providing examples for all use cases of an API may be too costly, not to say impossible. This is an additional trade-off designers should take into account in the semiotic engineering of APIs.

Date and time operations belong to one of the most recurrent domains that affect users due to the lack of useful scenarios in documentation (at least considering evidence from the analyzed data). A plausible explanation for this is the fact that date and time representation and operations belong to an 'ordinary domain', which is part of our daily lives and is assumed to be well known by most programmers. However, there are lots of subtle details in this domain that may affect API learning and use. Users' abductive reasoning when dealing with technology can also help us to understand this phenomenon. In the case of a 'familiar' domain like date and time, users can easily formulate hypothesized rules about API behavior, since they (probably) know what the goals of a date-time API are. In addition to this potential 'active behavior' from users, the date-time domain familiarity may be a reason for designers to spare themselves from providing basic scenarios of use in documentation.

A typical example of the common difficulties users face with date and time operations is adding or subtracting months to a date. When the result of the operation falls near month limits, the operation can be the source of ambiguous interpretations. This issue is also related to leniency problems, as discussed in subsection 5.3.2. PHP bug id 64052 ("PHP DateTime Add and sub")[32] provides interesting evidence to support this finding. In the report, a user complains about API behavior when attempting to subtract a month from date '2013-03-31' and obtaining the result '2013-03-03'. The bug evaluator replies:

"*This has been filed so many times, and it's still not a bug. For a full explanation, see: http://derickrethans.nl/obtaining-the-next-month-in-php.html*"

From the evaluator's response, we can conclude that adding and subtracting months to a date is a common user goal (as expected), and yet, it is the cause of trouble to many programmers, especially in PHP.

Java also provides evidence of communicability problems in date and time APIs associated with the lack of common scenarios. For instance, Java bug 8037392 ("Period.between() returns incorrect value")[33] illustrates the difficulties faced by users when dealing with time period calculations. In this report, user shows an understanding of the method *Period.between()* that

[31]`http://stackoverflow.com/questions/1816673`
[32]`https://bugs.php.net/bug.php?id=64052`
[33]`https://bugs.openjdk.java.net/browse/JDK-8037392`

differs from its actual behavior. It calculate the difference between two dates, and the user expects the operation to be symmetric. However, this is not the case, because it calculates the period in terms of its ‘conventional’ parts (months, days, hours, etc.). According to the evaluator:

“*[user] appears to want a rule where the days are calculated based on the original month length, not the one that results once the month-year difference is applied. The OP [user] is not wrong, its just that its not how we choose to make the calculation in java.time.*”

This is also a case in which code samples could help eliminate the misinterpretation of the API’s meanings, by making clear this type of boundary condition.

The role of scenarios is an addition to the discussion related to documentation presented in subsection 5.3.5. They provide a higher level and concrete description of the intended API’s goals, as paraphrased by the metacommunication template. Also, they contribute to the consolidation of a “rule of programming discourse” (31) by providing patterns to be reused by programmers in order to achieve a range of common goals.

5.3.7 Common ground

Effective communication relies on shared knowledge and assumptions between interlocutors. In the context of API metacommunication, shared knowledge comprehends at least basic computer science and programming language concepts, and also domain-specific knowledge, depending on the services provided. Shared assumptions can be associated with cultural aspects of programming like, for instance, conventions and styles commonly adopted by the API language community and programmers in general.

These elements provide the *common ground* that allows mutual understanding between API designers and users. Violation of the principles that govern this shared knowledge and assumptions can affect metacommunication effectiveness. In programming, common ground between people can be negatively affected when computer’s needs are privileged, optimizing time and space dimensions of program execution without taking into account possible effects on programmers.

A well-known example of violation of common ground in the Java API is the *Date* class. This class uses a zero-based representation for months, in which 0=Jan, 1=Feb, and so on. Therefore, code using the *Date* class can be very misleading, like the following sample:

The execution of this code produces the output string ‘Wed Mar 03

```
Date d = new Date(2015, 1, 31);
System.out.println( d.toString() );
```

00:00:00 BRT'[34]. Strange as it may seem, this output string results from the combination of zero-based month representation with lenient behavior. The parameters are interpreted as 'Feb 31', an invalid date that is automatically adjusted to a consistent date representation, i.e. 'Feb 31' is interpreted as 'Feb 28' + 3 days, which equals 'Mar 03'). In spite of being a notorious case of bad API design, this is a simple example to illustrate that violating people's beliefs and assumptions may have severe impact on API communicability and usability.

It is quite common to find bug reports in Java associated with this issue, and closed as 'not an issue'. For instance, in bug 6953809[35] a user submits code based on the *Calendar* class (which also uses a zero-based representation for months). User assumes that his code is correct (an instance of 'Looks fine to me' failure – see 5.2.3), and blames the API for the wrong output. He complains about the severity of the (supposed) bug: "*This is a serious bug if used for commercial applications! I want to use it for astronomy and it produces garbage*".

The Java *Calendar* class provides another example of the effects of disregarding user's knowledge and assumptions in API metacommunication. The class provides two generic methods, *set()* and *get()*, that respectively sets and gets the values of each date component: year, month, day, and so on. However, it shows an unusual behavior, since the object's internal fields are recalculated only when a call to *get()* occurs. This can be a source of surprise and misunderstanding to many users, as evidenced in bug reports analyzed. For instance, in bug 7072337[36] ("Call to Calendar.get() affects the calendar date and becomes incorrect"), user is puzzled by a side effect of calling method *Calendar.get()* in the code. He states:

"*Both parts are exactly the same except for the call : c1.get(Calendar.WEEK_OF_YEAR). Whatever a Java object is, doing a get on it should NEVER affects its state.*"

A number of problems with APIs can be associated with grounding issues, and many of them have well known causes, like the ones discussed in this subsection. However, a communicative perspective provides a different framing for this category of problem, contributing to new insights and greater awareness

[34]the 'BRT' part may vary, according to the default time zone
[35]`https://bugs.openjdk.java.net/browse/JDK-6953809`
[36]`https://bugs.openjdk.java.net/browse/JDK-7072337`

about its nature.

5.3.8 Specific domain concepts

This subsection deals with a subject that is, to some extent, a special case of the previous discussion about common ground. However, differently from the previous topic, this subsection deals with findings that refer to more specific domains, external to the programming environment and governed by its own conventions or formal rules. In this context, two types of implications for API metacommunication have been identified.

First, the use of concepts from external domains in APIs should observe the domain's terminology and conventions, in order to prevent users from missing API's affordances, or misunderstanding them. As an example, Java provides a *Set* interface that defines the implementation of various collection classes with set semantics. The *Set* interface documentation[37] describes it as a model of the mathematical set abstraction. However, common set operations are defined by unusual names. For example, intersection of two sets is obtained by calling method *Set.retainAll()*. Java bug report 4154473 ("Add difference operation to Collections")[38] provides evidence showing that user missed Set interface's affordances:

"*The Set class claims to represent the mathematical set operations, but it does not have any methods to find the intersection, union, or difference of two sets. Can these enhancements be added?*"

The second situation occurs when an API refers to specific standards or conventions that may not be widely known. When this is the case, users may create wrong assumptions about API behavior. A recurring example of this issue is the implementation of the ISO 8601 standard for exchanging date and time data, especially with respect to week numbering. In spite of being a formal standard, it recommends a way of numbering weeks of the year that is a frequent source of astonishment and misunderstanding among API users, both from Java and PHP. For example, PHP bug 65694[39] contains the following user statement:

"*For an internet tool I need to get a list of weeks after having selected one year and one month. By testing I found a BUG in the year 2016. PHP gives me: 1 January 2016 the number of week: 53 (instead of 1) January 31, 2016 the number of week: 4 (instead of 5)*"

The following excerpt shows the evaluator's reply:

[37]`http://docs.oracle.com/javase/7/docs/api/java/util/Set.html`
[38]`https://bugs.openjdk.java.net/browse/JDK-4154473`
[39]`https://bugs.php.net/bug.php?id=65694`

"*This seems to work just fine. PHP uses the ISO8601 calendar which indeed has a week '2015-53' for the period Mon Dec 28, 2015 to Sun Jan 3, 2016. Please see http://en.wikipedia.org/wiki/ISO_week_date for more information.*"

Metacommunication issues associated with lack of knowledge about formal standards may be viewed as users' fault, by not reading its specification. But, again, users formulate strong hypotheses in their abduction when dealing with more familiar domains. And sometimes standards contradict 'common sense'. Being aware of these aspects' possible implications, API designers can make an extra effort to effectively communicate their decisions with respect to the use of standards.

5.3.9 Identity and comparison

Issues concerning aspects of object identity and comparison are among the most frequent topics found in the analyzed bug reports. Most of these bug reports show closer association with programming language issues, as the language is responsible for the basic rules which define how entities are compared in a program. Therefore, API designers should carefully examine the interaction between an API and its underlying programming language, since the language defines the lexical, syntactic and semantic basis for API construction and use.

In this context, there is evidence from empirical data that provides insights about the nature of the problems that may occur with respect to identity and comparison aspects. For example, Java bug report 7060309[40] shows a user complaint about methods *Timestamp.equals()* and *Date.equals()* breaking the symmetry of the comparison operation. According to the user, "a comparison *Date.equals(Timestamp)* results in true, the comparison *Timestamp.equals(Date)* results in false". The evaluator provides a reasonable explanation for this behavior, but this is certainly a subtle and misleading API behavior that could have been avoided.

The PHP language has lots of comparison issues that affect how its APIs behave in this context. Due to its loose typing and comparison characteristics, many PHP API operations surprise users with unexpected behavior with respect to parameter and return values. As a consequence, some API operations offer two kinds of type comparison: loose or strict. As a general rule, loose comparison is the default behavior, for backward compatibility reasons.

Bug 66583 ("array_search always returns 0 for string elements for a mixed

[40]`https://bugs.openjdk.java.net/browse/JDK-7060309`

type array") [41], for example, shows a surprised user with the behavior of function *array_search()* when searching for a string in a mixed type array. The *array_search()* function has three parameters: *needle* (what to look for), *haystack* (an array where 'neddle' will be searched), and *strict* (boolean that indicates if comparison should be 'strict', default is 'loose'). The main problem is that the *strict* parameter is, by default, *false*, and loose comparison returns *true* when comparing any string to integer value zero.

In the report, user does not perceive the 'strict' parameter and its semantics, and gets unexpected results when searching for a string in an array that contains zero as an element. After receiving an explanation about this parameter and the API behavior, user replies:

"*From the user perspective (which is mine), I don't follow the logic. So, yes, I did read the array_search() documentation. But, in my case, the documentation didn't helped me. However, I must admit, that I wasn't smart (or idiot) enough to say to myself: "hey! There is this 'strict' option here. Just give it try, even if it's not obvious that it's sensible." In other words, the documentation may be more explicit about what is strict and loose search cases.*"

The *array_search()* example may be as well regarded as user error, for a careless reading of documentation and not perceiving the strict parameter. However, this is a common mistake with potentially severe impact on the stability and correctess of programs that use it. Even when users are aware of this condition, lapses may occur, which is associated with the cognitive characterization of the 'error-proneness' dimension. Additionally, this same issue affects other functions in PHP. In hindsight, API designers would probably adopt a different strategy for this decision if they had the opportunity to redesign the API (or the language).

5.3.10 Classification of collected evidence

This subsection presents the results of bug reports' classification according to the process described in chapter 4 and the criteria detailed in section 5.2. The main objective is to provide a qualitative view on the distribution of bug reports after their categorization, exploring existing relations among these categories. Since the data sample is composed of a non-randomly selection of bug reports, is not suited for statistical analysis. However, the co-occurrence of certain patterns provides interesting insights about the nature of collected data under a communicative perspective.

[41]`https://bugs.php.net/bug.php?id=66583`

The first classifications of bug reports provide an overview of results for each category previously discussed: failures, template components, effects and cognitive dimensions. Then, we examine the combination of categories in multi-level data aggregations to investigate the main co-occurrences and discuss their interpretation. The presentation of results follows the same order used in the investigation process: first, we identify users' communicative failures (the entry point in bug reports' analysis), followed by the classification of metacommunication intent (template components) and its effects (effect type and cognitive dimensions associated).

Bugs per failure type

Subsection 5.2.3 described the tags used to characterize the communication failures that may occur when users try to understand an API's design. These tags offer a detailed diagnostic of possible users' breakdowns, based on symptoms associated with complete, partial or temporary failures.

Figure 5.7 depicts the distribution of the analyzed bug reports according to the classification of their failure tags. As each bug report can be associated with more than one tag, some bugs have been counted in more than one category. For instance, there were many cases of bug reports associated with three tags, as follows: 1) the user presented code that, in her opinion, should work as expected. This is interpreted as a complete failure ('looks fine to me'); 2) the code presented by the user can also be interpreted as a temporary failure ('why doesn't it?'), which occurred while the user tried to use the API's features, without success ; 3) if the user provides a 'workaround' to the problem, this represents a partial failure ('I can do otherwise').

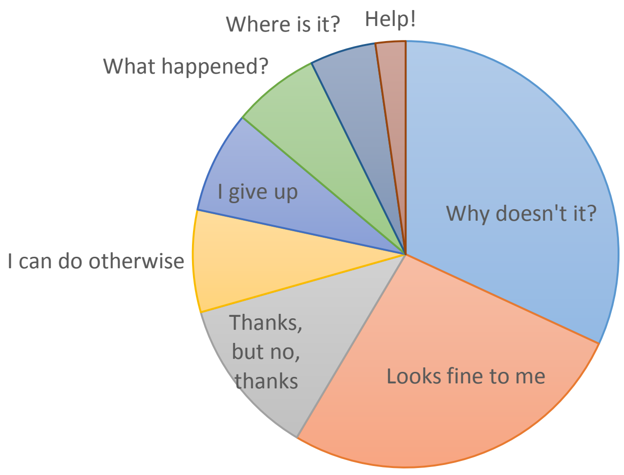


Figure 5.7: Bugs per communication failure

The most frequent tag in the failure classification was 'why doesn't it?' (a temporary failure in which user tries to clarify the API's intended signification by autonomous sense making). This may be explained, in part, by the interpretation given to the tag in the context of the bug reports' analysis. This tag has been associated with a number of bug reports that contained source code showing the user's attempt to make the API feature work as expected, without success. In most cases, tag 'why doesn't it' was the symptom of a temporary failure that preceded a complete failure, in which the user thought that her code was correct ('looks fine to me'), or when the user reached a dead end ('I give up').

The second most frequent tag identified was 'looks fine to me', which characterizes a complete failure. This was the most severe communicative breakdown, since the user wrongly assumes to have accomplished her goal, without being aware that there has been a failure in the communication with the API. This is consistent with the nature of the collected evidence, since bug reports that receive a 'not a bug' classification commonly involve a situation in which the user is convinced that the 'problem' is on the API's side, and that the code provided should work as expected.

The next most frequent failure corresponds to the 'thanks, but no, thanks' tag. It represents the cases in which the user understands the design solution available, but rejects some of its aspects. This may occur when an API provides an affordance which conflicts with the user's expectations, or when it lacks an affordance which is expected or assumed to be available by the user. It also applies to situations in which an API is used in unexpected or unintended ways.

Lastly, it should be noted that the number of bug reports in which the user presented a workaround ('I can do otherwise') is smaller than other types of failures, which indicates that a number of users could not (or have not tried to) find an alternative way to accomplish their goal.

Bugs per template component

As previously mentioned, the metacommunication template components paraphrase the designer's intent behind the construction of an API. In the course of the evaluation of bug reports, the classification of each bug's template components comprehended the examination of the content of the user's and the evaluator's discourse contained in the report, as well as the analysis of comunication breakdowns. The main objective of this classification was the selection of template components that corresponded to aspects of design intent that could have affected API metacommunication, as evidenced by the situated API use specified in the bug report.

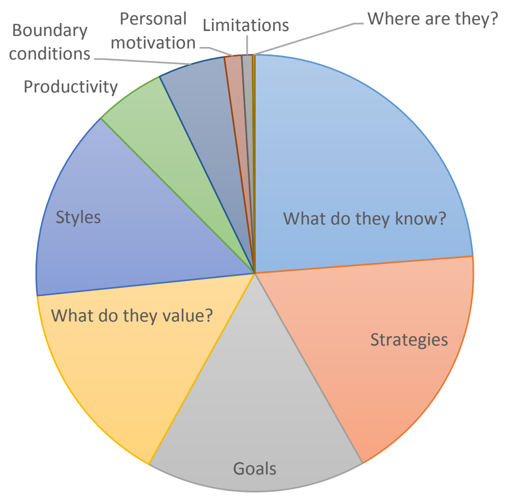


Figure 5.8: Bugs per metacommunication template component

Figure 5.8 shows the distribution of bugs per metacommunication template component. Once again, the sum of all bug reports per category exceeds the total number of reports, since there are cases in which the report is associated with more than one component.

The most frequent template component in the analysis of bug reports was 'what do they know?'. It describes two types of events: 1) user lacks knowledge about some aspect of the API or the domain, which could possibly be more effectively signified in API metacommunication; 2) user's previous experience and knowledge may have influenced her interpretation about the API's affordance being discussed. This emphasizes the need to take into account the knowledge required from a typical user to understand an API's affordance. It is also a hint for the importance of previous users' experiences and knowledge in the conduction of their abductive reasoning while interacting with a new API.

The second most frequent component was 'strategies', associated with situations in which there is a mismatch between design intent and user's preferences in the selection and combination of API and language elements to accomplish the intended tasks. It was closely followed by the 'goals' component, which corresponds to reports in which the API design does not properly fit user's objectives, either by not solving the user's problem as expected, or by not offering the required affordances. Both of these parts of the template are commonly associated with mismatches between the API's intended scenarios of use and users' needs and preferences. Metacommunication enhancement through the provision of clear scenarios and examples of API use may contribute to reduce misinterpretations associated with these categories of intent.

Still among the most representative findings in the intent dimension, the 'what do they value?' component was used to indicate conflicts between design intent and aspects that users consider as relevant when it comes to using an API effectively. For instance, adherence to common programming conventions is usually valued by users as an expected characteristic for an API, being a common source of breakdowns when violated.

Bugs per effect type

The classification of effect type provides an indication of how API metacommunication affected users in the situations that led to bug reports, with respect to their perception and comprehension of its affordances. Figure 5.9 illustrates the distribution of bug reports according to their effect on users. As described in subsection 5.2.2, there are two types of effects that do not show up in this study: 'successful' and 'unexpected'. The first type was naturally excluded from the type of evidence collected, since it represents the absence of communicability issues. The second type could not be identified in the bug reports, due to its similarity to the 'successful' effect, with the distinction that the API affordance surprises the user in some way (without being rejected). Differently from previous categories, only the most evident effect type has been associated with each bug report (i.e. one effect type per report).

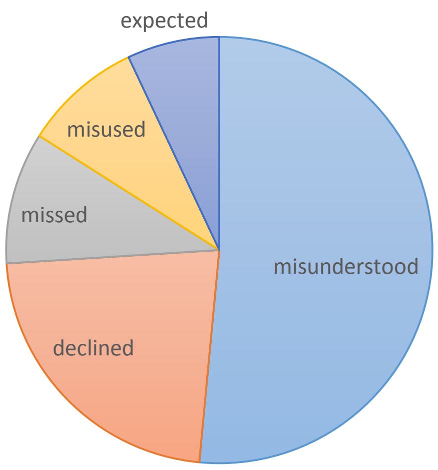


Figure 5.9: Bugs per effect

According to the results, 'misunderstood' was the most frequently identified effect type. It applied to situations in which users did not fully comprehend an API's affordance, due to mismatches in their conceptual or semantic model with respect to the actual API behavior. From a communicative perspective, this result can be associated with two kinds of mismatches: 1) API communicability is satisfactory, but user does not belong to the intended audience; 2) user matches the envisioned audience profile, but API lacks some communicability aspects. A third possibility is the combination of both problems (communicability issues, interlocutor mismatch). In the studies conducted as part of this research, bug reports that clearly indicated user's lack of basic knowledge required to understand API's affordances have been discarded. This was an attempt to exclude less representative evidence from the core dataset, dismissing cases in which communicability problems were essentially associated with gaps in users' programming education.

The second most frequent effect type was 'declined'. This is, by itself, an interesting result from a communicative approach to investigating APIs, since it shows that it is not uncommon to find situations in which users actively reject the proposed design. This effect usually corresponds to the communication failure described by the 'thanks, but no, thanks' tag. There are cases in which 'declined' is preceded by a 'misunderstood' effect instance. For example, it occurs when a user reports a 'bug' by misunderstanding the API design and, after receiving an explanation by the evaluator, realizes how the feature works, and rejects it for violating her preferences or needs.

The 'declined' effect, together with the diagnostic provided by the 'thanks, but no, thanks' tag, allows us to distinguish between two situations: 1) user understands and accepts the API's affordances; 2) user comprehends the API's features, but rejects at least some aspects of its design. This refinement in the characterization of users' reception of API metacommunication is not usually accounted for from API usability evaluation studies. These studies are mostly oriented to evaluate users' ability to achieve their goals when learning and using an API, apart from possibly existing mismatches between their preferences and the actual artifact's design.

The fact that more than half of the bug reports analyzed can be associated with understanding issues is an interesting result, especially considering that bug reports which clearly revealed users' lack of knowledge were discarded. It means that the remaining bug reports are more 'qualified' with respect to these misunderstandings, which reinforces the applicability of a communicative approach to the study of APIs.

Bugs per cognitive dimension

The classification of bug reports with respect to the effect dimension also included the characterization of cognitive impact on users. This classification was based on the cognitive dimensions of notations, as described in subsection 5.2.2. The distribution of bug reports per cognitive dimension is illustrated in figure 5.10. As a general rule, multiple cognitive dimensions were associated with a single bug report.

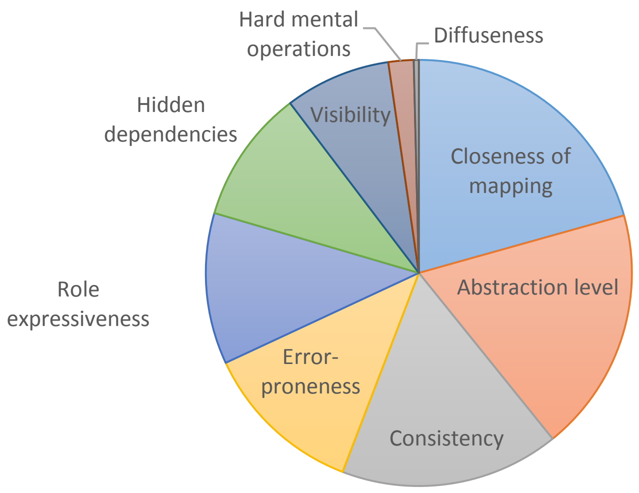


Figure 5.10: Bugs per cognitive dimension

‘Closeness of mapping’ was the most frequent dimension listed in the bug reports’ analysis, closely followed by ‘abstraction level’. As explained in subsection 5.2.2, this dimension was interpreted by taking the users’ perspective, indicating that an API’s affordance did not represent domain concepts in a way that could be easily grasped by the user. In the user’s abductive process, what is close to the domain from the designer’s perspective may not be as easily comprehended by the user, leading to interpretations that are not compatible with the design vision, and possibly leading to a complete failure.

‘Abstraction level’ issues occur when there is a clear impedance between the features and operations provided by an API and users’ expectations and interpretations of its design. It corresponds to situations in which users try to use the API’s features to accomplish tasks that do not fit its envisioned scenarios of use. It may also be the case that users face difficulties to understand and use the API’s features because their goals rely on a higher level of abstration than the one provided by the API. Abstraction level mismatches may also refer to situations where users decline from API’s design. For instance, sometimes users identify a ‘broken abstraction’ that makes knowledge of its inner details necessary for its proper use. This may contribute to users’ rejection of the related API feature.

‘Consistency’ was the third most frequent cognitive dimension identified in the studies. This dimension applies to bug reports illustrating situations in which the user identifies traces of inconsistencies in API’s affordances that may be associated with representational or semantic issues. Consistency issues are also frequently related to user's rejection of API's features, since it sometimes violate user's preferences and needs ('what do they value?').

Lastly, 'error-proneness' was also among the most frequent cognitive dimensions, and it was used to describe situations in which API's affordances could be easily misused or misinterpreted, offering little protection against mistakes or lapses. In spite of being closely related to usability aspects of API's, error-proneness impact on users may be increased by communicability issues that affect users' interpretation of its features or behavior. For instance, an API can be more error-prone when the combination of signs that encode its meanings (static, dynamic and metalinguistic signs) is inappropriate, possibly harming users' interpretation of its design and contributing to errors.

Bugs per template component and effect

This subsection provides a two-level aggregation between the classification of template components and metacommunication effects. The main objective is to support the analysis of co-occurrences of categories that may lead to further insights. To provide an overview of these nested clusters of bug reports, figure 5.11 depicts a treemap structure illustrating the aggregation of bug reports per template component, and for each component, the most frequent effects identified. The size of each box is proportional to the number of elements in the set, and the color of inner boxes also reflects the number of bug reports it contains (darker blue means a larger set, as also represented by its size).

The diagram shows that the most frequent co-occurrence of intent category and effect on users was ('what do they know?', 'misunderstood'). The 'misunderstood' effect also appears on the top position in the next three template categories by size, 'strategies', 'goals' and 'styles'. This is consistent with the results previously described in this section, and reinforces the importance of conducting users' abductive reasoning to an interpretation which is consistent with the artifact's encoded design vision.

An interesting observation in the next two categories ('productivity', 'what do they value?') is the 'declined' effect appearing as the most frequent. This is an indication that when API metacommunication does not match a user's profile, preferences and motivation for using it, there can be a negative effect on the user's reception, who may end up looking for an alternative solution. The 'declined' effect was also frequently identified in the top aggregations, indicating that is a common result of API metacommunication, at least in the analyzed dataset.

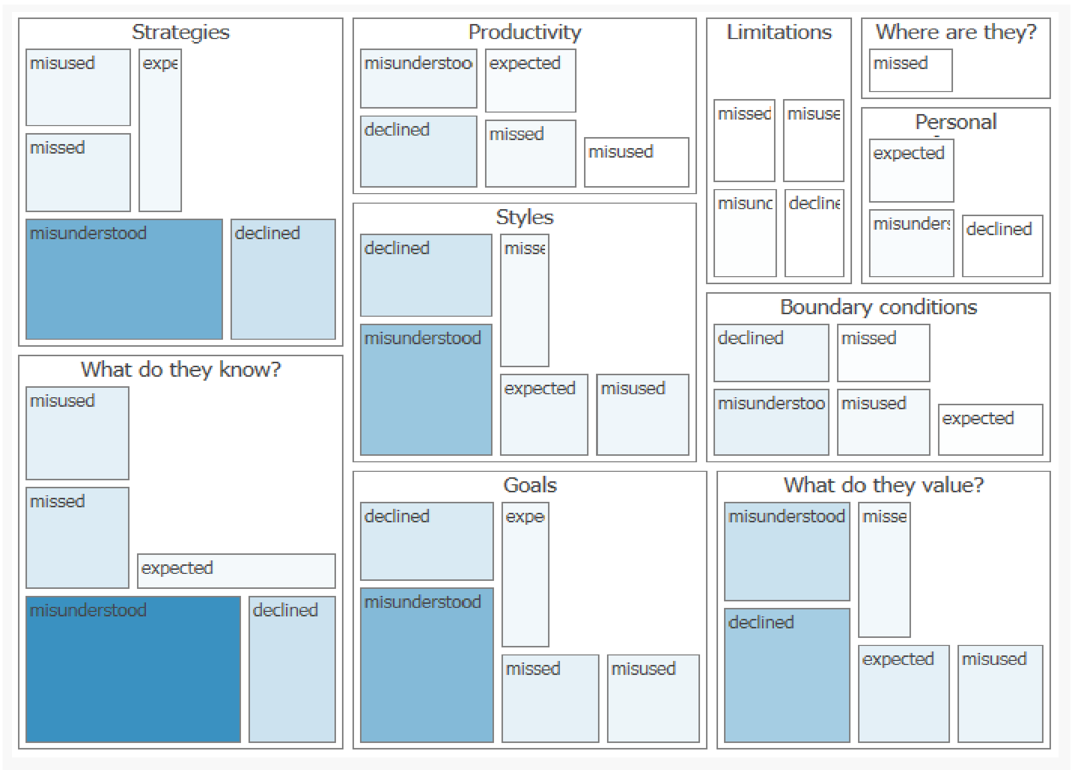


Figure 5.11: Bugs per template component and effect

Bugs per template component and cognitive dimension

Figure 5.12 illustrates a two-level aggregation by template component (intent) and cognitive dimension (cognitive effect).

The 'abstraction level' and 'closeness of mapping' dimensions appear in the top positions in the largest clusters of categories, confirming their relevance as illustrated previously in the analysis of the cognitive dimensions. An interesting result appears in the 'what do they value?' category, since the top cognitive dimension is 'consistency'. When compared to the previous clusters (template x effect), the top effect in this category was 'declined'. This suggests that users tend to reject API features that provide inconsistent representations or semantics. 'Closeness of mapping' appears next as the most frequent cognitive effect that may lead users to reject the artifact's design.

Bugs per template component and failure type

The next aggregation type presents the distribution of bug reports per intent category (template component) and failure classification. Figure 5.13 provides an overview of the co-occurrences of communicative failures inside the most frequent categories of intent.

The main cluster shows the top failures in the 'what do they know?' component: temporary failure 'why doesn't it' and complete failure 'looks fine

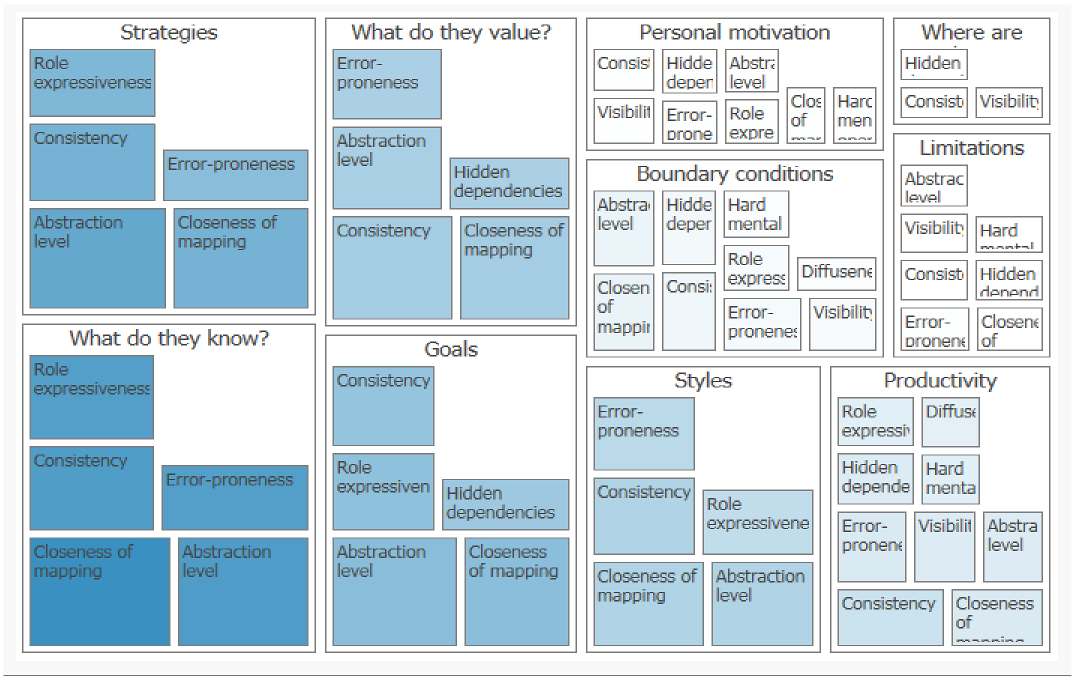


Figure 5.12: Bugs per template component and cognitive dimension

to me'. As previously discussed, 'why doesn't it?' was a frequently identified failure due to its association with sample code that revealed an attempt to use the API. However, it usually preceded a partial or complete failure, which also explains the high incidence of 'looks fine to me', due to the multiple classification of failures in the reports.

The high occurrence of 'looks fine to me' failure corroborates with the numerous 'misunderstood' effects in the template x effect aggregation. In general, this complete failure occurs when the user is not conscious of failure, which can be frequently explained by misunderstandings in the interpretation of the designer's discourse encoded in the API.

It should also be noted that in the 'what do they value?' category, the partial failure denoted by the 'thanks, but no, thanks' tag appears in the second position, after 'why doesn't it?'. This is consistent with the high number of occurrences of the 'declined' effect in the template x effect aggregation. An analogous result appears in the 'productivity' and 'personal motivation' boxes.

Bugs per template component, failure, effect and cognitive dimension

The last aggregation of results shows a four-level cluster of bug reports, distributed by template component, failure, effect and cognitive dimension. Due to space restrictions and the combinatorial effect, only the top 3 results inside each category has been selected. Figure 5.14 presents a map of the results reflecting this organization of data.

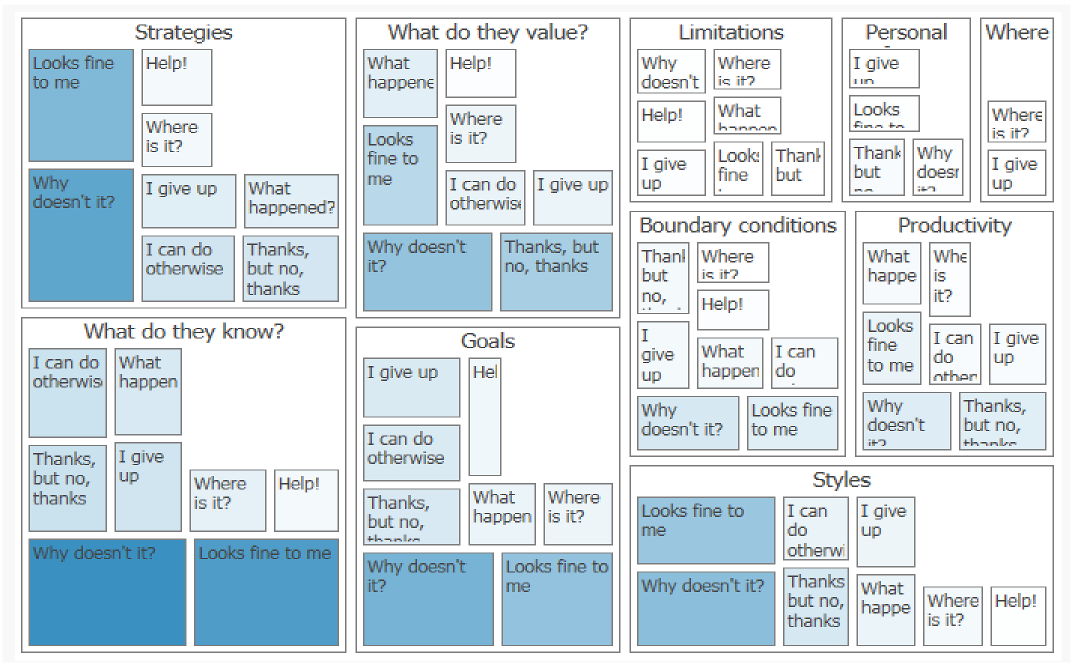


Figure 5.13: Bugs per template component and failure type

The diagram shows three top level categories of intent (template components): ‘what do they know?’, ‘strategies’ and ’goals’. In the first category, despite the larger cluster being located in the ‘why doesn’t it?’ box, the ‘looks fine to me’ failure shows more significant results, since it is a complete failure. Inside this failure, we can see that the ‘misunderstood’ effect appears as the most frequent, associated with the top cognitive impact descriptions: ‘closeness of mapping’, ‘abstraction level’ and ’error-proneness’. It should be noted that the ‘why doesn’t it’ box shows a similar configuration with respect to the ‘misunderstood’ effect, which may be explained by the high incidence of bugs classified with both failure types.

It is also interesting to observe that these failure types (‘why doesn’t it’ and ‘looks fine to me’) are the most frequent in all the top 3 top level categories (intent). The ‘thanks, but no, thanks’ failure appeared in the third position in two categories, except for the the ‘strategies’ category, where the ‘I can do otherwise’ appears as third most frequent failure. This can be interpreted as users searching for alternative strategies when their primary strategy could not be achieved, both because she could not understand the design (‘misunderstood’) or by having rejected it (‘declined’). The secondary strategy also appears when the user expects an affordance that was not provided (‘expected’ effect).

The map depicted in figure 5.14 provides an insightful overview of the results obtained in the categorization of bug reports, in spite of showing only the top 3 results inside each category. The domination of the ‘what do they

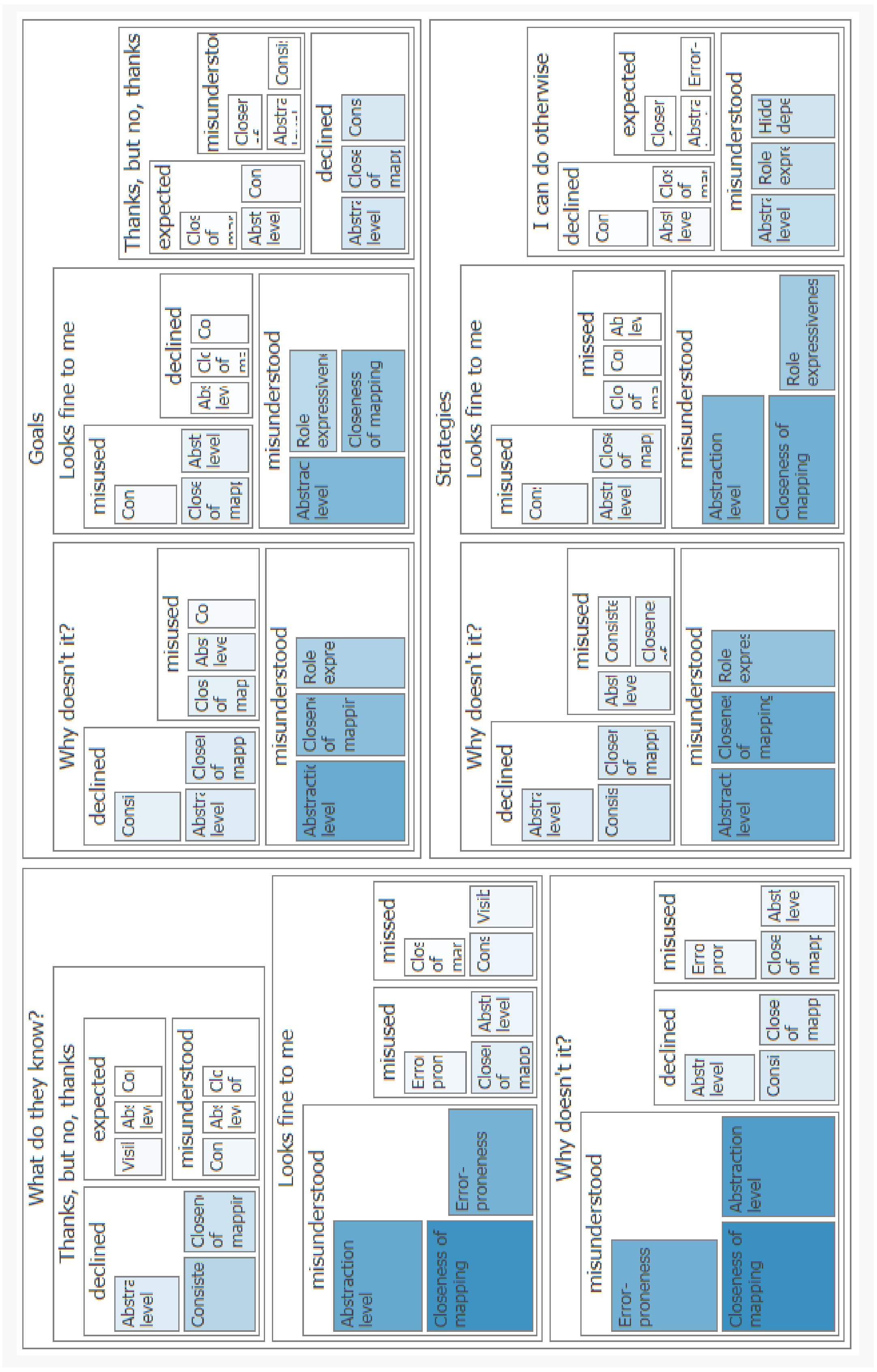


Figure 5.14: Bugs per template component, failure, effect and CDN (top 3)

know?' category shows the relevance of taking into account the two types of knowledge involved: 1) what users already know about other languages and APIs, and how this knowledge affects their reception of designer's message; 2) what they (don't) know about the API's affordances and its domain, which corresponds to what should be effectively encoded in API metacommunication. The high incidence of the 'strategies' and 'goals' categories also reinforces the importance of clearly communicating an API's envisioned scenarios and contexts of use, since they support users' interpretation and decision making process about adopting or not a certain API affordance, depending on its compatibility with users' goals and strategies.

Next chapter further discusses the results and findings presented in this chapter, analyzing their implications from a broader perspective of the study of API's. It also describes the main contributions of this thesis with respect to current knowledge in API design and evaluation, their implications, limitations and future work.

6
Final Discussions and Conclusion

This chapter concludes this thesis by providing a broader discussion of the research findings detailed in chapter 5 with respect to their contributions to scientific knowledge. It also discusses this research work's limitations and proposes future work to expand the possibilities of the current approach.

The chapter begins by discussing new insights about the case of the PHP Curl API, introduced in chapter 1.

In addition, this chapter promotes a reflection about API metacommunication as an expansion of the current concept of behavioral contract, introducing the notion of pragmatic contracts.

6.1
Revisiting the "most dangerous code in the world"

This section analyzes the communicative aspects involved in the example introduced in chapter 1, using some of the concepts and tools presented in chapter 5 and contributing to new insights about this case study.

Subsection 1.2.1 described a compelling example of problems that may occur in the interpretation of a specific API's meanings, based on the work of Georgiev et al (21). This work refers to a number of security problems in the use of SSL libraries which, according to the authors, are mostly due to misinterpretations and misunderstandings of its parameters, options, return values, and so on.

The main objective of revisiting this example is to provide a new perspective of the same problem by reflecting on the API's communicative aspects and how they may influence its proper use by programmers. The basic idea is to analyze this specific API in terms of the organization of the problem space proposed in this thesis. However, this should not be taken as a claim that the use of the conceptual tools proposed in this thesis would automatically contribute to the creation of a 'better' or 'more usable' API. The conceptual tools we propose are to be used 'by people', who are necessarily subject to the contingencies of their work situation and also to their level of skill and professional knowledge. In other words, we are not proposing (and we do not believe it is possible) to take people out of the loop in API design. We will thus show how our tools can promote a reflection on a concrete example of an API that has known issues, in order to discuss how our approach can contribute to greater awareness about the communicative aspects involved in the matter and how they may potentially affect users' interpretation of its meanings.

Georgiev's example illustrated problems with the PHP Curl extension, especially with respect to the options that dictate how the library checks the authenticity of the remote peer of an SSL connection, through the verification of its digital certificate contents and its domain name. The options that control this behavior are CURLOPT_SSL_VERIFYPEER and CURLOPT_SSL_VERIFYHOST. As a reminder of the previously introduced example, listing 6.1 reproduces the same code shown in listing 1.1.

Listing 6.1: Example of insecure PHP Curl code

```
$ch = curl_init("https://localhost");
curl_setopt($ch,CURLOPT_SSL_VERIFYPEER , false);
if( curl_exec($ch) ) echo "Request OK";
else echo "Error: ".curl_error($ch);
curl_close();
```

This helps us to illustrate the discussion in subsection 5.3.1 about APIs with a 'narrow protocol'. This one has a generic interface (function *curl_setopt()*) that controls a large number of options for its behavior, including the ones mentioned above. As it is usual with interfaces having numerous options, the library provides default values for most of these options. Following abductive reasoning, that is tightly associated with common practices and even common sense (95), users frequently adopt the basic option settings that they believe will allow them to accomplish their goal, sometimes without fully understanding what is really happening.

For instance, option CURLOPT_SSL_VERIFYPEER controls if the peer's digital certificate will be checked or not, a procedure that verifies the validity and authenticity of the remote host. Its default value used to be 'false' in older versions of the Curl library, which means that the default behavior was insecure, but usually worked 'out of the box'. However, it was not suited for production use, due to its insecurity. After its default value changed (CURLOPT_SSL_VERIFYPEER='true'), users commonly faced problems using the API to connect to SSL servers that used a self-signed certificate, since the library refuses to connect to a host when it cannot verify its identity through a trusted certificate (a self-signed certificate is a free alternative to the use of certificates generated by third-party certification authorities). The 'quick solution' to this problem is to turn off the security check (exactly what code in listing 6.1 does). This 'solution' can be easily found in code samples from the Internet, as it spread almost like a 'pattern' in the use of this library to establish SSL connections in PHP applications.

The API documentation is organized as a reference for each function, its description, parameters and return semantics. It does not provide common scenarios of use to illustrate examples to accomplish a recurrent goal (see 5.3.6). Despite the existence of many widely used APIs which provide only reference documentation of its features, code samples contribute to illustrate common scenarios and offer an authoritative view of recommended and intended use cases, preventing the informal construction of inappropriate 'idioms' by a community of users. In the PHP API, the documentation pages allow users to post comments, and it is not uncommon to find bad examples of code among these comments, which may sometimes be viewed by unexperienced PHP programmers as a recommended code sample.

The library options CURLOPT_SSL_VERIFYHOST and CURLOPT_SSL_VERIFYPEER can be regarded as examples of problematic choices of identifiers, for various reasons. The difference in meaning between 'verify host' and 'verify peer' may not be clear to programmers, either to the ones who write code or to those who read it later. Using the metacommunication elements terminology, the naming strategy (expression) chosen by designers is ambiguous and can be regarded as a bad choice of metaphors that may affect users' interpretation negatively. There can also be common ground issues related to the required knowledge about specific SSL details. The cognitive impact of this representation may be described in terms of the 'role expressiveness' and 'abstraction level' dimensions, since these terms may be easily mistaken, and probably carry abstractions that are not suited to most users' goals.

Many option names in the Curl library that follow the pattern '*verb + object*' have boolean semantics, and consistency is broken in the case of 'CURLOPT_SSL_VERIFYHOST', since it takes '0', '1' or '2' as possible values. If a programmer's abductive reasoning leads to the assumption of boolean semantics for this option (based on an expected consistency of API's elements), she may set this option with a 'true' value, interpreting that 'verify host = true' is the most secure option. However, due to PHP's lenient conversion between types, the parameter's boolean value 'true' is automatically converted to integer value '1' when used in an numerical context. Since '2' is the most secure option value (meaning full host name verification), the result is a less strict verification of host identity, which is not probably what users would like to achieve.

Considering the average profile of PHP users and their knowledge about SSL specifics (‘who are the users?’, ‘what do they know?’), a finer-grain control over SSL checks may be a useful feature to advanced users, but it is quite reasonable to think that average PHP programmers just want to achieve their goal of establishing an SSL connection (‘what do they need or want to do?’) as quick and easily as possible (‘in which preferred ways?’), in order to obtain a trusted and secure connection in their application (‘why?’).

The PHP Curl API metacommunication message has the potential to cause the ‘misunderstood’ effect on users when they write (or read) code like listing 6.1 and assume that it has an acceptable level of security. This would typically be tagged as a ‘looks fine to me’ failure: the user is unaware that her goal was not fully achieved.

Concerning the intent behind the API, the metacommunication template has the potential to make designers aware of the design space components that should be taken into account, helping to prevent the mismatch between the API’s affordances and users’ goals and needs. Also, designers can refer to the metacommunication elements organized in the dimensions of expression, content and intent, reflecting on the use of these elements and on the attributes that affect their quality. For instance, reflecting on the ambiguity and the metaphorical use of identifiers in the Curl API could prevent the poor choice of identifiers explained earlier in this section.

Additionally, the PHP Curl API has a particular characteristic that can also be analyzed from a communicative perspective: it is an implementation of a thin layer on top of the C Curl library [1], a widely used C library for multiprotocol file transfer. Again, the metacommunication template has the potential to make designers aware of their intent when creating a library to a large audience: do ‘regular’ PHP programmers have the same knowledge, needs, preferences and goals than C programmers ? Does a simple ‘translation’ of a C library to PHP generate an adequate abstraction for the intended audience? What differences between these two languages may affect the result of this library ‘translation’ ? Do PHP programmers have to know any details of the underlying C library to effectively use the PHP version of the library ?

In conclusion, this discussion about the PHP Curl API is not meant to make claims about ‘what should have been done’ or ‘what would have worked’, but rather provide a shift of perspective to shed new light over a class of problems that are still a source of difficulties to programmers of all levels of expertise and background knowledge.

[1]http://curl.haxx.se/libcurl/

6.2 Metacommunication as a 'pragmatic contract'

This section extends the current notion of software contracts, introducing a new abstraction that encompasses the main implications that derive from viewing APIs as a communication process between its designers and users.

Software contracts, also known as 'design by contract' or 'contract programming', describe a concept introduced by Bertrand Meyer (96) that promotes an analogy between software interfaces and legal contracts, in which both parties have duties and rights. A software contract is based on logical assertions describing program state that should be valid before the call (preconditions) and after the operation has been performed (postconditions). If client users satisfy the preconditions (their 'duty'), they have the 'right' to assume that postconditions will be validated. Conversely, the software provider has the 'right' to assume that users are compliant with the preconditions, but has the 'obligation' to conform to the postconditions contracted.

The Eiffel programming language[2] was one of the first to provide built-in support for contract programming, allowing the specification of pre and post-conditions, as well as invariants, enforcing these assertions at runtime to check program consistency. A precondition violation usually means a problem in the API caller, as opposed to a postcondition violation, interpreted as a defect in the callee. Besides Eiffel, only a few languages support contracts natively, e.g. Spec#[3]. However, there are several implementations of contract libraries for a number of languages. For instance, the Java Modeling Language (JML)[4] provides formal specification of Java interfaces, and the Code Contracts library[5] extends the C# language.

Beugnard et al. (97) proposed a classification of contracts for software components in four levels: syntactic, behavioral, synchronization and quality of service. According to the authors, these levels correspond to an increasing ability to dynamically negotiate each level's features. Level 1 starts with basic syntactic features (nonnegotiable), going up to the quantitative properties of quality of service features in level 4 (dynamically negotiable). Table 6.1 summarizes the contract levels and their meaning.

As illustrated above, the usual notion of contract promoted by Meyer and others corresponds to level 2 in the classification, which accounts for the semantic specification of the behavior of a software component or object. Level 1 corresponds to the basic requirements that should be satisfied to call an

[2] `https://www.eiffel.com/`
[3] `http://research.microsoft.com/en-us/projects/specsharp/`
[4] `http://www.jmlspecs.org`
[5] `http://research.microsoft.com/en-us/projects/contracts/`

Contract level	**Description**
Level 1: Syntactic	specification of an operation's syntax, parameters and returned values (interface definition languages and usual programming languages)
Level 2: Behavioral	specification of pre and postconditions and invariants (Eiffel, OCL)
Level 3: Synchronization	specification of synchronization conditions under concurrency (mutex, path expression)
Level 4: Quality of service	specification of efficiency-related parameters (response time, precision, throughput)

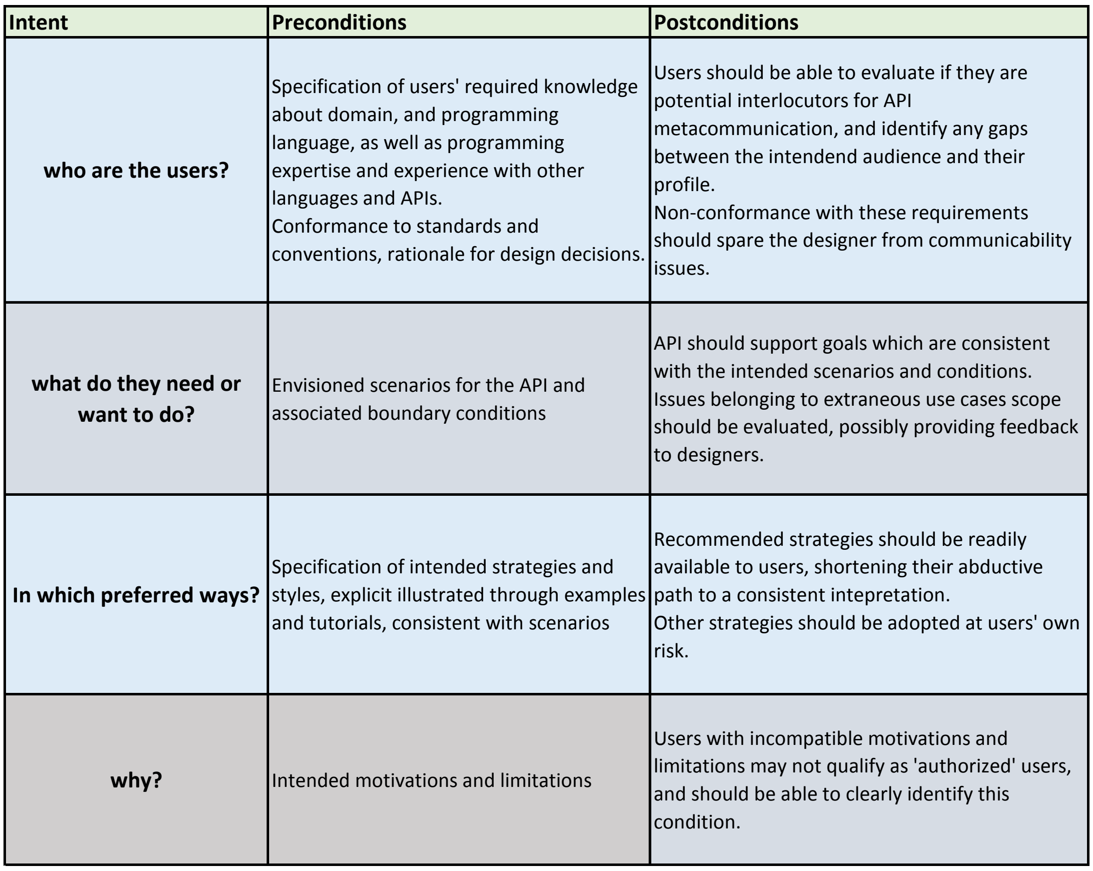
Table 6.1: Classification of contract levels

operation - use the proper syntax. Level 3 specifies the concurrency conditions that should be met to guarantee the operation's consistency, while level 4 is associated with efficiency aspects.

The 'traditional' concept of software contract (level 2 - behavioral) has been the object of many research studies since the 1990's (98) as a mechanism to serve a twofold purpose. First, it provides a formal specification of software behavior that can be enforced at runtime (and statically checked in some cases), contributing to its robustness and correctness. Second, it serves as documentation of an API's semantics, at least in terms of a partial specification of program state before and after an operation is carried out. The formal nature of contract languages offers the advantage that it allows for its automatic verification by the program runtime or the compiler. However, it limits the contracts' expressiveness and usability, since it is very difficult to specify complex conditions using a formal language, sometimes even more complex than the API implementation itself, which may be one of the factors that limited the widespread use of behavioral contracts. In addition, it is unfeasible to specify formally all the conditions, abilities, assumptions, modes and circumstances that surround the designer's intent encoded in a software artifact.

Viewing APIs as designer-to-user metacommunication allows us to interpret this communication process as a 'pragmatic contract' between designer and user, extending the concept of behavioral specification to a more abstract encoding of the *intent* behind the software artifact, as paraphrased in the metacommunication template. This perspective provides an analogy with the usual concept of contracts that describes an agreement between people, as opposed to the notion of a contract between pieces of code (caller and callee). This analogy provides a type of contract which should be reflected in designers' metacommunication to convey their design vision to users: who are their intended interlocutors, what are their goals, preferences, needs, motivations, and so on. By specifying the 'pragmatic contract' behind the design of APIs, software producers may contribute to prevent communicability issues from happening. Underspecification of software artifacts' intent aspects and their misinterpretation are common sources of breakdowns to API users, as evidenced by the results of the empirical studies detailed in chapter 5.

Figure 6.1 illustrates the interpretation of preconditions of a pragmatic contract as the specification of intent, and postconditions as the possible outcomes of satisfying (or not) the preconditions. Preconditions can be interpreted as the specification of intent behind the design of an API: the 'authorized' users, goals, strategies, and so on. They summarize the designers' 'rights', which means that what falls outside this specification does not imply an 'obligation' from the API designer's side, which are described by the postconditions.



Intent	Preconditions	Postconditions
who are the users?	Specification of users' required knowledge about domain, and programming language, as well as programming expertise and experience with other languages and APIs. Conformance to standards and conventions, rationale for design decisions.	Users should be able to evaluate if they are potential interlocutors for API metacommunication, and identify any gaps between the intendend audience and their profile. Non-conformance with these requirements should spare the designer from communicability issues.
what do they need or want to do?	Envisioned scenarios for the API and associated boundary conditions	API should support goals which are consistent with the intended scenarios and conditions. Issues belonging to extraneous use cases scope should be evaluated, possibly providing feedback to designers.
In which preferred ways?	Specification of intended strategies and styles, explicit illustrated through examples and tutorials, consistent with scenarios	Recommended strategies should be readily available to users, shortening their abductive path to a consistent intepretation. Other strategies should be adopted at users' own risk.
why?	Intended motivations and limitations	Users with incompatible motivations and limitations may not qualify as 'authorized' users, and should be able to clearly identify this condition.

Figure 6.1: Intent as a pragmatic contract

Regarding the preconditions illustrated in figure 6.1, envisioned scenarios and examples of 'authorized' API use should not be taken as restrictions to limit unforeseen use cases of the API. Conversely, a good API usually promotes 'unpredicted' combinations of its features, since attributes like orthogonality and flexibility allow these combinations. Therefore, the goal of examples is to illustrate the API's main use cases, uncovering common application scenarios and communicating to its users the designer's intent behind the artifact, or at least part of it.

Still concerning the analogy of metacommunication as a pragmatic contract, we can also describe its expected effects in terms of speech act theory (see 3.1.5). There are two classes of speech acts that help to characterize the contract conditions: 'directive' and 'commissive' acts. *Directive* speech acts aim at causing the hearer to do something, and *commissive* speech acts commit the speaker to taking some particular course of action in the future. Therefore, preconditions may be viewed as a directive speech act: it tells the 'hearer' (user) what to do in order to be eligible to use the artifact's features; conversely, as long as the user conforms to the intent specification, the 'speaker' (designer) is committed to deliver API features consistently with its pragmatic specification.

The 'directive' analogy may also be interpreted in the sense that the interface itself is a 'designer-to-user directive', which tells the user to 'do like this to use it'. This creates the reverse concept of 'user-to-API directive': by setting API parameters and calling its operations, user 'directs' the 'designer's deputy' to act as determined. As such, the API, as the designer's deputy, has to satisfy its part of the contract at runtime, and the user has to commit to the contract by using only valid 'directives'. Therefore, API use may be viewed as a 'commissive act' from users, even if they are not aware of it. This perspective illustrates the inherent complexities of this form of communication.

In conclusion, the basic idea behind this analogy of a pragmatic contract is to call the attention to a relevant aspect of API usage that is commonly underspecified: its intent dimension, which is mostly composed by tacit knowledge that frequently goes unnoticed or missed in software specification and use. By promoting this reflection, the goal is to highlight the fact that there is more to the mutual understanding between API producers and consumers than normally specified in the artifact's design.

6.3 Contributions

This section summarizes the main contributions of this research as a scientific work. Contributions are discussed with respect to API evaluation, API design, and Semiotic Engineering.

As previously mentioned in chapter 1, APIs play a key role in the current scenario of Software Engineering. However, the investigation, practical use and professional teaching of APIs has a tradition of focusing on language and programming aspects, not considering the API users' needs, goals, abilities and contextual differences as first-class objects of study. Evidence collected and analyzed in this research contributes to illustrate potential consequences of leaving these aspects out of scope when discussing API design and evaluation.

The communicative approach introduced in this thesis contributes to the study of APIs with a shift of perspective that takes it into a novel context. An immediate consequence of adopting the theoretical guidance of Semiotic Engineering is to include the designer in the ontology supporting this type of study. Framing API investigation as a communication process between designers and users provides new insights and deeper comprehension of a 'known problem' through the reinterpretation of its various circumstances, including some frequently missed ones.

The approach introduced in this thesis is inspired by an HCI view of programmers as users of APIs, which are intellectual artifacts encoded in a linguistic form and carrying a wide range of meanings, making its production and use a complex human activity. In order to analyze human aspects surrounding this phenomenon, this research has applied HCI concepts and methods to investigate programmers' experiences and difficulties with APIs, in particular Java and PHP APIs. These difficulties have been mapped and translated from bug reports that potentially reveal communicative issues by being labeled as 'not an issue'. When arguing against a presumed 'bug', API designers (or evaluators, playing the role of designers) express the intent behind the construction of the software artifact, revealing frequently overlooked and unnoticed aspects of API metacommunication.

The proposed approach consisted of a systematic collection and analysis of communicative and cognitive aspects contained in a sample of bug reports selected from a large dataset. One of the consequences of this approach is the conclusion that there is more to the study of APIs than the current 'code-oriented' view of the subject, since framing this problem as human communication adds new facets to its investigation. This reinforces the relevance of a 'holistic' view of APIs, comprehending its specification in formal languages, documentation, tutorials, examples, scenarios, runtime behavior, error conditions, limitations, and so on, in a coordinated articulation of static, dynamic and metalinguistic signs. A consistent 'discourse' in terms of metacommunication elements may contribute to reduce the communicative and cognitive impact on eligible API users, providing a more productive and effective programming experience.

6.3.1 Contributions to API evaluation and design

As explained in chapter 5, the iterative nature of the research resulted in the gradual construction of a conceptual framework based on semiotic and cognitive theories. This framework has been used to analyze the bug reports selected as empirical evidence of communicative issues in APIs, contributing to representative findings described in section 5.3.

The goal of the studies performed was not to find all classes of problems that occur involving Java and PHP APIs, but rather to identify and explain new perspectives on some representative issues. Even with respect to well known issues, their analysis from a different standpoint provides a novel diagnostic that enriches their discussion and increases designers' awareness about possible outcomes of certain decisions.

In the current state of the research development, there is no evidence to ensure that the evaluation framework could be used 'as-is' in the analysis of all kinds of APIs. It is quite reasonable to expect that the proposed approach would provide useful results if applied to APIs with similar characteristics and belonging to the same programming paradigms as the APIs studied in this research. However, more research work is needed to confirm the suitability of using the conceptual tools introduced to evaluate other sorts of APIs, and should be addressed by future work.

The conceptual framework to analyze APIs frames the object of study as human communication mediated by software, in accordance with Semiotic Engineering. This theory views every interactive software interface as a message from its producers to its consumers, conveying the producers' vision about consumers, among other things. Therefore, part of the results obtained in this research work are also entitled to inform API design.

However, in the current state of the research we cannot determine precisely the quantity, quality, relevance and mode of the results' influence on API designers to improve the process of API construction and its products.

The potential benefits of applying the proposed conceptual framework as an epistemic tool for API designers are quite plausible, but still lacks verification. This conjecture is supported by de Souza's description of an epistemic tool in (22), p.33: "*An epistemic tool is one that is not used to yield directly the answer to the problem, but to increase the problem-solver's understanding of the problem itself and the implications it brings about. (...) epistemic design tools are those that will not necessarily address the problem solution, but the problem's space and nature, and the constraints on candidate solutions for it*".

Therefore, the epistemic nature of the results of this research may help API designers to develop different perspectives on their object of study. In addition, viewing design intent as a first-class object of study provides new awareness to API producers about the organization of the design space and about their own role in the process of software construction. In this context, a communication perspective provides a proper ontology and vocabulary to reason about intent in API design, as shown in chapter 5.

In addition, a communicative approach to the API design problem allows us to characterize it as an effort from designers to find a balance in their communication with two simultaneous interlocutors: the mechanical (computer) and the human (API user). This thesis addresses only the human interlocutor's requirements in the context of API design, and results show that non-conformance with these conditions may affect negatively this interlocutor's reception of the designer's message.

It is a generally observed principle that when a design prioritizes the computer's performance requirements, the 'human interlocutor' efficiency may be hindered, and vice-versa. This trade-off in the attempt to satisfy both interlocutors optimality criteria is usually a non-trivial challenge, and a difficult question is to determine which of them should be served with higher priority. In order to effectively support the simultaneous requirements of the mechanical and human interlocutors, there is a need to find methods to measure and compare the costs and benefits associated with each interlocutor. This comparability criteria between these interlocutor's needs has yet to be uncovered.

The mechanical interlocutor's optimality criteria is usually characterized in terms of algorithmic efficiency, while the human interlocutor needs to be supported by criteria that encompass her cognitive and semiotic abilities and limitations. Other types of criteria may apply to both cases, but these are probably the most commonly used.

The differences between these interlocutors' requirements can be illustrated in terms of the theoretical foundations of computing. Algorithmic efficiency can be expressed by symbolic processing and manipulation represented by formal languages and finite automata. The optimization of state machines usually involves eliminating redundant and unnecessary states, whose primary function was to support human cognition. The optimized symbolic representation is frequently different from an equivalent human-generated structure, and this is a recurrent situation in different levels of abstraction in computing.

Peircean semiotic theory (76), which provided the foundations for the development of Semiotic Engineering, states that the relation between the components of a sign and its interpreter are determined by a process called *semiosis*, which establishes the meanings of a sign in its interpreter's 'mind'. In the case of a human interpreter, the boundaries of this process cannot be predicted deterministically because it is impossible do know exactly when or why someone will stop or resume inferencing about a particular sign. The interpretation may proceed indefinitely, in theory, and this is known as 'unlimited semiosis'. In practice, meaning is achieved by the influence of educational and cultural aspects that may stabilize this process when a plausible hypothesis is formulated ('abductive reasoning' – see 3.1.2). Regarding the 'mechanical mind', its semiosic process is bounded and deterministic, since it is defined by deductive reasoning based on pre-established rules. Despite these differences, Peircean theory does not restrict the type of 'mind' to which it applies, be it natural or artificial. Therefore, a semiotic characterization of these interlocutors' abilities and requirements has the *potential* to provide comparability criteria between these apparently irreconcilable 'minds'.

This thesis provides an initial approach to study the human interlocutor's optimality criteria, but does not address the perspective of the mechanical interlocutor. Despite this limitation, the need for this comparability and the conjecture of a semiotic approach to the problem is a theoretical result from this study, which requires further research to investigate its feasibility and implications, a topic for future work.

6.3.2 Contributions to Semiotic Engineering

The theoretical guidance of Semiotic Engineering permeated all stages of this research work, providing conceptual, ontological, methodological and epistemic support. However, Semiotic Engineering research methods have been initially designed to analyze interactive visual interfaces. Therefore, the methods and the framework used in this research derive from the adaptation and combination of Semiotic Engineering methods' concepts and tools to the context of API evaluation and design. This adaptation provided novel interpretations to these concepts and tools, as well as new insights about their applicability. For this reason, the analytical procedures introduced in this thesis can be regarded as a secondary contribution of the research.

In addition, this research amplifies the scope to which the Semiotic Engineering theory can be applied, since it has been traditionally used in HCI contexts. The results detailed in chapter 5 demonstrate that the theory provided solid support to the study of APIs as human communication mediated by software. Despite the adaptations previously mentioned, no major impedance has been identified between the theory's elements and its new context of application's requirements.

6.4 Limitations

In addition to the limitations described in previous sections, the fact that the studies in this research involved only APIs from two programming languages is a limitation in the scope of the approach. However, as oriented by the research question, the main goal was to comprehend how communicative and cognitive aspects of APIs influence the use and interpretation of their meanings. This was the first research work to provide this perspective on APIs, to the best of our knowledge. As such, it called for a qualitative approach with a deep and focused analysis of the object of study, in order to unveil its main aspects and also to shape the conceptual tools used in this investigation. For these reasons, the scope has been limited to only two programming languages.

As previously mentioned, the lack of validation of the research results as an epistemic tool to inform API design is also an important limitation of the thesis. However, this type of validation would involve activities that comprehend at least the following steps: 1) to construct an API with a minimum complexity and usefulness; 2) to carry out a systematic evaluation of its design and implementation; 3) to evaluate its effectiveness in a realistic setting; 4) to contrast its results with a ‘control’ API, constructed without the orientation provided by the framework. Due to the effort needed to implement the research for this validation, it was out of the scope of this thesis, and is the subject of future work.

The results from the API studies described in this thesis, due to their qualitative nature, are inherently descriptive and non-predictive. As such, these results do not provide heuristics, guidelines or general rules for API design, which may be taken as a limitation of the research work. However, these results provide a detailed account of *what happens* with respect to API use in practice. These findings provide the basis for the formulation of novel research questions and hypothesis, to be further investigated by quantitative methods and possibly generate predictive results for what will (or will not) happen.

6.5 Future work

The current research approach derives from studies involving issues from the Java and PHP APIs. As such, it should be expanded to other types of APIs, domains, programming languages and paradigms, in order to better determine its suitability to these different contexts. This type of study may also provide new insights and results to be incorporated in the framework.

This thesis' results can be used to revisit the object of study from related work on API usability, as an exercise to provide new perspectives to the knowledge acquired in these studies. This is an interesting type of study to perform, as it has the potential to further develop the conceptual tools proposed in this work, and also provide new findings on top of the original study's results.

As mentioned earlier in this chapter, there is a need to perform detailed studies to determine precisely if and how well the epistemic framework may inform API design. This would involve the systematic development and evaluation of APIs in production environments and real software projects, in order to effectively account for the benefits of the epistemic tools. A possible extension of this work would be the creation of metrics to quantify the improvement (or not) of the resulting APIs.

The empirical studies we carried out involved the selection of bug reports from a database of thousands of elements. In many cases, the discarded bug report consisted in our judgment of issues that clearly revealed the user's lack of knowledge in programming concepts or some other computer science topic. Despite the large number of bug reports discarded in this condition, many of these bugs may be useful to help in the investigation of educational issues, since there is a significant recurrence of some types of basic knowledge issues in the reports. For instance, results from the analysis of bug reports indicate issues related to floating point representation and other programming concepts like references, object mutability and concurrency.

Due to the current state of the research results and the need to validate precisely how it informs API design, the conceptual framework needs further refinement to enable its effective use in a more technical context, as opposed to its scientific use. This also motivates the development of evaluation methods based on the communicative approach to be used by technical specialists.

6.6 Final considerations

The characterization of an API as a 'shortcut' is a good metaphor to describe the fact that it enables programmers to have a 'quick access' to different types of abstractions and services that shortens their path to an existing goal, at least in the successful case.

API producers' motivations and objectives may vary greatly. Many programmers simply write API's and programs to themselves, and this is a perfectly valid use of APIs (99). Still, the communicative approach to APIs holds even in this peculiar application of the intent dimension, in a trivial instantiation of the metacommunication template: “I know *exactly* who you are, and also your wants and needs.”

However, technology producers in enterprise or academic settings that envision a wide adoption of their APIs and other software artifacts have a more complex problem to deal with. If their motivation includes a more effective programming experience, communicability should be a concern in the design of these artifacts.

This thesis does not intend to be prescriptive about what should be done with respect to API design. Rather, it provides a novel perspective on a complex problem which permeates software development. This approach highlights the fact that the use of APIs involves a ‘conversation’ between (at least) two people with potentially different cultures, experiences, education, values, needs (designer included). As such, to effectively address these differences, a communicative approach has the potential to bring the pragmatic aspects of software development to this discussion. This shift in perspective is a contribution from this research, since it allows us to view things in different ways from which they were perceived before. For this reason, a new view on API programming has the potential to contribute to advance current knowledge in relevant directions. The conceptual framework that results from this research increases designers’ awareness of the need to reconcile multiple interlocutors’ perspectives of an API. It also provides epistemic support for the design activity by offering an organization of the design space and the relations among its components.

In conclusion, this chapter ends by quoting three excerpts (dated, but still relevant) from renowned computer scientists, in which they expose their positions about some human aspects of programming, much in the same spirit as the motivation for this research work.

“The most obvious failing of traditional notations (...) is that too little attention is paid to readability. It is sometimes brusquely asserted that ‘real programmers’ can get used to anything, given a little ‘syntactic sugar’, and that worrying about readability is namby-pamby stuff. It is indeed true that some programmers can cope with very demanding notations, and it is also true that some chess grandmasters can play twenty games blindfold simultaneously. A conscious design decision to cater only for such prodigies could not be faulted, but it would be stupid for a notation designer to ignore readability in a notation designed for widespread acceptance.” (Thomas Green, *‘Cognitive Dimensions of Notations’* (27))

“Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a

computer to do." (Donald Knuth, *'Literate Programming'* (100)).

"Perhaps our troubles with programming languages stem from the unidirectionality of the communication. Only when the theoreticians turn to the 'dialogue' aspects of programming 'language' will they finally be forced to recognize that they are not students of symbol manipulation, but of human behavior." (Gerald Weinberg, *'The Psychology of Computer Programming'* (101))

Bibliography

[1] KRAMER, J. Is abstraction the key to computing? **Commun. ACM**, ACM, New York, NY, USA, v. 50, n. 4, p. 36–42, apr. 2007.

[2] WING, J. M. Computational thinking and thinking about computing. **Philosophical transactions. Series A, Mathematical, physical, and engineering sciences**, v. 366, n. 1881, p. 3717–3725, oct. 2008.

[3] HANENBERG, S. Faith, hope, and love: An essay on software science's neglect of human factors. In: **Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications**. New York, NY, USA: ACM, 2010. (OOPSLA '10), p. 933–946.

[4] SOUZA, C. R. B. de et al. Sometimes you need to see through walls: a field study of application programming interfaces. In: **Proceedings of the 2004 ACM conference on Computer supported cooperative work**. New York, NY, USA: ACM, 2004. (CSCW '04), p. 63–71.

[5] HENNING, M. The rise and fall of CORBA. **Commun. ACM**, ACM, New York, NY, USA, v. 51, n. 8, p. 52–57, aug. 2008.

[6] HENNING, M. API design matters. **Queue**, ACM, New York, NY, USA, v. 5, n. 4, p. 24–36, may 2007.

[7] BLOCH, J. How to design a good API and why it matters. In: **Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications**. New York, NY, USA: ACM, 2006. (OOPSLA '06), p. 506–507.

[8] TULACH, J. **Practical API Design: Confessions of a Java Framework Architect**. 1. ed. Berkely, CA, USA: Apress, 2008.

[9] CWALINA, K.; ABRAMS, B. **Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries**. 2nd. ed. [S.l.]: Addison-Wesley Professional, 2008.

[10] IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; CELES, W. Passing a language through the eye of a needle. **Queue**, ACM, New York, NY, USA, v. 9, n. 5, p. 20:20–20:29, may 2011.

[11] MYERS, B.; KO, A. The past, present and future of programming in HCI. Institute for Software Research, Carnegie Mellon University, 2009. Available from Internet: <`http://repository.cmu.edu/isr/782/`>.

[12] ARNOLD, K. Programmers are people, too. **Queue**, ACM, New York, NY, USA, v. 3, n. 5, p. 54–59, jun. 2005.

[13] DAUGHTRY, J. M. et al. API usability: Report on special interest group at CHI. **SIGSOFT Softw. Eng. Notes**, ACM, New York, NY, USA, v. 34, n. 4, p. 27–29, jul. 2009.

[14] CLARKE, S.; BECKER, C. Using the Cognitive Dimensions Framework to evaluate the usability of a class library. In: PETRE, M.; BUDGEN, B. (Ed.). **Proc. Joint Conf. EASE & PPIG**. Keele, UK: Keele University, 2003. p. 359–366.

[15] STYLOS, J.; CLARKE, S. Usability implications of requiring parameters in objects' constructors. In: **Proceedings of the 29th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (ICSE '07), p. 529–539.

[16] FAROOQ, U.; ZIRKLER, D. API peer reviews: a method for evaluating usability of application programming interfaces. In: **Proceedings of the 2010 ACM conference on Computer supported cooperative work**. New York, NY, USA: ACM, 2010. (CSCW '10), p. 207–210.

[17] ROBILLARD, M. P.; DELINE, R. A field study of API learning obstacles. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 16, n. 6, p. 703–732, dec. 2011.

[18] DAHOTRE, A. et al. Using intelligent tutors to enhance student learning of application programming interfaces. **J. Comput. Sci. Coll.**, Consortium for Computing Sciences in Colleges, USA, v. 27, n. 1, p. 195–201, oct. 2011.

[19] GRILL, T.; POLACEK, O.; TSCHELIGI, M. Methods towards API usability: A structural analysis of usability problem categories. In: WINCKLER, M.; FORBRIG, P.; BERNHAUPT, R. (Ed.). **Human-Centered Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7623). p. 164–180.

[20] HOVEMEYER, D. H. **Simple and effective static analysis to find bugs**. Thesis (PhD) — University of Maryland at College Park, College Park, MD, USA, 2005.

[21] GEORGIEV, M. et al. The most dangerous code in the world: validating SSL certificates in non-browser software. In: **Proceedings of the 2012 ACM Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2012. (CCS '12), p. 38–49.

[22] SOUZA, C. S. D. **The Semiotic Engineering of Human-Computer Interaction**. Cambridge, MA: The MIT Press, 2005.

[23] AFONSO, L.; CERQUEIRA, R.; SOUZA, C. de. Evaluating application programming interfaces as communication artefacts. In: **Proceedings of the Psychology of Programming Interest Group Annual Conference 2012 (PPIG'2012)**. London, UK: The Psychology of Programming Interest Group, 2012. p. 151–162.

[24] FERREIRA, J. et al. Combining cognitive, semiotic and discourse analysis to explore the power of notations in visual programming. In: **Proceedings of VL-HCC'2012 – IEEE Symposium on Visual Languages and Human-Centric Computing**. Innsbruck, Austria: IEEE, 2012. p. 101–108.

[25] FERREIRA, J.; SOUZA, C. de; CERQUEIRA, R. Characterizing the tool-notation-people triplet in software modeling tasks. In: **Proceedings of 13th Brazilian Symposium on Human Factors in Computer Systems**. Porto Alegre, RS: SBC, 2014. p. 31–40.

[26] SOUZA, C. D.; LEITÃO, C. **Semiotic Engineering Methods for Scientific Research in HCI**. Princeton, NJ: Morgan and Claypool Publishers, 2009.

[27] GREEN, T. R. G. Cognitive dimensions of notations. In: SUTCLIFFE, A.; MACAULAY, L. (Ed.). **People and Computers V**. Cambridge, UK: Cambridge University Press, 1989. p. 443–460.

[28] SHEIL, B. A. The psychological study of programming. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 13, n. 1, p. 101–120, mar. 1981.

[29] ROSSON, M. B.; CARROLL, J. M. The reuse of uses in smalltalk programming. **ACM Trans. Comput.-Hum. Interact.**, ACM, New York, NY, USA, v. 3, n. 3, p. 219–253, sep. 1996.

[30] MCLELLAN, S. G. et al. Building more usable APIs. **IEEE Softw.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 15, n. 3, p. 78–86, may 1998.

[31] SOLOWAY, E.; EHRLICH, K. Empirical studies of programming knowledge. **Software Engineering, IEEE Transactions on**, SE-10, n. 5, p. 595–609, sep. 1984.

[32] CLARKE, S. Evaluating a new programming language. In: KADODA, G. (Ed.). **Proceedings of the 13th Workshop of the Psychology of Programming Interest Group**. Bournemouth, UK: Bournemouth University, 2001. p. 275–289.

[33] CLARKE, S. Measuring API usability. **Dr. Dobb's Journal**, v. 29, p. S6–S9, 2004. Available from Internet: <`http://www.drdobbs.com/windows/measuring-api-usability/184405654`>.

[34] CLARKE, S. Describing and measuring API usability with the Cognitive Dimensions. In: **Cognitive Dimensions of Notations 10th Anniversary Workshop**. [s.n.], 2006. Available from Internet: <`http://www.cl.cam.ac.uk/~{}afb21/CognitiveDimensions/workshop2005/index.html`>.

[35] MAIA, R. et al. A qualitative human-centric evaluation of flexibility in middleware implementations. **Empirical Software Engineering**, Springer Netherlands, v. 17, p. 166–199, 2012.

[36] BORE, C.; BORE, S. Profiling software API usability for consumer electronics. In: **Consumer Electronics, 2005. ICCE. 2005 Digest of Technical Papers. International Conference on**. [S.l.: s.n.], 2005. p. 155–156.

[37] RATIU, D.; JURJENS, J. Evaluating the reference and representation of domain concepts in APIs. In: **Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on**. Amsterdam, NL: IEEE, 2008. p. 242–247.

[38] WATSON, R. Improving software API usability through text analysis: A case study. In: **Professional Communication Conference, 2009. IPCC 2009. IEEE International**. Waikiki, HI: IEEE, 2009. p. 1–7.

[39] O'CALLAGHAN, P. The API walkthrough method: A lightweight method for getting early feedback about an API. In: **Evaluation and**

Usability of Programming Languages and Tools. New York, NY, USA: ACM, 2010. (PLATEAU '10), p. 5:1–5:6.

[40] GERKEN, J. et al. The concept maps method as a tool to evaluate the usability of APIs. In: **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2011. (CHI '11), p. 3373–3382.

[41] FAROOQ, U.; WELICKI, L.; ZIRKLER, D. API usability peer reviews: A method for evaluating the usability of application programming interfaces. In: **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2010. (CHI '10), p. 2327–2336.

[42] ELLIS, B.; STYLOS, J.; MYERS, B. The factory pattern in API design: A usability evaluation. In: **Proceedings of the 29th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (ICSE '07), p. 302–312.

[43] STYLOS, J.; MYERS, B. A. The implications of method placement on API learnability. In: **Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2008. (SIGSOFT '08/FSE-16), p. 105–112.

[44] PICCIONI, M.; FURIA, C.; MEYER, B. An empirical study of API usability. In: **Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on**. Baltimore, MD: IEEE, 2013. p. 5–14.

[45] SPIZA, S.; HANENBERG, S. Type names without static type checking already improve the usability of APIs (as long as the type names are correct): An empirical study. In: **Proceedings of the 13th International Conference on Modularity**. New York, NY, USA: ACM, 2014. (MODULARITY '14), p. 99–108.

[46] DOUCETTE, A. On API usability: An analysis and an evaluation tool. **CMPT816 - Software Engineering, Saskatoon, Saskatchewan, Canada: University of Saskatchewan**, Saskatchewan, Canada: University of Saskatchewan, 2008.

[47] SOUZA, C. de; BENTOLILA, D. Automatic evaluation of API usability using complexity metrics and visualizations. In: **Proceedings of the**

31st International Conference on Software Engineering (ICSE 2009) – Companion Volume. Vancouver, BC: IEEE, 2009. p. 299–302.

[48] SCHELLER, T.; KUHN, E. Measurable concepts for the usability of software components. In: **Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on**. [S.l.]: IEEE, 2011. p. 129–133.

[49] CATALDO, M. et al. The impact of interface complexity on failures: An empirical analysis and implications for tool design. **Technical Report CMU-ISR-10-101, School of Computer Science, Carnegie Mellon University**, 2010.

[50] RAMA, G. M.; KAK, A. Some structural measures of API usability. **Software: Practice and Experience**, John Wiley & Sons, Ltd., v. 45, n. 1, p. 75–110, 2015.

[51] STYLOS, J.; MYERS, B. Mapping the space of API design decisions. In: **Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing**. Washington, DC, USA: IEEE Computer Society, 2007. (VLHCC '07), p. 50–60.

[52] ZIBRAN, M.; EISHITA, F.; ROY, C. Useful, but usable? factors affecting the usability of APIs. In: **Reverse Engineering (WCRE), 2011 18th Working Conference on**. Limerick, Ireland: IEEE, 2011. p. 151 –155.

[53] MEYER, B. Lessons from the design of the Eiffel libraries. **Commun. ACM**, ACM, New York, NY, USA, v. 33, n. 9, p. 68–88, sep. 1990.

[54] BLOCH, J. **Effective Java (2nd Edition) (The Java Series)**. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[55] ROBILLARD, M. P. What makes APIs hard to learn? answers from developers. **IEEE Softw.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 26, n. 6, p. 27–34, nov. 2009.

[56] HOU, D.; LI, L. Obstacles in using frameworks and APIs: An exploratory study of programmers' newsgroup discussions. In: **Program Comprehension (ICPC), 2011 IEEE 19th International Conference on**. Kingston, ON: IEEE, 2011. p. 91–100.

[57] SILLITO, J.; MURPHY, G. C.; VOLDER, K. D. Asking and answering questions during a programming change task. **IEEE Trans. Softw.**

Eng., IEEE Press, Piscataway, NJ, USA, v. 34, n. 4, p. 434–451, jul. 2008.

[58] DUALA-EKOKO, E.; ROBILLARD, M. P. Asking and answering questions about unfamiliar APIs: An exploratory study. In: **Proceedings of the 34th International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 266–276.

[59] KUHN, A.; DELINE, R. On designing better tools for learning APIs. In: **Search-Driven Development - Users, Infrastructure, Tools and Evaluation (SUITE), 2012 ICSE Workshop on**. Zurich, Switzerland: IEEE, 2012. p. 27–30.

[60] DEKEL, U.; HERBSLEB, J. D. Improving API documentation usability with knowledge pushing. In: **Proceedings of the 31st International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. (ICSE '09), p. 320–330.

[61] STYLOS, J. et al. Improving API documentation using API usage information. In: **Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)**. Washington, DC, USA: IEEE Computer Society, 2009. (VLHCC '09), p. 119–126.

[62] KO, A.; RICHE, Y. The role of conceptual knowledge in API usability. In: **Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on**. Pittsburgh, PA: IEEE, 2011. p. 173–176.

[63] MONPERRUS, M. et al. What should developers be aware of? an empirical study on the directives of API documentation. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 17, n. 6, p. 703–737, dec. 2012.

[64] MAALEJ, W.; ROBILLARD, M. Patterns of knowledge in API reference documentation. **Software Engineering, IEEE Transactions on**, v. 39, n. 9, p. 1264–1282, Sept 2013.

[65] ENDRIKAT, S. et al. How do API documentation and static typing affect API usability? In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: ACM, 2014. (ICSE 2014), p. 632–642.

[66] BLACKWELL, A. F. Metaphors we Program By: Space, Action and Society in Java. In: **Proceedings of the 18th Psychology of Programming Interest Group (PPIG 2006)**. Brighton, UK: University of Sussex, 2006.

[67] DUBOCHET, G. Computer code as a medium for human communication : Are programming languages improving ? **Proceedings of the 21st Psychology of Programming Workshop (PPIG 2009)**, University of Limerick, Ireland, p. 174–187, 2009.

[68] ORCHARD, D. The four Rs of programming language design. In: **Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software**. New York, NY, USA: ACM, 2011. (ONWARD '11), p. 157–162.

[69] ZEMANEK, H. Semiotics and programming languages. **Commun. ACM**, ACM, New York, NY, USA, v. 9, n. 3, p. 139–143, mar. 1966.

[70] KAMTHAN, P. A framework for the pragmatic quality of Z specifications. **International Journal of Software Engineering and Knowledge Engineering**, v. 16, n. 5, p. 769–790, 2006.

[71] TANAKA-ISHII, K. **Semiotics of Programming**. 1st. ed. New York, NY, USA: Cambridge University Press, 2010.

[72] SOUZA, C. R. de; REDMILES, D. On the roles of APIs in the coordination of collaborative software development. **Computer Supported Cooperative Work (CSCW)**, Springer Netherlands, v. 18, n. 5-6, p. 445–475, 2009.

[73] BURNS, C. et al. Usable results from the field of API usability: A systematic mapping and further analysis. In: **Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on**. Innsbruck, Austria: IEEE, 2012. p. 179–182.

[74] STEFIK, A. et al. What is the foundation of evidence of human factors decisions in language design? an empirical study on programming language workshops. In: **Proceedings of the 22Nd International Conference on Program Comprehension**. New York, NY, USA: ACM, 2014. (ICPC 2014), p. 223–231.

[75] WURSTER, G.; OORSCHOT, P. C. van. The developer is the enemy. In: **Proceedings of the 2008 Workshop on New Security Paradigms**. New York, NY, USA: ACM, 2008. (NSPW '08), p. 89–97.

[76] PEIRCE, C. S. **Collected Papers of Charles Sanders Peirce**. Cambridge, MA, USA: Harvard University Press, 1931–1958. (Collected Papers of Charles Sanders Peirce).

[77] SAUSSURE, F. de. **Cours de linguistique générale**. Paris: Payot, 1972.

[78] JAKOBSON, R. Linguistics and Poetics. In: SEBEOK, T. A. (Ed.). **Style in Language**. Cambridge, MA: The M.I.T. Press, 1960.

[79] AUSTIN, J. **How to Do Things with Words**. Cambridge, MA: Harvard University Press, 1962.

[80] SEARLE, J. R. **Speech Acts: An Essay in the Philosophy of Language**. Cambridge, London: Cambridge University Press, 1969.

[81] GRICE, H. P. Logic and conversation. In: COLE, P.; MORGAN, J. L. (Ed.). **Speech Acts**. New York: Academic Press, 1975, (Syntax and Semantics, v. 3). p. 41–58.

[82] PRATES, R. O.; SOUZA, C. S. de; BARBOSA, S. D. J. Methods and tools: A method for evaluating the communicability of user interfaces. **interactions**, ACM, New York, NY, USA, v. 7, n. 1, p. 31–38, jan. 2000.

[83] SOUZA, C. S. de et al. Can inspection methods generate valid new knowledge in HCI? the case of semiotic inspection. **Int. J. Hum.-Comput. Stud.**, Academic Press, Inc., Duluth, MN, USA, v. 68, n. 1-2, p. 22–40, jan. 2010.

[84] GREEN, T.; PETRE, M. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. **Journal of Visual Languages & Computing**, v. 7, n. 2, p. 131 – 174, 1996.

[85] CHURCH, L.; GREEN, T. Cognitive Dimensions - a short tutorial. In: **Proceedings of 20th Psychology of Programming Interest Group (PPIG 2008)**. Lancaster, UK: Lancaster University, 2008.

[86] PETRE, M. Cognitive dimensions 'beyond the notation'. **Journal of Visual Languages & Computing**, Elsevier, v. 17, n. 4, p. 292 – 301, 2006.

[87] CRESWELL, J. W. **Research Design: Qualitative, Quantitative, and Mixed Methods Approaches**. 3. ed. [S.l.]: Sage Publications, 2009.

[88] BRANDÃO, R. R. d. M. **A Capture & Access technology to support documentation and tracking of qualitative research applied to HCI**. Thesis (PhD) — Departamento de Informática, PUC-Rio, 2015. Advisor: Clarisse Sieckenius de Souza.

[89] CARROLL, J. M.; ROSSON, M. B. Interfacing thought: Cognitive aspects of human-computer interaction. In: CARROLL, J. M. (Ed.). Cambridge, MA, USA: MIT Press, 1987. chapter Paradox of the Active User, p. 80–111.

[90] SOUZA, C. S. de. Semiotics: and human-computer interaction. In: SOEGAARD, M.; DAM, R. F. (Ed.). **Encyclopedia of Human-Computer Interaction**. Aarhus, Denmark: The Interaction-Design.org Foundation, 2012. Available from Internet: <`http://www.interaction-design.org/encyclopedia/semiotics_and_human-computer_interaction.html`>.

[91] MAGNANI, L. An abductive theory of scientific reasoning. **Semiotica**, v. 153, p. 261–286, 2005.

[92] CLARKE, S. What is an End User Software Engineer? In: BURNETT, M. H. et al. (Ed.). **End-User Software Engineering**. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[93] GIBSON, J. J. The theory of affordances. In: SHAW, R.; BRANSFORD, J. (Ed.). **Perceiving, Acting, and Knowing: Toward an Ecological Psychology**. [S.l.]: Lawrence Erlbaum Associates, 1977. p. 67–82.

[94] NORMAN, D. A. **The Design of Everyday Things**. New York, NY, USA: Basic Books, Inc., 2002.

[95] SANTAELLA, L. Abduction: The logic of guessing. **Semiotica**, De Gruyter, v. 153, p. 175–198, 2005.

[96] MEYER, B. Applying design by contract. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 25, n. 10, p. 40–51, 1992.

[97] BEUGNARD, A. et al. Making components contract aware. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 32, n. 7, p. 38–45, jul. 1999.

[98] AFONSO, L. M. **Um Estudo Sobre Contratos em Sistemas de Componentes de Software**. Master's thesis (Master) — Departamento de Informática, PUC-Rio, sep. 2008. Advisor: Renato Fontoura de G. Cerqueira.

[99] GABRIEL, R. P. I throw itching powder at tulips. In: **Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software**. New York, NY, USA: ACM, 2014. (Onward! 2014), p. 301–319.

[100] KNUTH, D. E. Literate programming. **The Computer Journal**, v. 27, n. 2, p. 97–111, 1984.

[101] WEINBERG, G. M. **The Psychology of Computer Programming (Silver Anniversary Ed.)**. New York, NY, USA: Dorset House Publishing Co., Inc., 1998.