Pontifícia Universidade Católica
do Rio de Janeiro

João Antonio Dutra Marcondes Bastos

Promoting Conversational APIs: A Conceptual Framework and a Method for API Design

Tese de Doutorado

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática.

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
August 2020

Pontifícia Universidade Católica
do Rio de Janeiro

João Antonio Dutra Marcondes Bastos

Promoting Conversational APIs: A Conceptual Framework and a Method for API Design

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática. Approved by the Examination Committee.

Prof. Alessandro Fabricio Garcia
Advisor
Departamento de Informática – PUC-Rio

Prof. Alberto Barbosa Raposo
Departamento de Informática – PUC-Rio

Prof. Renato Fontoura de Gusmão Cerqueira
IBM Research Brazil – IBM

Prof. Gleison dos Santos Souza
Departamento de Informática Aplicada – UNIRIO

Prof. Rafael Maiani de Mello
Escola de Informática & Computação – CEFET-RJ

Rio de Janeiro, August 27th, 2020

João Antonio Dutra Marcondes Bastos

Graduated in computer science at Universidade Federal de Viçosa and Master in Informatics at Pontifícia Universidade Católica do Rio de Janeiro.

Bibliographic data

Bastos, João Antonio Dutra Marcondes

Promoting Conversational APIs: A Conceptual Framework and a Method for API Design / João Antonio Dutra Marcondes Bastos; advisor: Alessandro Fabricio Garcia. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2020.

v., 149 f: il. color. ; 30 cm

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

Inclui bibliografia

1. Informática – Teses. 2. Engenharia de software – Teses. 3. Interfaces de Programação de Aplicações;. 4. Método de Design;. 5. API Conversacional;. 6. Engenharia Semiótica.. I. Fabricio Garcia, Alessandro. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my wife and mother for their unconditional support

Acknowledgments

To my advisor Professor Alessandro Garcia for the stimulus and partnership for this work. Thank you for believing in my work and for accepting the difficult mission of concluding a doctoral work that was already underway.

To my first teacher counselor Clarisse de Souza and to my co-coordinator Luiz Afonso for all the help and guidance in the initial part of the Doctorate. By chance of fate we ended up not concluding the doctorate together, but your help was fundamental for my success. Thanks also to Professor Renato Cerqueira for the opportunity to carry out part of the research in the IBM Research Brazil laboratory.

To all friends and research partners of OPUS and SERG groups. Thank you for all your support throughout my Doctorate. Counting on such well aligned and tuned groups helped a lot on the path to individual success. Thank you very much to all of you who in various ways contributed to my work. In particular, my thanks to Daniel Tenorio and Rafael Maiani who participated in an active way dedicating much time to my Doctoral research. Thank you very much for your valuable comments, suggestions and reviews.

To my mother, Miriam, and to my wife Erica, without whom I would not have achieved success in this thesis. Thank you for all your support throughout this journey. To my in-laws, Alcemir and Dione for their support and affection over the past few years. To my brother Ernesto and his family. To my grandmother Zica and my aunts Marlene and Neuza. To all my friends and family who have supported me in this journey in various ways, thank you very much.

To CNPq (Process number 142344/2016-8) and PUC-Rio, for the grants without which this work could not have been carried out.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Bastos, João Antonio Dutra Marcondes; Fabricio Garcia, Alessandro. **Promoting Conversational APIs: A Conceptual Framework and a Method for API Design**. Rio de Janeiro, 2020. 149p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

APIs (application programming interfaces) play a crucial role in software development. Almost any programmer is often at the position of using third-party APIs. Currently, we find several researches that seek to explore and understand users' interactions with the API from an usability point of view. However, such studies leave out an important aspect of an API quality of use, the communicability. Unlike usability, whose definition is associated with ease of use and learning, communicability is associated with an artifact's ability to communicate its design logic. An API that lacks communicability can lead users to misuse and produce bugs in their code. This doctoral thesis addresses this problem from a Semiotic Engineering perspective. By characterizing an API as a mediating artifact for communicability, we started our thesis with the proposal that APIs should be conversational. We diagnosed in our first study that users, in certain cases, have difficulty in understanding the internal operating logic of an API just by looking at its interfaces. In this study, we found that APIs often lack in communicability. While usability is about the user's ability to learn and use an API, communicability is about the API's ability to transfer the designer communication to the user, thus exposing its design rationale. A conversational API is the one that can expose its internal logic through its interfaces, attending the pragmatic contexts of its users. From this study, we then set out to define what a conversational API is in practice and to investigate what methods or technologies would be needed to assist designers in creating such APIs. In this thesis, we propose a conceptual framework and a method to support the design of conversational APIs. When designing an API, the designer has in hands three different ways to send his message to his user: the source code, the documentation, and the behavior of the API. Our conceptual framework explores how to characterize and classify a conversational API according to the three types of messages from the designer's perspective. Our method of supporting conversational API design, which was inspired by the results and lessons learned from an action-research we conducted, consists of three steps. The first step is to help the designer on identifying who the API users are and their specific conversational needs. In the second step, the method helps the designer on modeling possible API conversations with the different mapped users to achieve their goals. Finally,

the method provides a set of guidelines to guide the designer in defining the API interfaces, including their parameterization. Using this method, we perform a case study with an API design, which aims at supporting the refactoring of Java programs. From the API designer's point of view, the method helped him on creating empathy with his users and better deriving and reflecting upon the requirements and conversations that the API should provide to the different user profiles.

Keywords

Application Programming Interfaces; Design Method; Conversational API; Semiotic Engineering.

Resumo

Bastos, João Antonio Dutra Marcondes; Fabricio Garcia, Alessandro. **Promovendo APIs Conversacionais: Um Framework Conceitual e um Método para o Design de APIs**. Rio de Janeiro, 2020. 149p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

APIs (do inglês - application programming interfaces) desempenham um papel crucial no desenvolvimento de software. Quase todos os programadores estão frequentemente na posição de utilizar APIs de terceiros. Atualmente, encontramos várias pesquisas que procuram explorar e compreender as interações dos usuários com a API do ponto de vista da usabilidade. No entanto, tais estudos deixam de fora um aspecto importante da qualidade de uso de uma API, a comunicabilidade. Ao contrário da usabilidade, cuja definição está associada à facilidade de uso e aprendizagem, a comunicabilidade está associada à capacidade de um artefato de comunicar sua lógica de projeto. Uma API que carece de comunicabilidade pode levar os usuários ao uso indevido e produzir bugs em seu código. Esta tese de doutorado aborda este problema a partir de uma perspectiva de Engenharia Semiótica. Ao caracterizar uma API como um artefato mediador da comunicabilidade, começamos nossa tese com a proposta de que as APIs deveriam ser coloquiais. Diagnosticamos em nosso primeiro estudo que os usuários, em certos casos, têm dificuldade em compreender a lógica interna de funcionamento de um API apenas olhando para suas interfaces. Neste estudo, descobrimos que as APIs muitas vezes carecem de comunicabilidade. Enquanto a usabilidade é sobre a capacidade do usuário de aprender e usar uma API, a comunicabilidade é sobre a capacidade da API de transferir a comunicação do projetista para o usuário, expondo assim sua lógica de projeto. Uma API conversacional é aquela que pode expor sua lógica interna através de suas interfaces, atendendo aos contextos pragmáticos de seus usuários. A partir deste estudo, nós nos propusemos então a definir o que é uma API conversacional na prática e a investigar quais métodos ou tecnologias seriam necessários para auxiliar os projetistas na criação de tais APIs. Nesta tese, propomos uma estrutura conceitual e um método para apoiar o projeto de APIs de conversação. Ao projetar uma API, o projetista tem em mãos três maneiras diferentes de enviar sua mensagem ao seu usuário: o código fonte, a documentação e o comportamento da API. Nossa estrutura conceitual explora como caracterizar e classificar uma API de conversação de acordo com os três tipos de mensagens da perspectiva do projetista. Nosso método de apoio ao projeto da API de conversação, que foi inspirado nos resultados e lições aprendidas de uma pesquisa-ação que realizamos, consiste em três etapas. O

primeiro passo é ajudar o projetista a identificar quem são os usuários da API e suas necessidades de conversação específicas. Na segunda etapa, o método ajuda o projetista na modelagem de possíveis conversas de API com os diferentes usuários mapeados para atingir seus objetivos. Finalmente, o método fornece um conjunto de diretrizes para guiar o projetista na definição das interfaces API, incluindo sua parametrização. Usando este método, realizamos um estudo de caso com um projeto de API, que visa apoiar a refatoração de programas Java. Do ponto de vista do projetista da API, o método o ajudou a criar empatia com seus usuários e a melhor derivar e refletir sobre os requisitos e conversas que a API deve fornecer aos diferentes perfis de usuários.

Palavras-chave

Interfaces de Programação de Aplicações; Método de Design; API Conversacional; Engenharia Semiótica.

Table of contents

List of figures

List of tables

The task must be made difficult, for only the difficult inspires the noble-hearted.

Søren Kierkegaard, *The essential Kierkegaard.*

1
Introduction

With increasingly complex and extensive software, several techniques had to be deployed over time. Among them, code modularization and reuse (39). Most programming languages have a wide variety of code packages ready to be used for various purposes. Other programmers can reuse these code packs through interfaces that display what operations are available. These types of interfaces, which encapsulate and abstract the internal content of a reusable code package, are called APIs (Application Programming Interfaces).

An API can be defined as a contract, with documentation representing an agreement between the interested parties. If one of these parties sends a remote request structured in a specific way, this will determine how the other party's software will respond, e.g., Web Services APIs. Moreover, an API can also be seen as a layer of abstraction and definition about a code package's internal behavior, such as APIs that define interface standards for different implementations, for example, Java and Android standard APIs. However, the research we developed in this Ph.D. thesis is agnostic to these definitions. For whatever it is, from the point of view we study here, an API will be a layer of abstraction and encapsulation of an internal behavior of a particular block of code that other programmers will use.

APIs play a vital role in improving software productivity and quality. For example, if a programmer needs to perform common operations on date and time objects, he can use some well-established API to perform such operations. The cost and time of software development are positively impacted by using APIs, as it prevents programmers from having to recreate programming routines that are already well defined and tested (16). The use of APIs also enhances software reliability as they are often code packages widely tested by a large community of programmers. These characteristics make the use of APIs common in the life of every programmer.

There is a wide variety of APIs for several purposes across various programming environments. Despite their relevance to software development, using and learning APIs still represent significant challenges. As a result, there is a growing number of forums and websites to discuss and ask questions about APIs. Usually, programmers can find on the internet numerous examples

and explanations about usage scenarios of popular APIs. In many of these discussions and examples, we can find a fair amount of misinterpretation about what an API does and how it should be used[1,2]. When some of these issues become recurrent, it usually indicates that the API's design was inappropriate. Moreover, these designs should have been better guided along its early design. The lack for proper support for API design requires additional designers' effort later in order to make the API use and learning by programmers more effective. Therefore, investigating the phenomena involving the interaction between programmers and APIs is relevant in the context of software development (21).

Recently, we have noticed an increasing demand for API development, especially when it comes to supporting the execution of complex activities (9). Consequently, the need to properly design APIs increases. Due to the complexity of the tasks they perform, we have observed an increasing use of APIs to support artificial intelligence. There are various complicating factors on the design of such APIs. Although API designers can expect professionals in the field of artificial intelligence to have some computer skills, they cannot expect from these professionals extensive knowledge and experience in software development. Therefore, the APIs used by these professionals should provide adequate interfaces to abstract the complexity of their programming tasks (8). Besides, API users should be able to understand the design rationale behind API interfaces in order to use them correctly. For instance, an Artificial Intelligence API sometimes omits the design rationale on some aspects of model training, such as why some hyper-parameters are set by default.

However, designing an API that explains the design rationale of who created it is not a trivial task. One way to accomplish this task is through the creation of *conversational APIs* (3). The notion of conversational APIs, characterized in this thesis, contains potential pre-established dialogues for the API to have with its users. The lack of explicitly defined conversational resources hinders the API users in understanding how to properly use their interfaces (3). In chapter 2, we demonstrated through a study with date and time APIs in Java how harmful it could be the use of an API that lacks in conversations. Although *"conversational"* is a widely used characteristic in the field of artificial intelligence to address chatbots and other types of interfaces. In this thesis, we treat *"conversational"* as a characteristic of interaction based on conversation, which was not necessarily created by artificial intelligence but pre-established by the artifact's designer.

[1]http://stackoverflow.com/questions/1755199
[2]http://stackoverflow.com/questions/14618608/

When creating an API, the designer sets in the code his beliefs and values regarding the operation that the API will perform. His design rationale is then established and needs to be made explicit through the API interfaces. In cases where this design rationale can be controversial, it is even more critical the conversational nature of an API. Whenever a design decision on a particular operation may be in disagreement with the users' understanding, the API needs to use conversational means to explain the design rationale and, consequently, the decision made for that operation. These issues are important for both simple and complex APIs. Even simple APIs such as those operating on date and time can have many controversial operations. Complex APIs, such as refactoring APIs present in many IDEs, can also benefit from conversationality to improve user understanding (38).

A example of lack of conversation extracted from this link[3] can be seen on the figure 1.1. In this example, a user reports an issue addressing the result of the operation of adding a month to January 31 (see Figure 1.1 - area 1 in red). Among the several answers provided by the other users, we observed that several users misinterpreted the API's logic. For instance, one of the answers to this issue argue that adding one-month is the same that adding add 30 days in the API, which would explain the unexpected result (see Figure 1.1 - area 2 in red).

The case reported above does not address difficulties for using the interface, but it exemplifies the poor communication between the API designer and the APIs user on how to properly using the interface to support the users' needs. The users know how to call the API operation, set parameters, and which data should be passed. However, it is noted that users do not understand the API's internal operating logic, based on inappropriate assumptions about its use. Once they are convinced that they know how to use the interface, they will not resort to its original documentation. For them, they will consider the interface has some *bug* as it did not work as they expected. Although the design rationale is present in the documentation, the lack of an explicit conversation in the interface leads to the misunderstanding.

In figure 1.2, we can see another real example of an API with insufficient conversational capabilities extracted from this link[4]. In this example, a user of the Google Maps Geolocation API reports a failure to calculate the distance between two locations. The user of the API claims that he is requesting the result of the distance calculation in miles (see Figure 1.2 - area 1 in red). However, the answer provided by the API is always with the distance calculated

[3]https://stackoverflow.com/questions/14618608
[4]https://issuetracker.google.com/issues/35829619

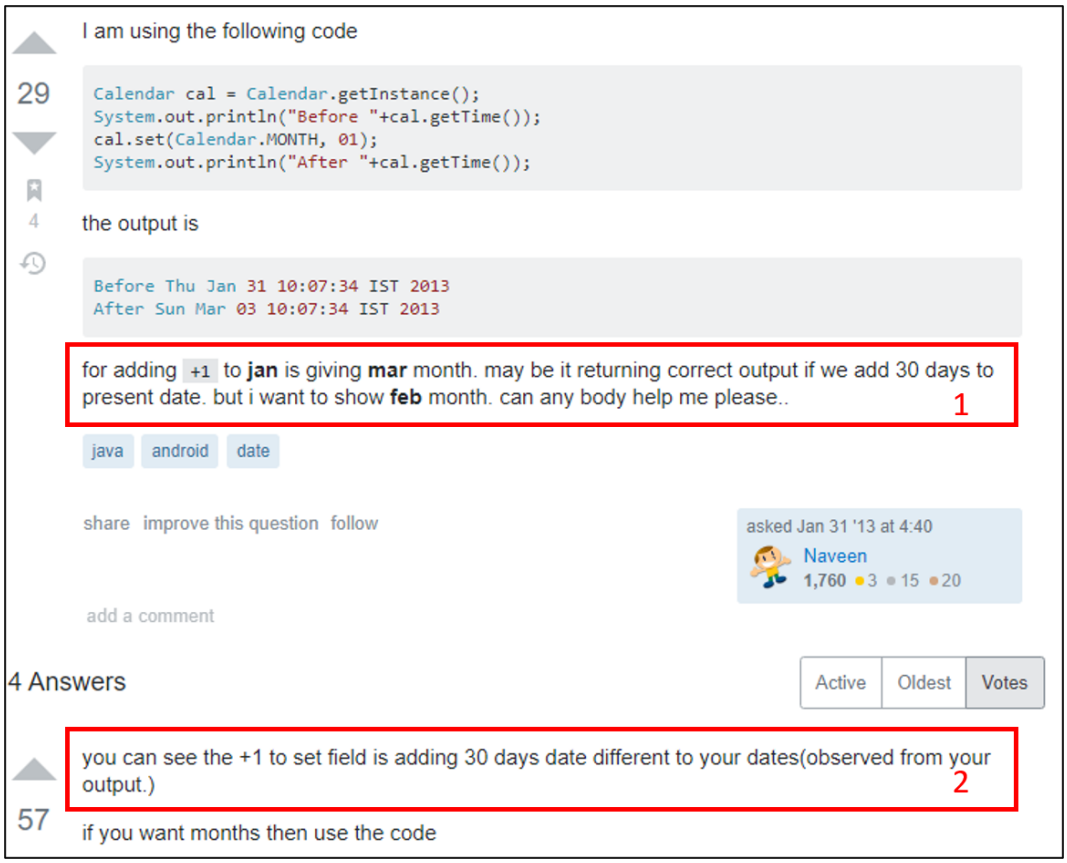

Figure 1.1: API with Insufficient Conversations - Java 7 Calendar API

in meters. Surprisingly, the API designer has re-classified this bug to expected behavior by transcribing the following excerpt from the API documentation: *"Note: this unit system setting only affects the text displayed within distance fields. The distance fields also contain values which are always expressed in meters"* (see Figure 1.2 - area 2 in red).

Misunderstandings between API designers and users are common. The example described above illustrates how the geolocation API is unable to get users to understand its design rationale. Unfortunately, the documentation is often perceived by designers as sufficient and mandatory to support the proper use of API interfaces. However, issues like this can be avoided by improving the APIs conversational capabilities. For instance, if the API designer added an extra field in the response with an explanation or a warning about this behavior, the user would pay attention to this fact, and the problem might not occur. As we saw in the examples above, sometimes it is not possible to make users and designers understand each other about the expected results of some operations. However, we can make the API interface, through pre-made conversations, expose the design logic and what result is expected from each operation.

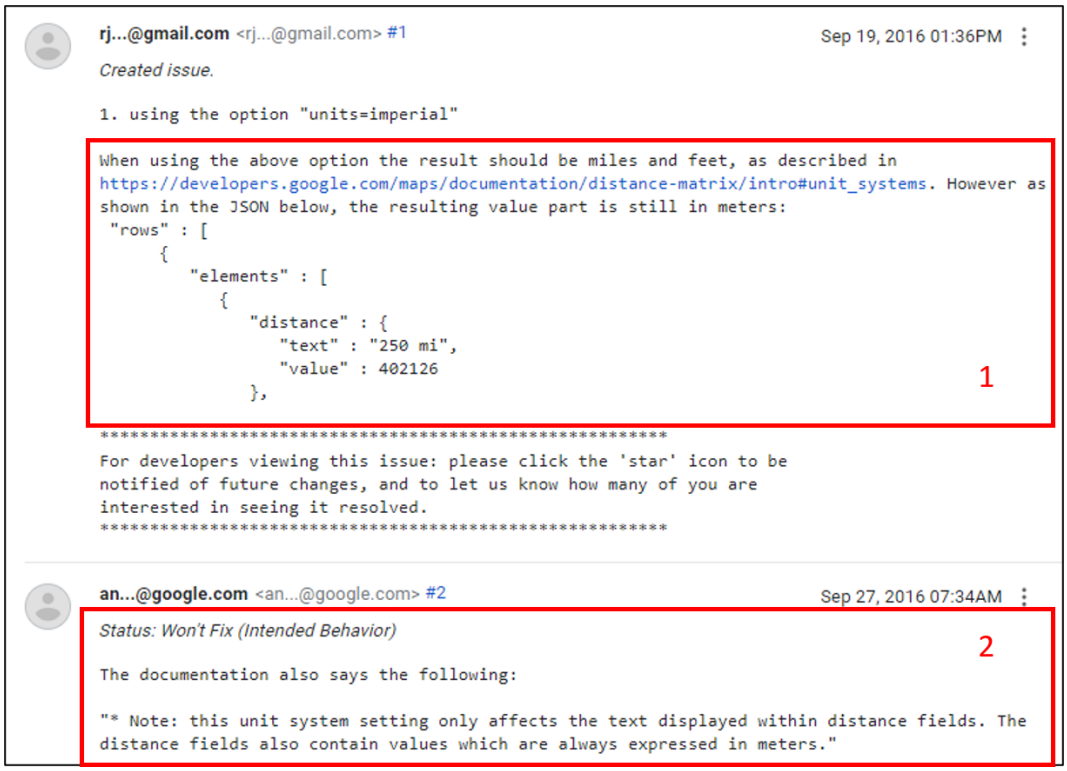

Figure 1.2: API with Insufficient Conversations - Google Maps API

1.1 Problem Statement and Related Work Limitations

As we saw in the previous section, a lot still needs to be studied about the APIs quality of use. In this section, we discuss the studies we found in the literature and show the problems that are still open in this area. Since the 1980s, researchers have been discussing the best ways to design APIs (30). The work of McLellan et al. (26), in 1998, is one of the first studies to gain importance in addressing APIs' usability, that is, the simplicity and ease of use of an API. The study relied on techniques commonly adopted in Human-Computer Interaction (HCI), such as interviews, usage scenarios, recordings, and others. The results of this study describe how code snippets can influence programmers when using software interfaces, making programmers rely on code snippets that contain errors and leading to misunderstandings on how to use an API effectively. This work has influenced many researchers to investigate APIs' use and design in a more human-centered way.

Several of these researchers have successfully tested HCI theories in APIs usability, where the programmer assumes the role of the software user. Studies carried out so far have used cognition-based theories, heuristic analysis, interviews, and questionnaires typically aimed at evaluating the usability of APIs (10, 41, 50). Farooq and Zirkler describe a group-based usability

inspection method to evaluate the usability of APIs, the API Peer Reviews (17). They contrast their method with usability tests, arguing that they can be used together and complement each other. According to the authors, despite identifying less usability defects than a usability testing, API Peer Reviews has a lower cost and execution time, thus making it useful for early usability assessment of APIs. In all of these initiatives, the focus of interest is what happens to the user-programmer interaction trough the API. They do not address the problem of misunderstandings between users and designers.

In this thesis, we look at the quality of use of an API through another aspect, the communicability. Unlike usability, whose definition is centered almost exclusively on users, communicability is defined as the ability of an interactive digital artifact to communicate to users, effectively and efficiently, the intention of its designers. Previous researches are concerned with the API's usability, and they lack in investigating communicability aspects, including pragmatic issues as promoting the understanding between users and designers.

Afonso et al. (1) is the first work on this topic. In this work, the authors propose investigating the quality of use of an API under the lens of communicability. The authors use the theory of Semiotic Engineering to explore how communicability (or the lack of it) can affect the final use of an API. Although Afonso and colleagues have investigated the communicability of APIs, they did not focus on the API design. Instead, they focused on API evaluation, and did not address the design process or provide tools to help the designer to communicate his design decisions. This thesis research positions itself as a complementary work that Afonso carried out, taking the principles proposed by him to create the concept of conversational API. Moreover, we propose a design method for conversational APIs based on the Semiotic Engineering theory. Here we have our first research problem that was treated by this thesis.

Problem 1: How to Communicate Design Decisions to API Users?
In other words, how to make users and designers understand each other when pragmatic conflicts of interest occur?

This problem emerged after the understanding that APIs, which the use may modify according to its context, should have enhanced communicability for better pragmatic adequacy. From this conclusion, we have created the concept of conversational API. So, with the definition established, we needed to offer tools or technologies to assist the designer in creating such APIs.

Searching for solutions that support API design, we found some approaches for improving the API design process. Watson (54) and Mindermann (27) introduced approaches focused on the API' easiness of use, grounded on consolidated usability concepts and techniques from HCI. Eduardo Mosqueira-Reya et al. (28) generate a compilation of guidelines and heuristics that should be applied along the design process to achieve adequate usability in APIs. Although these approaches are concerned with the API's quality of use, they do not address communicability aspects, including pragmatic issues as promoting the understanding between users and designers.

The lack of conversational APIs lead users to face difficulties on applying APIs in their projects. Alternatively, technical literature presents tools for assisting developers in properly using APIs. Yessenov et al. (56) propose *DemoMatch*, a tool to support programmers in discovering how to use an API based on interactions with software already using it. Ichinco et al. (22) propose *Example Guru*, a tool for recommending APIs based on the context of the programmer's code. Addressing code verification, Nguyen et al. (32) present a tool for scanning the source code of Android applications to find possible security flaws resulting from the inappropriate use of APIs.

More recently, some studies investigated the pragmatic issue of APIs misunderstandings (34). Although they did not propose methods or techniques for improving the API communication, these studies may represent resources for supporting understanding the limitations of the communication among designers and users. Nielebock et al. investigated the misunderstandings on using APIs, leading to their misuse and even to the incidence of bugs. To mitigate this risk, the authors introduce a tool for identifying API misuses and offering rules for fixing those misuses. The work of Lamothe and Shang aims at understanding the appropriations made by API users. The work of Lamothe and Shang aims to understand the appropriations made by API users, i.e., ways of using the API other than the one originally proposed. The authors found three appropriations patterns followed by the API users. These patterns can help API designers to understand the possible derivations made by API users and adapt them to these realities in new versions (24).

Although some of the papers discussed offer design solutions for APIs, they do not focus on our thesis's central problem, which is the support to the designer to expose his design rationale in order to reduce the users misunderstandings. On the other hand, the more recent studies (34, 24) that deal with this issue do not offer a method to support the API design. Such studies focus on providing tools to mitigate the user's difficulty, but they offer none to support the one who is creating the API. Thus, this is another research

problem we are covering with in this thesis.

Problem 2: How to Support the API Design with Focus on Conversation?

1.2 Main Contributions

To address both problems raised in the previous section, this research has investigated the phenomenon of communicability in software APIs. Through qualitative studies, having Semiotic Engineering as the supporting theory, this research started with a study on the interpretation and inspection of meaning in date and time APIs available in the Java development toolkit. In this study, we had two main conclusions. The first is that programmers are not used to routinely consult the documentation of an API. They prefer uncomplicated APIs that share their values and do not need much configuration or parametrization to perform operations. The second conclusion was that designers usually believe that the documentation is part of the API interface and is practically mandatory to be consulted. As a conclusion to this study, we characterize an API as a conversation mediator artifact. Consequently, we established that an API should be conversational.

Contribution 1: Characterization of an API as a Mediating Artifact of Conversation Between Users and Designers

From this study, we went in search to establish a conceptual framework that could help designers to improve conversation of their APIs, thus promoting better communicability and understanding of design decisions. We propose our conceptual framework for the characterization and classification of conversational APIs through theoretical inferences when transporting the concepts of Semiotic Engineering to the context of software APIs. In this framework, we list how to classify the signs of an API and explain how the designer can use them in pursuit of a fully conversational API. We consider a fully conversational API when it support all the necessary conversations between user and design in the API interfaces. For example, suppose an API has some internal decision made that can be controversial, such as the *"add"* operation in a date and time API. In that case, the designer should indicate this on the interface, even if a more in-depth explanation of his decision is in the documentation.

To characterize and classify the possible levels of conversation that an API can have, we define a conceptual framework. Based on Semiotic

Engineering, our conceptual framework appropriates the concepts of signs proposed by the theory to characterize the presence of conversation at three different levels: rudimentary, metalinguistic, and fully. So, our second scientific contribution of this thesis is our conceptual framework.

Contribution 2: A Conceptual Framework for Characterizing and Classifying Conversational APIs

As much as our conceptual framework offers theoretical support for conversational API, it does not fulfill the methodological support needed for a API design process. Thus, seeking to understand how to design an API within a professional software programming environment, we conducted a technical action research to experiment techniques and tools that could improve the conversation in the created API. In this action research, which lasted about six months, we had the opportunity to carry out three complete cycles of an action research, from the identification of one problem to the evaluation of the proposed solution to this problem. At the end of the research, we had a set of lessons learnt that could be used in building a step-wise method to support the design of conversational APIs.

Based on this results, we propose our method. Colloquy, as we call, helps designers on introducing conversations into their APIs. Basically, Colloquy consists of three steps. The first step is to help designers on identifying who are the users of the API and their specific conversation needs. In the second step, the method helps the designer to model the possible API conversations with the different mapped users so that they achieve their goals. Finally, the method provides a set of guidelines to guide the designer in the declaration and parameterization of API interfaces. Therefore, we have as our third scientific contribution, the proposal of a method to support the design of conversational APIs.

Contribution 3: A Method to Support Conversational API Design

With our proposed conversational API design method, we now need to perform a feasibility assessment and the positive and negative results it could bring to API design. To do this, we planned a case study with the design of a code refactoring API. In this API, whose primary focus was to explore the customizations users could do with refactoring, the conversation was something critical, and that should be significantly explored.

By using our method, the designer was able to create an API with interfaces that were able to negotiate the meanings of refactoring with its users. As the main results of this study, we pointed out the creation of the conversational API for refactoring Java programs and a set of potential improvements that we could promote in our method. Therefore, adding the case study findings to the action research results that inspired the method, we have our last scientific contribution.

Contribution 4: Report of Two Empirical Studies Conducted in Supporting the Design of Two Different Conversational APIs

1.3 Thesis Outline

The remainder of this thesis, which is a compilation of technical papers (accepted or under submission), is organized as follows.

Chapter 2 presents a Semiotic Engineering (13) study on the communicability of date and time APIs. In this study, we analyzed metacommunication – a central concept of the theory we use - between designers and users of date APIs, specifically, using the "SigniFYIng APIs" method (15). We relate the results of the analysis with the testimony of professional programmers collected during an interview concerning their experience with APIs and programming. Chapter 2 presents, illustrates and discusses the value of the results achieved with an API communicability analysis that, in our view, is a promising addition to research initiatives that have been exploring API usability.

Chapter 3 presents the conceptual basis of the doctoral thesis we are defending. We define what a conversational API is according to the Semiotic Engineering view and demonstrate how the classification of signs proposed by Semiotic Engineering can be applied to the context of software APIs. Furthermore, we demonstrate how our conceptual framework can be applied to assist designers in defining conversations in their APIs. Finally, we also present and discuss an evaluation of the framework on top of three date and time APIs.

Chapter 4 presents a technical action-research as a key study performed during this Doctoral research. In this research, which lasted about six months, we had the opportunity to apply the concept of conversational API to develop a real API, within the context of deep learning in a professional software development environment. To apply the concept, we followed the API redesign and searched for techniques and tools that could help the design team to create a conversational API. The chapter presents as main contribution a set of lessons learnt that can improve an API design process.

Chapter 5 presents a design method for conversational APIs. This method guides the designer through the implementation process of the API conversations. In the three-step method, we use HCI tools and techniques adapted to the context of API to help the designer reflect on their interfaces. These steps help the designer to think about the best conversation paths his API should offer users. Moreover, this chapter also brings the results of a case study of API design. In this study, we use the method with a participating designer to create a conversational API within the context of customizing source code refactorings.

Chapter 6 summarizes the conclusions of our work, presenting the main contributions to the state-of-art and future work.

2
API as Conversation Mediator Artifact

When programmers use an API, they play the role of "software user". Since the quality of program and system usage affects user productivity, more attention has been paid to API usability issues (21, 31, 49). In this thesis, however, we focus on another dimension of quality of use, "communicability", which is a characteristic of a semiotic engineering approach (13). We also discuss how communicability plays a different role as compared to usability, and how communicability based methods can offer different, sometimes complementary, results from usability based methods. In addition, we discuss how Semiotic Engineering concepts can be smoothly incorporated into the context of software APIs to create the concept of conversational APIs (section 1.2).

In this chapter, we discuss the understanding that APIs should have adequate communicability for better pragmatic adequacy (section 2.5). We show that putting all design rationale only into the API documentation can cause serious misunderstandings by the user. We conduct a empirical study with seven experienced programmers in order to understand how they use and infer meaning in APIs (section 2.3). Through an analysis of date and time APIs, we show that programmers usually infer meaning by looking just at the interfaces and sometimes do it incorrectly (section 2.4).

We also analyzed two date and time APIs using the “SigniFYIng APIs” (15), a method developed for the analysis of API communicability, taking advantage of concepts proposed by Semiotic Engineering, but adding facets that are specifically relevant in the context of APIs (section 2.3.1). The results of the analysis were related to the evidences collected in the empirical study with professional programmers. This chapter presents, illustrates and discusses the value of the results achieved with an API communicability analysis that can help the designer in reflecting on his API conversations.

This chapter presents a paper that was published in the *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* under the name *"Metacommunication Between Programmers Through an Application Programming Interface: A Semiotic Analysis of Date and Time APIs"* (3). This paper is presented as the first contribution from the work we developed in this thesis: the creation of the concept of conversational APIs (section 1.2).

Metacommunication Between Programmers Through an Application Programming Interface

A semiotic analysis of date and time APIs

João A.D.M. Bastos, Luiz M. Afonso, Clarisse S. de Souza
Department of Computer Science
PUC-Rio
Rio de Janeiro, RJ, Brazil
{jbastos, lafonso, clarisse}@inf.puc-rio.br

2.1 Introduction

Application Programming Interfaces (APIs) are everyday tools to any software developer. In most programming languages, there is a wide variety of APIs for a number of purposes. In spite of their relevance to software development, APIs still represent a great challenge of use and learning for programmers. As a result, there is a growing number of forums and websites to discuss and ask questions about the use of APIs. Usually, programmers can find on the internet numerous examples and explanations about usage scenarios of popular APIs. In many of these discussion and examples, we can find a fair amount of misinterpretation about what an API does and how it should be used[1,2]. When some of these issues become recurrent, it usually indicates that the API's design may be improved in order to make its learning and use by programmers more effective.

Since the 1980s, researchers have been discussing the best ways to do the design and evaluation of APIs (30). Several of these researchers have successfully tested Human Computer Interaction (HCI) theories in the context of APIs usability, where the programmer assumes the role of software user. Studies carried out so far have used cognition-based theories, heuristic analysis, interviews and questionnaires typically aimed at evaluating the usability of APIs (10, 41, 50). In this work, we focus on the communicability of APIs, from a Semiotic Engineering (13) perspective. Unlike usability, whose definition is centered almost exclusively on users, communicability is defined as an interactive digital artifact's ability to communicate to users, in an effective and efficient way, its designers' intent. We look at both, designers' and users' aims, while evaluating the quality of communication through the artifact's interface. In this context, we sometimes refer to these interlocutors as

[1]http://stackoverflow.com/questions/1755199
[2]http://stackoverflow.com/questions/14618608/

"designer-programmer" and "user-programmer", to emphasize their dual role in the production and consumption of reusable software packages, respectively.

The main motivation for our approach is the fact that communicability addresses different aspects, sometimes complementary, to those addressed by usability approaches. Cultural aspects for example, may play a different role in communication than in cognition. Consider, for instance, the case of an API in PHP that uses the "needle in the haystack" metaphor to communicate how a string search function works:

strpos(string $haystack, mixed $needle [, int $ offset])[3]

For the programmer who is not familiar with this metaphor, this method signature would not make any sense[4]. The API designer, in turn, does not have a *cognitive* problem with the metaphor, but probably a *communicative* problem to solve.

Another example can be found in Google's Issue Tracker tool[5]. A user-programmer says that the Directions API is wrong because although his program specifies that an operation should represent the result in imperial units, the API insists on returning the result in meters. The user-programmer does specify *"units = imperial"* in his program, however the API returns the *"distance.value"* field in meters. If we look at the API documentation, we can understand his confusion. In fact, the API returns the value in the requested unit, but it does so in a different field (*"distance.text"*), which is not what the user-programmer is used to seeing. Thus, communicability is not good in this case. The intent of the designer-programmer, although expressed in the documentation, is not well communicated in the signature of the method.

Once again, usability-based investigations might identify issues that are similar or related to the one above. However, a communication perspective addresses aspects involving both interlocutors, and not only the user-programmer. This shift in perspective has implications to the design of APIs, since it is generally the API designer's wish to communicate his design vision for the software artifact in an effective way. Bringing the designer as sender of this communication process into the scope of analysis potentially promotes a stronger commitment of the designer with the quality of use of the API's artifacts.

To support the detection of problems such as those described above, there is a methodological tool based on Semiotic Engineering theory (13) called

[3]http://php.net/manual/en/function.strpos.php
[4]http://stackoverflow.com/questions/4808758
[5]https://issuetracker.google.com/issues/35829619

SigniFYIng APIs (15). The study presented in this article is one of the first practical illustrations of how this method works, and it focuses on a very common domain in the daily routine of programmers: operations with dates. We used the SigniFYIng APIs method to perform the inspection of two date-related APIs, Java7 (*Calendar*) and Java8 (*LocalDate*).

In an attempt to support our analytical results with evidence provided by practitioners, we also carried out an interview with seven participants, all of them experienced programmers. In addition to the *Calendar* and *LocalDate* APIs, we discussed other APIs in the same domain but from different programming languages. Part of the interview was a kind of quiz about two types of commands involving Date and Time in APIs for these languages.

The results of this study point out at how a combined semiotic-cognitive approach can: i) shed new light on what happens when a programmer uses an API from another programmer in the development of his or her own software; ii) analyze the quality of the APIs in relation to its designer's values and intentions, as well as to its user's; iii) inform the redesign or correction of APIs with communicability problems, as well as the design of new APIs, about issues directly related to social communication experiences which are familiar to their creators (designer-programmers).

We believe that item (iii) above may directly benefit user-programmers, once designer-programmers perceive what good communication of their own interests and intentions is about, and begin to explore the means of achieving it. In this context, user-programmers will probably be better equipped to understand what APIs are, what purposes they serve and how they should be used. Good communicability may also support user-programmers' decision about whether an API is the best for them or not, and why. However, at the current stage of our research, these benefits are still only a likely possibility, which we should investigate in the next stages of our project.

This article is divided into 5 more sections. In section 2.2, we discuss how selected studies relate to our research. In section 2.3, we provide a theoretical basis on Semiotic Engineering along with a brief description of the SigniFYIng APIs method, as well as a detailed description of how the studies were conducted. Section 2.4 presents the results and in section 2.5, we discuss our findings. Finally, in section 2.6 we conclude with a summary of the results and future work that we wish to accomplish.

2.2 Related Work

The phenomenon of interaction between programmers and APIs is not a field of study restricted to the area of Human Computer Interaction. Several studies in Software Engineering and Programming Languages have their way of approaching this problem. However, our report of related work focuses on studies that concentrate on the programmer, his activities, interpretation or experience with APIs.

Early research focused on this theme appeared already in the 1970s and 1980s. However, it was not until the late 1990s and early 2000s that such work became more prominent. The study by McLellan et al. (26), in 1998, is one of the first studies to gain importance in addressing the usability of APIs. It used techniques commonly adopted in HCI studies of use experience, such as interviews, usage scenarios, recordings and others. In their results, the authors describe how code samples can influence programmers when using software interfaces, making programmers rely on code samples that contain errors, leading to misunderstandings on how to use an API effectively. This work has influenced many researchers to investigate the use and design of APIs in a more human-centered way.

In the last decade, the interest in the quality of use of APIs has increased and several researchers have been studying their usability. Some work focuses on strategies to improve the API design: we can highlight Henning and Michi (21), Watson (54), and Mindermann and Kai (27). They all try to define ways of designing APIs that are easier to use by the programmer. In addition, we also have work focused on usability evaluation. Farooq and Zirkler (17), for instance, describe a group-based usability inspection method to evaluate the usability of APIs, the API Peer Reviews. They contrast their method with usability tests, arguing that they can be used together and complement each other. According to the authors, in spite of identifying less usability defects than a usability testing, API Peer Reviews has a lower cost and execution time, thus making it useful for usability assessment of APIs. In all of these initiatives, the focus of interest is what happens to the user-programmer during interaction with the API.

Recent work by Myers and Stylos, 2016 (31) has brought more people to the problem space of API usability studies. They define other stakeholders besides user-programmer, who may play a role in the context of an API usage, from API designers (programmers, documentation writers), concerned with making more efficient APIs at the lowest possible cost, to final product consumers, who may be indirectly affected by the code produced using an

API. Since an API has to satisfy the needs of different stakeholders, its design becomes more complex. The authors argue that there are a variety of "human centered" methods that can help design more usable APIs. Among such methods, we highlight two that are widely applied in the classic view of HCI: Nielsen's heuristic evaluation (35) and the cognitive dimensions of notations framework (CDNs) (19).

Like Myers and Stylos, we also think of different stakeholders when addressing the quality of API use. However, in our perspective, the API designer is not only a stakeholder, but truly a participant of the interaction process that takes place through the protocols defined for the API interface. Traditional methods of usability evaluation, such as those described by the authors, frame API usage as a matter of computer-mediated human communication, that is, as communication between API producer and API consumer through the programming protocols that constitute the API interface. In our work, we explore this complex social communication process based on Semiotic Engineering.

2.3 A Semiotically-Based Research Study

The study presented in this article was divided in two blocks: an interview with experienced professionals and a semiotic analysis of two Date & Time APIs. Both were conducted in parallel and guided by intentionally problematic computations, such as: "Add a month to January 31, 2016" and "Create the date February 31, 2016." The analysis of such computations is important because they may silently emerge in corner cases reached during iterations, for instance, leading to faulty program behavior. In this section, we will present in detail the methodology of the study, but first we present a very brief overview of the semiotic theory that we adopt.

2.3.1 Semiotic Engineering and SigniFYIng APIs

Semiotic Engineering is originally an HCI theory that characterizes human-computer interaction as a process of metacommunication between digital technology users and designers, mediated by the technology's interface. That is, when designing a product, the designer inscribes in it his or her vision of how, where, when, why, and to what effects the user can communicate with the product. The user, in turn, can only interact with the product through communication means, modes and codes or languages that have been predefined by the designer. We then say that the interaction process has three

interlocutors. The designers who encode their intentions into software (and other features of the technology), the users who, when interacting with the product, express their own intent and interpretations, and the technology itself, which represents the designer at interaction time.

In figure 2.1, we sketch a comparison between the more widespread user-centered view of HCI and the view proposed by Semiotic Engineering. The user-centered view is concerned with mental and physical workloads associated with a user's interaction with software. Norman's well-known 7 Step Theory of Action, for example, supports a conceptual model of human-computer interaction where the users' cognitive activities are so important that HCI design is framed as a matter of cognitive engineering, the construction of artifacts to be known, understood, memorized and controlled (37). The Semiotic Engineering view conceptualizes HCI as a computer-mediated conversation between software producers and software users. The language of such conversation is the interface language, in which a system's interface – representing its designers at interaction time – communicates to users, as the conversation unfolds, the designers' intent and design rationale. By comparison, in this alternative view HCI is framed as a matter of communication engineering, or more precisely, Semiotic Engineering (13). The designers necessarily "participate" in interaction and are just as entitled to having intentions and communicative needs as users do.

Figure 2.1: Cognitive View vs Communicative View

The Semiotic Engineering view can be smoothly carried over to a programmer's interaction with APIs created by another programmer. There is metacommunication in place between the user-programmer and the designer-programmer. The API designer communicates his intentions by means of expression defined in method signatures and protocols. The user-programmer, while developing his software and using the API, expresses his own interpre-

tation of designer-programmer's intent. Therefore, we may have cases of good and bad communicability in APIs, just as with digital technology user interfaces.

SigniFYIng APIs is a method developed for the analysis of API communicability, taking advantage of concepts and methods already proposed by Semiotic Engineering for HCI, but adding facets that are specifically relevant in the context of APIs. Since the interaction between designer-programmer and user-programmer involves the use of a programming language, SigniFYIng APIs also incorporates a cognitive analysis of programming notations using the CDN framework (19). However, by focusing on human communication mediated by a "programmed" interface, SigniFYIng APIs strongly emphasizes the "communicative intent" of the parties involved (API producer and consumer), whose analysis is best evidenced in cases of communication failures. For this reason, as in the communicability evaluation method (CEM) (14), communication failures and breakdowns are the starting points to the analysis of API communicability.

SigniFYIng APIs consists of three steps (see figure 2.2) that can be performed in successive iterations. The first is the reconstruction of metacommunication, where the researcher characterizes the presumed intent of the API designer and creates scenarios of use that focus on relevant aspects of the API under investigation. From these scenarios, the expressions and semantics of the API language are analyzed in the second step. The researcher identifies their presumed effect, where expressions and semantics can cause the user-programmer to misunderstand or even miss the designer's intent. Finally, the researcher places himself in the position of a user-programmer and executes the selected scenarios. Each identified communication failure is then classified in accordance with a set of categories provided by the method.

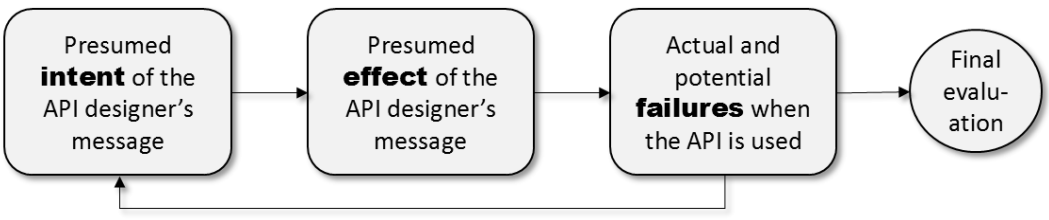

Figure 2.2: SigniFYIng APIs Steps

2.3.2 Methodology

The goal of our study was to verify the correspondence between what a semiotic analysis of widely used APIs diagnoses as "communicability issues" in their design, and the perception of experienced programmers about their practice and use of APIs. Therefore, we elaborated a study that was carried out in two parallel blocks, one was analytical and the other consisted of interviews with programmers. We selected a set of APIs from a very common domain in software development: date and time. Nine APIs of various programming languages were used in the interview block, and two of them were used in the analytical block, along with the SigniFYIng APIs method. Figure 2.3 provides a general illustration of our methodology.

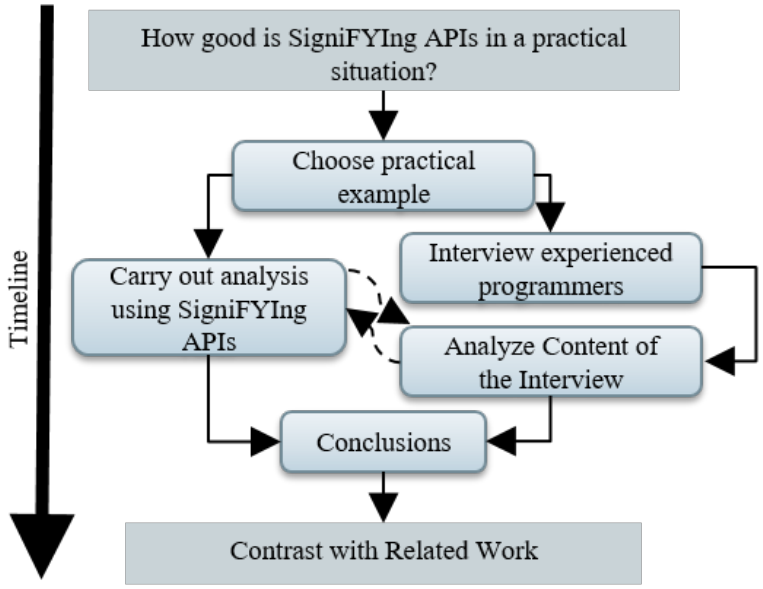

Figure 2.3: Metodology

In the interview, we had the contribution of seven participants, all programmers with five to fifteen years of professional experience in software development. With each participant individually, we conducted an interview that lasted about 45 minutes regarding their preferences and tastes in the use of APIs in their work in general. The first part of the interview asked open questions meant to let the participants speak freely about their preferences. We encouraged them to talk about situations they had experienced themselves to illustrate what they were saying.

After this initial part of the interview, we presented them a small quiz about the expected behavior of date and time APIs in seven programming

languages. For this, we elaborated a scenario with two very simple operations involving date manipulation. The first one was adding a month to a certain date. We purposely determined that the date to be incremented was January 31, 2016. The other operation was the creation of a date. Also purposely, the parameters for the new date creation were: 31 as day; February as month; and 2016 as year. Since we wanted to collect the participants' impressions and interpretations about the different API protocols and behavior, we provided them with syntactically correct code snippets that attempted to perform the above-mentioned operations. Then we asked the participants to analyze and comment on each one of them.

The programming languages and APIs chosen for the study were Java 7 (Calendar), Java 8 (LocalDate), JavaScript (Date and Moment), Lua (time), SQL-PostgreSQL (date), C# (DateTime), Python (datetime) and PHP (DateTime). The choice of these languages was such as to promote some variety among them. There were object-oriented languages, such as Java and C#, scripting languages such as Lua and JavaScript, and declarative languages such as SQL. We chose APIs that are standard in the corresponding language, that is, those that are distributed with the base language package (as in Java, C#, SQL and Lua), or that are indicated as standard in official language websites (as in Python and PHP). In particular, in the case of JavaScript, we used two APIs, the language's native API (Date) and a second API called Moment.js, which is the most used by the JavaScript programming community. This was necessary because the JavaScript Date API does not have suitable commands for all the operations described in our scenario.

During the interview, we introduced the participants to the languages and related APIs, and their respective code snippet to perform the operations proposed in the scenario. Participants were then invited to fill in a table where they described their expected outcome for each of the operations performed by the APIs. In addition, they should also fill in their experience on a scale of one to five, where one would be "I know very little or I do not know the API" and five would be "I have full knowledge of the API" on each of the APIs investigated. Table 2.1 illustrates the headers and two of the eight programming language rows of the table that was handed to participants.

Table 2.1: Part of Table Presented to Participants

	Add a month to date January 31, 2016	Create the date February 31, 2016	Indicate Your familiarity with PL (1 to 5)	Indicate Your familiarity with API (1 to 5)
Java 7	*«code»*	*«code»*		
Java 8	*«code»*	*«code»*		

After the tables were filled out, we conducted a second phase of interview with questions concerning the participants' thoughts about how the code snippets would execute, in order to get their perceptions, anticipations and understanding of the APIs behavior. At this stage, we wanted to find out the possible aspects that led them to expect this or that outcome. We asked if previous knowledge of the programming language and the APIs themselves had any influence on the results. Then, in order to instigate them into deeper reflection, we showed other possible answers and asked them to think of reasons for such answers. In the end, we showed them a third version of the table with the actual computed results in each case, and asked them to comment on what could be the API designers' intent for such API behaviors.

In the analytical block, we selected a subset of the APIs used in the interview block, namely the Java 7 and Java 8 APIs. We reduced our scope to only these two APIs because our method requires much deeper analysis of the APIs, and our main goal was not to compare the APIs. We just wanted to verify the kind of correspondences that might be found between communicability issues raised by the inspection method and the participants' perceptions of their practice and contact with APIs. Therefore, we believed the inspection of a smaller number of APIs would provide sufficiently relevant results.

While executing the SigniFYIng APIs method, we first reconstructed the metacommunication template according to the method proposed in order to capture the overall logic of the API. By so doing, we characterized how the API presents itself, and how it relates to the programming language. Then various iterations were performed in the cycle comprehending the communicative dimensions "intent, effect and failure", where we analyzed scenarios in which potential communicability breakdowns could affect the use (or misuse) of the API. In the end, we classified these communicability breakdowns using the method's tags (or categories). The following section presents the results from both the analytical and the interview parts of the study.

2.4 Results

This section begins with the interview block, where we describe the outcome of the activity and provide a categorized set of interview statements that have been selected for relevance and illustrative power. We then proceed to the communicability issues identified during the inspection with SigniFYIng APIs.

2.4.1 Interview

Table 2.2 presents a compilation of the data filled out by the participants regarding the expected result for each operation in the different APIs. For lack of space in this paper, the table includes only the Java 7 and Java 8 APIs, the same ones that were used in the analysis with SigniFYIng APIs. The first column contains the names of the languages and APIs. The second column summarizes all the answers proposed by participants for the operation "add a month to date January 31, 2016", with the number of coincidental answers in parentheses. The third column, following the same notation, presents the results for the operation "create the date February 31, 2016". The first value of each cell, in boldface, is the value actually produced by the corresponding API.

Table 2.2: Participants Results

API	Add a month to date January 31, 2016	Create the date February 31, 2016
Java 7 (Calendar)	**Feb. 29, 2016 (2)** Feb. 31, 2016 (1) Mar. 01, 2016 (1) Mar. 02, 2016 (1) Mar. 03, 2016 (1) Error (1)	**Mar. 02, 2016 (1)** Mar. 03, 2016 (1) Error (5)
Java 8 (LocalDate)	**Feb. 29, 2016** (2) Mar. 01, 2016 (2) Mar. 02, 2016 (1) Mar. 03, 2016 (1) Error (1)	**Error (7)**

With the other APIs used in the interviews, we get different results than those presented by Java 7 and Java 8. In Lua, PHP and JavaScript, for example, the operation of adding a month to the date January 31, 2016 results in the date March 2, 2016. Including the APIs not shown in table 2.2, no more than two participants (per API) matched their expected result with the actual API output for the operation. This is a clear indication that there are issues related to this interaction. The APIs do not follow the same pattern, and the participants do not understand what is going on behind the interface. Even experienced programmers fail to make a correct anticipation of API behavior. It seems that the notion of "counted month" is less dependent on formal convention than contextual interpretation. In this corner case context, it is clear that there are variant interpretations, with no consensus about the outcome due to the inherent ambiguity in this case. The point of interest here

is that each API "implements" its designer's interpretation, which matches some user-programmers' own interpretation, but not of all of them.

In the operation to create the invalid date "February 31, 2016", we may notice less confusion among the participants. However, the APIs still do not agree with each other. Some APIs accuse error, with messages saying that it is impossible to create the requested date, while others, as seen in Java, adjust the value for the following month, producing the date "March 2, 2016". In Java 8, all participants hit the result, while in Java 7, only one did. The relevant fact here is that most of the participants anticipated an "error" outcome for this operation. We suspect that this is due to the participants' reliance on common sense knowledge about dates. An explicit command to create an impossible date can naturally be expected to fail.

Considering that this test puts the "user-programmer" in a (pragmatically or semantically) faulty situation, which is not the case with the other operation, what is surprising is that, in spite of much narrower distribution than with the other command, there is still no consensus among programmers. Two of them consider that the API response will not be an error. This is probably due to the lenient behavior of the Java 7 API. One of the respondents precisely anticipated the leniency and the result, while the other only correctly anticipated the leniency, but missed the result. We have associated this with the professional programming experience of both.

The most important point in the above is the leniency of the programming language. What reasons may have led the API designer to opt for leniency in a command explicitly inconsistent with the API domain? Although we cannot know for sure, what matters is that most programmers did not understand this behavior or anticipate the outcome correctly. Failure to communicate frustrates both users and designers, and is more detrimental to users. The theory used in our study emphasizes that the designers' intent is also thwarted. Therefore, the ultimate goal of this type of research is to help designers learn what they need in order to formulate and achieve their intent more effectively.

During the interview, we noticed a common point raised by all the participants, the need for examples or tutorials to perform tasks using an API. We have selected two testimonies that highlight this point. Participant 4 said, "*When I get third-party software, the first things I look for are small, ready-made examples of how the API is used*". Participant 2 reinforces the same need in his testimony, "*The first step I take when I study an API is to look at some Wiki or blog that explains the general steps of what it is and [gives] quick examples I can use.*" Participants also expressed a preference for simpler APIs where they do not need to configure or parameterize an initial state. For

participant 1, "*The less I need to set up, the better the environment is.*" This is similar to what, in other words, participant 6 said: "*what helps is to be less bureaucratic.*"

Participants want the API (proxy of its designer) to communicate seamlessly with them, unambiguously, sharing assumptions, and performing what in pragmatics we could characterize as adopting Grice's "cooperative principle" (20). This principle is divided into four maxims: i) give all and only the necessary information, ii) give only true information (not false or doubtful), iii) maintain focus and relevance, and iv) do not use obscure or ambiguous expressions . The problem is that this principle is "pragmatic" and not "semantic", that is: it totally depends on the context, purpose and participants in the conversation. The principle is general, but the result of its application is always contingent. Therefore, these expectations of participants point to the importance of API design being as HCI design has already learned to be: sensitive to many usage context variations. However, this requires that the communication protocol between the user and the designer's "proxy" be enriched to a level in which it would be possible to infer or express the specific context of conversation. In the case of our participants, as some of them do not want to configure / parameterize the API, they clearly expect that the API will "infer" the context of use. However, to support this inference, we need to have clues, and thus to be more conversational.

2.4.2 SigniFYIng APIs

The inspection with the SigniFYIng APIs method is based on the selection of scenarios with potential communicability failures, which support the analyst's reflection throughout the method's steps. Since the Java 7 and 8 APIs were analyzed individually, we describe these scenarios separately.

1) Java 7 (Calendar)

In the Calendar API, we found four scenarios of communicability failure regarding the operations with dates already mentioned. In this article, we focus on two scenarios that we find most relevant, which have the potential to cause the most serious communicability failures.

The first scenario is related to the expected result when adding a month to a certain date. In the Java 7 Calendar API, the intended method for this operation is the "*public final void add (int field, int amount).*" The presumed intent of the designer is to provide a single method capable of performing arithmetic sum operations on the various fields of a date object. That is, we

see here an option for a homogeneous arithmetic treatment of Date and Time objects, which clearly disregards pragmatic aspects, agreed by local cultures. For example, in the banking field the deadline for payment of bills due on weekends can be conventionally postponed to the first working day after the date. Therefore, monthly installments will have deadlines that do not result from an arithmetic about day, month and year, but rather a contextualized interpretation of the meaning of monthly payment. If the API designer's intent is for the user-programmer to understand it, he must "communicate" that his decision should be supplemented by extra programming every time an arithmetic interpretation of the operation over dates is pragmatically inadequate. Pragmatically, SigniFYIng APIs points to the risk of "Unconscious task failure" if the user-programmer does not make by his own initiative an appropriate corner case test, which is equivalent to a pragmatic resiliency test of his use of the API.

The other scenario, which is closely linked to the previous one, is based on the difference in behavior of API methods for the same ambiguous situation. In addition to the "*add*" method, another method that can cause ambiguous situations is the "*public final void set (int year, int month, int date).*" Imagine that the parameters values are year = 2016, month = 1 (in the Calendar API, month value is zero-based, so February is represented by integer 1) and date = 31. In this case, the API could trigger an error. However, the Java 7 Calendar API is lenient and attempts to fix certain "invalid" dates automatically. The user-programmer, already accustomed to how the "*add*" method handles ambiguity, might expect as a result the date February 29, 2016. The API, however, returns the date March 2, 2016. Once again, we have a potential case of "Unconscious task failure."

2) Java 8 (LocalDate)

In LocalDate API, we find two relevant scenarios of communicability failure regarding the operations with dates previously mentioned. The first one is similar to the first item reported from Java 7, the expected result when adding a month to a certain date. In Java 8, the method responsible for this operation is "*public LocalDate plusMonths(long monthsToAdd).*" Although it is much more communicable than in Java 7, cases of ambiguity still persist. Arithmetic metaphor is still in place. Therefor, corner cases will generate the same kind of communicability failure, which we classify as "Unconscious task failure".

Another scenario of communicability failure occurs due to a feature added to the Java 8 LocalDate API, immutability. Let us make it clear that

immutability by itself is not a problem, on the contrary, it is something desirable in many situations. What we will point out in this scenario is the potential failure of the API to communicate to the user-programmer that this behavior is in place. When we examine the method "*public LocalDate plusMonths (long monthsToAdd)*", we can assume that the API designer expects the user will correctly interpret the existence of immutability when he realizes that there is a return value of the *LocalDate* type in the function (the same class that owns the method). However, this kind of return value is also used for other common purposes like, for instance, to allow the chaining of consecutive method calls. We can then imagine that the user-programmer does not understand that the object is immutable, at first. After all, the previous version of the date-related API was not immutable, and the vast majority of the APIs present in the standard language pack are not, either. Thus, when calling the method, the programmer could expect that the object itself would be changed "in-place", which would not happen. For this kind of failure, we can again assign the "Unconscious task failure" tag as a classification.

2.5 Discussion

2.5.1 SigniFYIng APIs and Interview Data

When we contrast the results found in the two stages of the study, we can observe that some of the results of one phase reinforce those of the other, that is, they say the same in different ways. The communicability problems identified through the SigniFYIng APIs method, somehow, also appeared in the participants' statements. For example, the ambiguity in add operations was diagnosed with a communicability failure in the analytic part. This failure is corroborated by the results of the quiz carried out during the interview, since only one or two (maximum) of the seven participants were able to anticipate the correct value returned by the operation of adding one month to January 31, 2016. This contrast shows the potential of the SigniFYIng APIs as a tool to guide the analyst's reflection in order to anticipate relevant and real problems.

The study also showed aspects that SigniFYIng APIs could anticipate to API designers. An example is the communicative cost of implementing a more formal and algorithmically efficient semantics if it somehow contradicts the signs of the user-programmer's programming culture. In such cases, the chances that the designer's communication with the user through his/her proxy (API protocols) will go awry are very high. Consequently, there is a great risk

that the API as a software development tool will not be secure or adequate.

Analytically, we realize that a detailed reading of the API documentation is required for the programmer to presume the designer's intentions in a satisfactory and effective way. Contradictorily, during interviews, participants were averse to large readings and extensive documentation. Simplicity was pointed out as desirable. We believe that a more dynamic interaction, as in the traditional interaction with software, is the solution to this problem. We need to make programming interfaces that can develop dynamic dialogues with their programmers.

While recent work based on cognitive theory (CDN) or heuristic rules (Nielsen's Heuristics) focuses on the problems of user-programmers and / or APIs, our semiotic analysis with SigniFYIng APIs brings together user-programmers, APIs, and designer-programmers. Moreover, through a software-mediated human-communication logic created by one party, our approach points to pragmatic conditions of reciprocal understanding between human parties when mediated by the software artifact. We explicitly address design issues that must be raised during the process of creating the API, in which the designer is asked to think about the conditions that he offers through his proxy, so that his human interlocutor solves ambiguities, understands design decisions and, ultimately, benefits from the contributions that the API is offering for the user-programmer's work as a developer.

As discussed in the results section, the main conclusion of the study is the fact that APIs with some degree of comprehensiveness (like the ones we analyzed) should be more "conversational" in order to achieve pragmatic adequacy. The claim of today's programmers about documentation load and parameterization makes us assume they are referring to the "interaction" style required for dealing with these today (which is roughly equivalent to the style that early system users had available in the early days of HCI studies in the 1970s and 1980s). Therefore, the semiotic approach is not strictly looking at the usability of the APIs, but rather to how communication with the APIs can or should unfold. That is, we are looking at another aspect of HCI, which complements that of the studies done so far.

A more conversational API would be an API that can communicate with its interlocutor in a more interactive way, giving more feedback as the interaction occurs. Currently, the interaction between programmers and APIs happens at two different times. First, in a static way, with little feedback from the API, when the programmer encodes his or her program. At this point, the programmer interacts only with the interfaces and documentation, and it is only possible to infer the internal behavior of the API from these

elements. Second, the programmer can test his code, and consequently the API. In this more dynamic phase, the API manages to give more feedback to the user-programmer. With a set of input and output data, or through error messages, the programmer can more effectively infer the internal behavior of the API. However, we believe that if there were a more dynamic interaction at coding time, the user-programmer would be able to presume the intentions of the designer in a faster and more accurate way. No doubt, creating dynamic interactive APIs is a big challenge, but it is probably the best way to handle those aspects of communicability. We suspect that, in order to implement these features, changes go beyond APIs. Development environments and programming languages play a key role in this issue.

2.5.2 Evolution of the API, from Java 7 to Java 8

Through the inspection with SigniFYIng APIs, we could observe the evolution between the two Java APIs for date and time. Although in this study we did not inspect the entire Java date and time APIs, we noticed that some communication failures were corrected in-between versions (and which ones) while others identified with SigniFYIng APIs persist in the new version of the API. We notice that problems of inconsistency have been solved. In Java 8, when you try to create an invalid date, the API reports an error. In Java 7, in addition to not accusing the error and being lenient, the associated ambiguity produced different results than when using the "add" method.

Despite great and remarkable advances, we still identified a serious communicability issue in the new API. The operations on dates continued to use metaphors based on the addition and subtraction arithmetic. The API still communicates poorly its behavior in cases of ambiguity. The API documentation explains, in its way, how it behaves in such cases. However, there are no good examples of corner cases and we believe that an arithmetic metaphor with date operations may still not be the best approach to communication. Think of a sequential metaphor, for contrast, using "next" rather than "add".

Based on the method, we have two implications in this comparison: i) Some problems that we have detected, and which have remained (at least until now), even with the evolution of language, leads us to think that there is a class of problems that cannot be solved without considering pragmatic aspects that permeate communicability between users and designers; ii) Problems that we detected with the method and that were solved suggest that the method can be used as a tool to anticipate and understand problems, providing favorable conditions for better API design. Therefore, we believe that with this study,

we show the value of the SigniFYIng APIs method.

2.5.3 Implications

We believe that our work has the potential to influence in some aspects. For Software Engineering, for example, diagnosing communicability problems in APIs can have a major impact on the quality, cost and efficiency of building software that uses that API. Even simpler problems of communicability, which in theory, would be easily circumvented, can at least bring inefficiency to the process. More severe communicability failures, such as those pointed out by the method in the previous section, can affect software reliability. If a programmer does not notice the failure at coding time, surely at some point this will appear in the final product.

The results from our study, similarly to other researchers' results, indicate that APIs have a lot to evolve, which motivates further research on API design and use. However, there is also room for improvement in programming languages and tools, in order to provide better support for APIs. To make APIs more conversational, for example, we may need tools that allow dynamic interactions between user-programmers and the software artifacts.

2.5.4 Threats to Validity

Because this is not a predictive study about correspondences that should be always found between SigniFYIng APIs results and programmers' perceptions and understanding, the small domain scope of analysis and the small number of interviewees does not seem to be a great threat to validity. However, the fact that our two main study blocks occurred in parallel can threat validity. Although we have been cautious to control rigor, we know that there is the possibility of contamination of the results. A problem of communicability found in the interviews stage may have biased the analysis with the SigniFYIng APIs method. However, since SigniFYIng APIs is based on a solid theoretical background that supports an analyst's deeper reflection during the inspection (meaning that a good analyst might anticipate many, although not all, of the issues brought about in the interviews), we believe that our findings are still sufficiently valid. However, in order to evaluate and, if needed, remedy a validity problem with this study, we intend to carry out new research with some methodological adjustments. In next and final section, we describe such adjustments and the work we wish to accomplish in the future.

2.6 Conclusion and Future Work

This is the first of a series of studies that we wish to undertake regarding the use of the SigniFYIng APIs method. We know this study was limited by methodological and domain issues. Thus, in the future we intend to carry out new work in different domains of APIs and programming languages. A methodological evolution, as seen previously, is also necessary. For further studies, we will adopt new strategies to eliminate possible interferences between analytical and interview parts.

Other studies that we find relevant are the ones that guide us to the final software generated from APIs. We believe that identifying communicability failures in an API can be a way to improve the quality of a software artifact, for example, by generating fewer *bugs* in the final product. Therefore, we intend to carry out exploratory studies in an attempt to relate failures of communicability with the final quality of the software product.

In this article, we show that it is possible, from an analytical method, to identify communicability failures in a software API that may have undesirable effects on the user-programmer's coding activity. In addition, we collected evidence through interviews with experienced programmers, which supported our analytical results. This leads us to believe that the method used is an effective tool to support this kind of investigation and provide relevant findings. The results of this article demonstrate that a semiotic view of API design and use may also be valuable to the software development community. Although we acknowledge the value of HCI usability methods used in this new context, we believe that our approach has the potential to demonstrate problems that go beyond those of cognitive nature and which are not currently captured by other methods.

2.7 Summary of Chapter 2

This chapter shows a study made with professional programmers regarding the inscription of meaning in date API interfaces (section 2.3). Crossing with empirical data, we demonstrate how a method of analysis, based on Semiotic Engineering (13), can benefit the inspection of software APIs in the search for problems that may compromise their use. This type of inspection evaluation can support API design by pointing out problems that could be corrected before the API be offered to a third party. Through this study, we also demonstrate that an API with more interaction, or as we define it, conversation, can benefit its users (section 2.5). We were able to identify communicability prob-

lems in APIs that directly impact their use, and consequently, the final quality of the generated software.

This study was the first carried out in the context of this Ph.D. thesis and served as motivation and guide for the following activities. This first study inspired us to establish the concept of conversational API. Although the study demonstrates the need for API conversations, it does not discuss in depth how these conversations would take place and what levels of conversation an API could offer. Therefore, we needed a better characterization, not only in the concept but also in the levels of conversations that an API could offer (section 2.5). Thus, we base our research into the theory of Semiotic Engineering and use the concepts of signs established by the theory to define our conceptual framework for characterization and classification of conversational APIs. This conceptual framework is demonstrate in detail in the chapter 3. This study also does not offer any tools or techniques to support API design process. Although we have used an inspection method (section 2.3.1), it cannot be directly transported to the API design process. So, this study also served as a motivation to seek and develop a method to support the design of conversational APIs, which we show and discuss in chapter 5.

3
A Conceptual Framework for Conversational APIs

A conceptual framework is a set of concepts, tasks and techniques, used to characterize and classify a problem in a specific domain. In this chapter, we present a conceptual framework to characterize and classify conversational APIs. A conversational API is one that is able to communicate to its users its form of use and its internal logic of operation, making clear the design rationale when abstracting concepts and tasks (3). As we saw in the chapter 2, the lack of conversation on an API can lead users to not understand how to use it properly. So, we needed a conceptual and theoretical definition on how to build a conversational API, which we discuss in this chapter. As we will discuss in this chapter, every API has some level of conversation. However, in this thesis, we aim to support the design of what we call as fully conversational API.

When creating an API, the designer has three different ways to send his message to his user: the source code, documentation, and API behavior. This chapter discusses how to classify a conversational API according to the three types of signs proposed by Semiotic Engineering (13). This is a conceptual work that aims to provide theoretical and conceptual support in the characterization and classification of conversational APIs (section 3.4). We also present and discuss an evaluation of the framework on top of APIs from different domains (section 3.5). Such a conceptual framework can be useful for API designers to understand the concept of signs and the possible distributions of the conversation through them. So, the API designer can decide whether or not to adopt the conversation at different levels when designing their interfaces. For example, in the APIs discussed in chapter 2 (section 2.4), a designer informed by our conceptual framework would realize that putting the meaning of an operation only in the API documentation could cause user misunderstandings.

This chapter presents an extended version of a paper accepted at *Brazilian Symposium on Software Engineering (Qualis A3)*, entitled *"A Conceptual Framework for Conversational APIs"* (4). The theoretical basis presented (section 3.2) is the same as in the previous chapter (section 2.3), but it covered additional details focused on the context of software APIs. The reader may

want to skip the introduction since it is almost the same as the already presented in chapter 1.

A Conceptual Framework for Conversational APIs

João Antonio D. M. Bastos
PUC-Rio
Rio de Janeiro - RJ, Brasil
jbastos@inf.puc-rio.br

Rafael Maiani de Mello
CEFET/RJ
Rio de Janeiro - RJ, Brasil
rafael.mello@cefet-rj.br

Alessandro Fabricio Garcia
PUC-Rio
Rio de Janeiro - RJ, Brasil
afgarcia@inf.puc-rio.br

3.1 Introduction

Application Programming Interfaces (APIs) are software components designed to ensure the portability and reuse of a source code. APIs can be developed in several programming languages, offering functions to support the execution of activities in specific domains. Recently, we have seen a growing demand for API development, especially in terms of supporting the execution of complex activities. Consequently, we can also see the increasing use of APIs by non-programmers. These professionals, experts in various areas such as medicine, pharmacy, geology, and others, are usually involved in projects to create predictive models for the most different contexts (9).

Although API designers can expect artificial intelligence professionals to have some computing skills, they cannot expect from these professionals extensive knowledge and experience in software development. Therefore, the APIs used by these professionals should provide adequate interfaces to abstract the complexity of their programming (8). Besides, API users must be able to perceive the design rationale behind API interfaces in order to use them correctly.

However, designing an API that explains the design rationale of who designed it is not a trivial task. One way to accomplish this task is by creating conversational interfaces (2). Such interfaces, which we will address in detail in this article, are those which promote pre-established potential dialogues for the API to have with its users. The lack of explicitly defined conversational capabilities hinders API users in understanding how to properly use their interfaces (section 2.5).

In this link[1], we can see a real example of an API with insufficient conversational capabilities. In this example, a user of the *Google Maps* geolocation API reports a failure to calculate the distance between two locations. The user of the API claims that he is requesting the result of the distance calculation in miles. However, the answer provided by the API is always with the distance calculated in meters. Surprisingly, the API designer has re-classified this *bug* to an *expected behavior* by transcribing the following excerpt from the API

[1] https://issuetracker.google.com/issues/35829619

documentation: "Note: this unit system setting affects only the text displayed on the distance object *(distance.text)*. The distance object also contains values that are always expressed in meters *(distance.value)*".

Misunderstandings between designers and API users are common. The report described in the preceding paragraph exemplifies how the geolocation API is incapable of getting users to understand its *design rationale*. Unfortunately, the documentation is often perceived by designers as sufficient and mandatory to support the proper use of API interfaces. However, issues like this can be avoided by improving API conversational capabilities. Therefore, we believe that developing APIs for different domains would benefit from a design process aimed at specifying and improving conversation capabilities.

Problems caused by the lack of conversation of an API often occur in other contexts than the example mentioned above. Previous studies report similar problems with the design of APIs for reflection in the *Java* programming language and with the design of APIs for *IDEs* refactoring (38). These studies have detected problems caused by the lack of API conversation. However, as far as we know, there is no proposal in the literature on how to treat this problem. Our approach uses semiotic-based theories to provide tools that are able to promote software APIs conversation.

In this work, we present a conceptual framework to define and to classify conversational APIs. As we present in detail in this paper, a conversational API is the one capable of communicating to its users its form of use and its internal logic of operation (2). After a detailed description of the theoretical basis, we list the different levels of conversation and ways to send the communication that a software API can present. This is a conceptual work that aims to provide theoretical and methodological support in the definition and classification of conversational APIs. We also present a study where we make the analysis of three APIs regarding the levels of conversation that we propose in the conceptual framework. This result can help API designers understand the proposed concept and apply it in the process of developing their APIs

In the following sections, we will describe the theoretical basis of our research (3.2); present how to introduce conversation into APIs (3.3); show our conceptual framework with the signs and levels that an API can present (3.4); represent, through real cases, APIs at different levels of conversation (3.5); discuss the impacts of the concepts presented here (3.6); briefly discuss the related work (3.7) and finish with the conclusion and future work (3.8).

3.2 Theoretical Basis

3.2.1 Semiotic Engineering

Semiotic Engineering is a theory of Human-Computer Interaction (HCI) built based on semiotics (13), a field of linguistics focused on the study of signs, meanings, and processes of meaning. In particular, Semiotic Engineering is strongly guided by the theories of Charles Sanders Peirce, one of the founders of Semiotics as a field of study. The sign, central element in semiotics, is defined by Peirce as anything (for example, the source code or the documentation of an API), that in a certain aspect or way, means something to someone (40). For Peirce, the existence of a sign is necessarily associated with the existence of an interpreter. That is, the meaning of the sign only exists through someone's interpretation.

Signs: Semiotic Engineering classifies signs into three different categories: Statics, dynamics and metalinguistics. *Static signs* are signs that are interpreted independently of temporal relations and user interactions with the system. They are signs that must be interpreted without the user having to interact with the system, such as buttons, menus and toolbars. *Dynamic signs* are those in which the temporal relations are fundamental for the correct interpretation by users. The interpretation of a dynamic sign is necessarily associated with a process of user interaction with the system. We can cite as an example, the interaction process of saving a new file.

Thus, we can say that static and dynamic signs are part of the same whole. While the static invites the user to perform an interaction based on his current interpretation, the dynamic is the result of that interaction, which can confirm the user's current interpretation or revoke it. Finally, we still have the class of *metalinguistic signs*. These are signs that speak about other signs. This category has the purpose of explaining, through a metalanguage (natural language, for example), the meaning of static and dynamic signs (e.g.: help menus, documentation, tooltips).

Semiotic Engineering and Software Engineering: Semiotic Engineering has expanded to support studies and reflections beyond the classic Human-Computer Interaction. In this sense, the expansion towards software engineering is due to the increasing need to go beyond the end-user interface of software when evaluating the human-computer interaction. Several problems found in the end-user interface of software can be identified and remedied during the development process. Problems that appear to the end-user are often problems that should have been identified during various phases of the project, such as the specification, modeling or coding of software. Semiotic Engineering has tools and methods that allow us to look at various artifacts used in the

process to understand the communication problems that can occur. For example, an API uses misinterpretation may have caused an error that appears to the end-user.

The process of developing software can be seen as a process that, besides being logical and specified, is also temporal. The ways of users and designers to think and rationalize about a given computing artifact change with time and new information. Thus, in order to investigate a phenomenon of this nature, we need theories that support these temporal changes in the behavior of users and designers. Semiotic Engineering, for being based on semiotic theory and has as basis the abductive process of reasoning, has the necessary to offer tools to investigate these phenomena.

Metacommunication: Semiotic Engineering (13) characterizes the interactive process as a particular case of human communication mediated by computers. In this vision, communication occurs between users and designers through a computer artifact. The designer can then perform his communication through the interfaces designed by him, which are named *designer proxy*. It is then up to the user to interact and communicate with this interface in order to achieve his goals.

Therefore, the process involved in the communication between designers and users is the central point of investigation of the Semiotic Engineering theory. We say that during this process, a phenomenon called *metacommunication* occurs, in other words, the communication about the communication. While communication deals with the message sent from the designer to the user, metacommunication deals with the message sent to the user about how he should interpret the original designer message. According to the theory, when creating a software artifact, the designer defines a complete and unchanging message, which will be transmitted to the user through the designed interfaces.

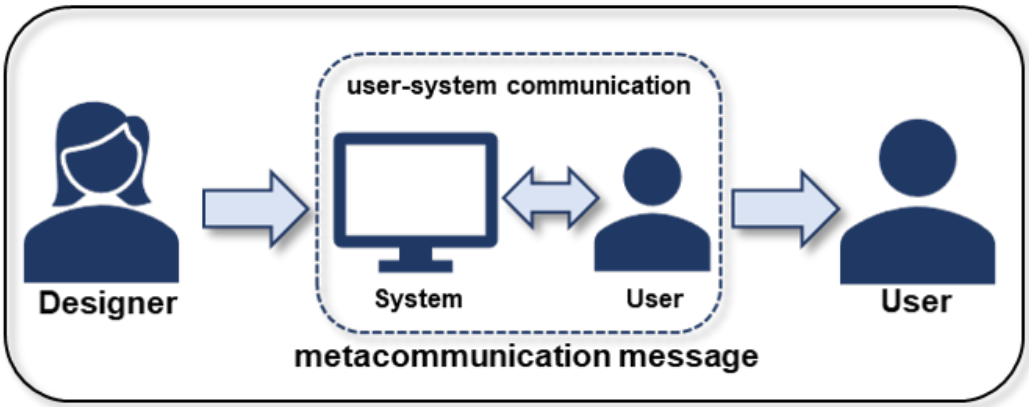

Figure 3.1: Semiotic Engineering and Metacommunication

Figure 3.1 represents the message exchanges between metacommunica-

tions participants, that is, the designer and the user. In interaction time, the designer is virtually present, being represented by the interfaces he created. On the other hand, the user can use a set of commands and interfaces designed to send his message. When performing this process, we say that a metacommunication message is being exchanged between users and designers of the computational artifact.

Communicability: the central quality of metacommunication is the *communicability*, which may or may not be present in a computer artifact. We can say that an artifact has good communicability when it is able to communicate to its users the design logic of its designer. In practice, the presence of communicability allows the user to have a more complete set of information to help them decide which is the best way to use the artifact.

For communicability to be effective, it is up to the designer of a technological artifact to establish and define the signs that will be used during user interaction with the artifact. When we talk about APIs and programming languages in general, we are certainly limiting the scope of signs we have at our disposal when creating our artifacts. When designing an API, the designer does not rely on icons, sounds, and images in a general way. Therefore, it is up to him to pass his message effectively using only texts, patterns, and reserved words of the programming language used.

3.2.2 Abductive Reasoning and Semiosis

At the moment when communication is taking place between designer and user, an exchange of signs is happening. When a user perceives a sign and tries to interpret it, he generates his own understanding of what that sign means to him. This process meaning creation is called semiosis (45), and is totally influenced by our context, world knowledge, culture, and experience. As an immediate consequence of this characteristic, we realize that the same sign can awaken different interpretations, depending on the users and contexts of use involved.

Moreover, the theory also defends that the context, knowledge and experience of the user can modify over time, making it also modify its current interpretation on that sign. This temporal process of modifying the current interpretation of meaning is called unlimited semiosis and is built upon the idea of abductive reasoning (45). In abductive reasoning, a current interpretation is not an absolute truth, but a causal relationship, which can be modified from the moment new information about that interpretation emerges with time. In the figure 3.2, we can see how a possible abductive process of a date API users

works. We use the *"add"* function of the Java 7 Calendar API to illustrate.

We can imagine the following scenario (see figure 3.2): (i) the first user interpretation is the simplest. If the user asks to add a month to the "Aug-15-2020" date, the API will return "Sep-15-2020". At this point, the user is satisfied with its rationalization on the internal behavior of the API - The API only changes the field "month" by adding the value in the parameter. At a given time, the user wants to add a month to *"Aug-31-2020"*, and the API returns *"Sep-30-2020"* as a result. With this, (ii) the user redoes his rationalization on the API's internal behavior - The API does not just change the month field; it adds 30 days to the date. So, possibly, the user can perform another operation. Now, when adding a month to the date *Jan-31-2021*, the API returns the date *Feb-28-2021*. Thus, (iii) the user redoes its rationalization on the internal operation of the API - The API changes the month field and "rounds" the day to the end of the month when it exceeds it. This abductive process can be repeated indefinitely, causing the user to redo its rationalization on the internal behavior of the API.

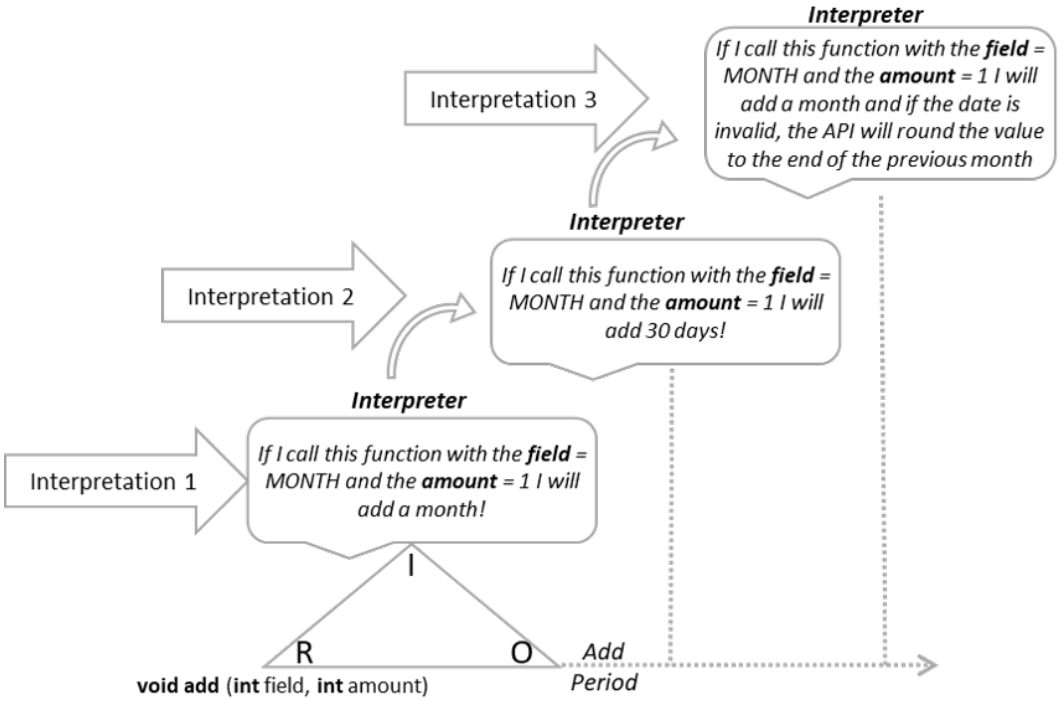

Figure 3.2: Abdutive Reasoning and Semiosis - Example with Java 7 Calendar API

By analyzing the situation illustrated above, we can say that due to a communicability failure, the user is led to create a new interpretation about the meaning inscribed in the *"add"* sign. This process can be infinite, leading us to the concept of unlimited semiosis. Thus, with each communicability failure

that occurs, an opportunity for an abduction process is opened, make the user go into semiosis, and start to understand better what message the designer was trying to send when creating that interface.

3.2.3 Conversational Interface

An interface is conversational when it has the ability to develop a conversation with its interlocutor in a particular language. In this sense, we can classify a software as conversational when it offers interaction resources that can be represented as an exchange of dialogues between its designers and its users. Within the context of human-computer interaction, we have some decades of research that seek to understand the best way to design software, so it offers the most effective and efficient conversations for its users. Among these theories, we highlight Semiotic Engineering (13), which we described in the previous subsection as a semiotic based theory. This theory sees the interaction process as a communication between designer and users, using the software as the means to enable this message exchanges.

During the interaction, the designer, represented by the interface, is talking to the user through the signs that he thought most appropriate. This conversation evolves with each new interaction, and the user has the opportunity to reinterpret when he thinks it is necessary, the speech sent by the designer. This conversation between user and designer is the predominant factor so that a semiosis process can occur. As the user receives a new message from the designer and uses this message to re-evaluate his or her current interpretation of a particular aspect, he or she is performing a semiosis process.

3.3 Introducing Conversations in APIs

An API can be seen as a mediator of the conversation between two programmers. On one side, we have the designer programmer, a term we will use from now on to refer to the creators of the API. On the other side of the conversation, we have the user programmer, which we will also use to refer to the users of the API. In this conversation, the metacommunication that occurs between programmers must be effective and efficient to make clear the designer intentions, so the user does not misinterpret and misuse the programming interface.

3.3.1 Syntax, Semantics and Pragmatics

The user's understanding of a given API can be analyzed through three layers of abstraction referring to the interfaces of this API: the *syntax, the semantics and the pragmatics*. The first level is the *syntax* of the interface, where the programmer needs to understand the syntactic ways of the programming language in which that API was developed. This includes understanding its formation rules, reserved words used, among others. In this first layer, the user can identify the API vocabulary and the types of data used as parameters and feedback, and understand if they are in accordance with the programming language used.

After that, the user passes to the *semantics* of the interfaces. Through a careful reading of the documentation, an analysis of the names, returns and parameters of the API methods, the user can infer what meaning and behavior is expected by that API. A good API, with extensive documentation and with method names and parameters rich in meaning, should allow a good understanding of its use by the user.

The last layer is the *pragmatic*. This layer is directly associated with contexts and with the concrete use of those meanings that had been previously inferred in the semantic stage. It is at this stage that the user will realize if that semantic meaning previously made is really adequate to his context of use, his culture and his way of acting and thinking about that domain in which the API is inserted. That is, semantics is the meaning inferred, while pragmatics is the use of that meaning in practical and concrete situations, where besides semantics, we also have to take into account the interlocutors (users and designers) and the context of use.

Let us take as an example the two methods below, taken from the API *Calendar*, from the programming language *Java 7*, and the API *DateTime* from the programming language *Java 8*:

```
public abstract void add(int field, int amount)
```

Listing 3.1: Java 7 Calendar API

```
public LocalDate plusMonths(long monthsToAdd)
```

Listing 3.2: Java 8 DateTime API

We believe that this API illustrates in a simple way the concepts that we are presenting and defending in this article. We know that manipulating dates is a challenging activity for software developers. The concept of date is

something abstract, invented by humankind, and not part of nature. Thus, date is something that varies significantly among different cultures and contexts, with different calendars and ways of representing them. Even in cultures that follow the same calendar, as in the case of the Gregorian calendar in western Christian countries, we have a wealth of details that usually make the task of programmers difficult. For example, time zone, daylight savings time, date and time representation format, are some examples of features that vary within the same calendar.

When we manipulate date, even in our own culture and calendar, we can have contexts of use that make a certain operation possible in many different ways. Returning to the code snippets above, how can we interpret what the operation of "add a month to a certain date" consists? The choice of appropriate syntax will vary according to the programming language and the programming environment that the API is associated. While the first API chooses the syntax *"add"*, the second one presents the syntax *"plusMonths"*.

In the semantic layer, the user tries to understand the meaning of the interface. The interface of the API *DateTime* from *Java 8* certainly presents a more informative semantics than the API from *Calendar* from *Java 7*. In the first, we have the term *"plusMonths"* which carries with it the semantics that a number of months will be added to the date on the object on which it is being operated. Clearly there is a leap in quality from its predecessor. In the API *Calendar*, we have the method *"add"*, which alone, without a careful reading of the documentation, does not have an adequate semantics about which operation will be performed when calling the method. Even so, the user, when appealing to the documentation, will be able to understand the semantics of the method.

It is on the third layer, the pragmatics, that the problems appear, and for both APIs. The operation of "add a month to a certain date" (*add* or *plusMonths*) is not like a mathematical operation, where regardless of the context of use, the result is absolute. In most cases, where the API does not have to face any decision making, the operation is basically mathematical. For example, if we add a month to January 15, 2020, it is clear and well established that the expected result is February 15, 2020. However, it is in special cases that pragmatics comes into play. For example, what would be the expected result when adding a month to the date January 31, 2020?

Answering the above question is a difficult task and we will certainly not find a single answer that is acceptable to everyone. To get around such a question and clarify possible doubts, we can use the conversational API concept.

3.3.2 Conversational API

Conversational API, in the most general and abstract sense, is that API that can expose its internal logic and provide the possibility to modify its behavior through dialogues with its user. The internal logic of an API is its designer's view of existing functionality, including how the API should or can be used. Thus, it is expected that the user, when interacting with a conversational API, will be able to perceive its functionalities and understand how it should be used. In addition, the user will be able to change the API's internal behavior and its way of use. In a maximum degree of conversation, the user can be considered as a co-designer of the API.

A conversational API must be able to meet the different users and their contexts. The API can have a standard behavior on how to react in a decision situation like the one presented above. In such cases, this internal logic has to be quite explicit through the dialogues that the user will have with the API during use. In the last case, the conversational API must be flexible enough for the user himself to express his logic for the decision making represented above.

Thus, we can derive the two main concrete characteristics that a conversational API should have. The first is the ability to explore the principle of cooperation between users and designers. This means that an API must go beyond semantics, also acting in pragmatics. The second characteristic of a conversational API is its *customization capability*. Basically, this means that the user can adapt its API when the usage scenarios offered by the designer do not cover a specific need. In the following subsections, we describe in more detail the concepts related to the main features of a conversational API.

3.3.2.1 Principle of Cooperation

The principle of cooperation is a concept proposed by Grice (20), where the information sender must take care to interact with the information receiver in the most complete and explicit way so that all messages being correctly interpreted. According to Grice, this principle is achieved from the four conversational maxims: the maximum of quality, the maximum of quantity, the maximum of relevance, and the maximum of mode. The *maximum of quality* says that the interlocutors must present only true information, avoiding false or doubtful statements. The *maximum of quantity* says that interlocutors must present all necessary information, but without exaggerating with information irrelevant to the context of the conversation. The *maximum of*

relevance says that interlocutors should maintain focus and relevance, i.e. only present information focused on the context of the conversation. And finally, the *maximum of mode* text says that interlocutors should avoid the use of ambiguous expressions when transmitting the information.

Respecting the principle of cooperation reveals that the speaker has sufficient conversational competence to interact according to the requirements of successful conversational exchange. It is the respect for the principle of cooperation that makes it possible the development of the discursive interaction, in which the speakers participate to proceed in accordance with the specific purpose and direction of the statements. For Grice, irony, the ambiguity of discourse and metaphor, by violating certain conversational maxims, also violates the principle of cooperation.

Such conversational maxims should be used as a guideline during the design process of a conversational API. Avoiding ambiguous language and the use of metaphors is fundamental for the designer and user to understand each other during the process of using an API. Furthermore, providing all and only the necessary and relevant information for the emission of the desired message through the API interfaces is fundamental for the conversation to take place effectively and efficiently. For this, it is necessary that the designer, when designing and implementing his API, can create as many usage scenarios as possible. Thus, he will have enough information to implement the necessary conversations, respecting the principles of cooperation described above.

In this sense, the programmer designer can make use of the scenario-based technique. Scenario is a computer-human interaction technique widely used in the traditional software design process (43). This technique involves imagining and describing the possible interaction scenarios that a potential user needs to perform in the system. These scenarios are a narrative, textual or pictorial, concrete, rich in contextual details, of a situation of application use, involving users, processes, and actual or potential data (43). A scenario must contain the environment or context of application use, the definition of who the users are (their characteristics, motivations, objectives), and the sequence of actions and events that take place in the unfolding of the narrative, which must be concluded with the user's perception about the success or not in achieving its objective.

3.3.2.2 Customization

The other key feature in a conversational API is the *customization*. As significant as the semantics of an API may be, it is not possible to meet

all the pragmatic contexts (scenarios) of use. Therefore, even if the designer has detailed several scenarios during the API design process, it is virtually impossible to meet all users' needs without customizing the internal behavior of an API. The customization can be on several levels, from the simplest, such as changing a configuration parameter about a certain behavior, to the most sophisticated, such as the complete modification of a certain method by another created by the user.

Thus, we can use customization to modify the internal behavior of an API, using dialogues with the user in order to modify the internal logic. For example, in the Calendar API example, we could add a new parameter to the *"add"* method with the name *"roundType"*. This parameter would give the user the opportunity to communicate what type of rounding he would like to perform, within his context of use, when the operation of adding a month would result in an invalid date. However, a single type of adaptation may not solve all cases. A user of an API might, for example, want the result to ignore working days or weekends. Several other adaptations may be needed to the method, which would imply that the API provides a way to customize its internal behavior.

3.4 Conceptual Framework

In the previous section, we present the concept of a conversational API. We established that every user interaction with the API could be considered a communication process. Its efficiency is measured by the satisfaction of the parties involved in the process (designer and user). Thus, the key to designing a good interface is in the conversation. In this section, we will present our conceptual framework for defining and classifying the conversation levels that an API can achieve. For that, we will explore the concept of sign defined by semiotics and its separation in three classes as Semiotic Engineering defends it.

3.4.1 API Signs

Unlike the traditional human-computer interaction process, where we have several types of signs (graphic, textual, sound, engines, etc.), in APIs and programming languages interaction process, we are limited to textual signs (function signatures, documentation, error messages, etc.). In this section, we will unravel these different types of textual signs according to the time the user interacts with them. We will classify them according to the original logic of

Semiotic Engineering: Static, Dynamic, and Metalinguistic. After that, we will describe the conversation levels that an API can reach from each signs types.

Static signs: according to the Semiotic Engineering definition, a static sign is one that the user can infer meaning without needing an interaction. In the case of software APIs, we can consider static signs the vocabularies used in method signatures (name of a function and the parameters), the types of data (types of parameters and return types), and the structure (packages and classes). These are elements which the user does not need to interact in order to infer any meaning. When faced with the method signature below, a Java experienced user would be able to infer that this function serves to add the amount of months passed by the parameter *"monthsToAdd"* to the date object used as a reference in the function call.

```
public LocalDate plusMonths(long monthsToAdd)
```

Listing 3.3: Java 8 DateTime API

Dynamic signs: the dynamic signs are those where the user can only infer meaning after some interaction with the object, in our case, the API. This category includes the signs related to the internal behavior of the API (the result generated at each execution) and the return messages (messages indicating success or error in the call execution). The dynamic signs can be in accordance with the meaning inferred by the static sign, indicating to the user that he is in the correct path. Alternatively, the dynamic signs can go against the meaning inferred by the static sign, opening the user's way to enter a semiosis process and create a new understanding of how to use the API. In the same example listed above, we can imagine the API user creating the following code snippet.

```
/* Jan-01-2020 */
LocalDate myDate = LocalDate.now();
/* Add 1 month */
myDate.plusMonths(1);
/* Print Jan-01-2020 */
System.out.println(myDate);
```

Listing 3.4: Java 8 DateTime API

Notably, the result represented above is not what the user expected. Thus, the interpretation of the dynamic sign goes against what had been interpreted in the static sign. The API user can then make a new inference and realize that when executing a method on the date object, a new object is created, and the previous object can be discarded. The user can then create

the following code below, and go through a new semiosis process to confirm or not his hypothesis.

```
/* Jan-01-2020 */
LocalDate myDate = LocalDate.now();
/* Add 1 month */
myDate = myDate.plusMonths(1);
/* Print Feb-01-2020 */
System.out.println(myDate);
```

Listing 3.5: Java 8 DateTime API

Metalinguistic signs: in the previous example, the user's interpretation is more easily inferred from an analysis of the third class of signs, the metalinguistics. Metalinguistic signs are the one which speak about other signs. That is, through them, the user can be explicitly communicated the meanings coded in the system. In the case of APIs, metalinguistic signs can present themselves in various ways, most commonly found in the form of official API documentation, or bug report tools and official API forums.

3.4.2 Conversational API Levels

Now that we have established the three types of signs that an API designer can use to pass their communication through an efficient conversation, we will present a classification of conversational APIs from the efficient or non-efficient presence of those signs in the APIs. In the next section, we will make a classification of a set of date and time APIs according to the levels proposed here.

3.4.2.1 Rudimentary Conversational APIs

From a Semiotic Engineering view, every interaction can be seen as an exchange of messages between designer and user. Thus, every computational artifact has some level of conversation. We will start our classification by a more basic level, which we call rudimentary conversational API. A rudimentary conversational API is one where the user and the designer cannot establish complex and continuous conversations. Communication failures frequently occur, either due to poor API design or inefficiency of the signs used in communication. As we will discuss in the next section, there are many APIs that fall into this category.

Another characteristic of an API with rudimentary conversation is the inability to provide complete dialogs with the user to perform core tasks in

the API's context. It is common to find APIs that transfer responsibility for performing a particular task to the user. For example, in the context of a date, it would be reasonable for the API to provide interaction dialogs where the user can convert a text to date (or vice versa). As we will see in the illustrative example in the next section, the absence of such functionality (or dialogue) impacts the quality of use and interrupts the flow of conversation between user and API.

3.4.2.2 Metalinguistic Conversational APIs

The second level in our conversational API classification is metalinguistics. At this level, we say that the interfaces and behaviors of the API are not enough to pass the designer's entire metacommunication message, and that for the user to make good use of and properly understand the design rationale and the internal behavior of an API, he necessarily needs to consult metalinguistic signs.

In this type of interaction, the user's speeches are sent through interaction with static and dynamic signs. However, the designer is only able to pass his message effectively through metalinguistic signs. As we will discuss in the next section, most of the existing APIs fall into this category. They are APIs that, although well designed, cannot establish a conversation with the user using only static and dynamic signs. The user always needs an extra explanation from the designer to understand how to use the API fully. This explanation is found in the form of documentation or through help forums.

3.4.2.3 Fully Conversational APIs

A fully conversational API is an API that can bring dynamic signs to encoding time. Static and dynamic signs, in general, are already enough for the designer to send his message. This is the level of conversation best suited for several types of APIs, especially those where cultural and contextual aspects can profoundly influence the abductive process and consequently, the user rationalization of the API.

This full conversation needs to be achieved whenever the API designer is in a decision-making situation that may be controversial. If a decision can be misinterpreted, then the designer must choose to add some signs in the interaction (be it static or dynamic). Dynamically, the designer can, for example, add some warning messages to the function call back. Alternatively,

in a static way, the designer can work with more meaningful vocabularies, even if this implies a more verbose API.

However, by adding much information in static and dynamic signs, the API may end up becoming very verbose and not very efficient during execution. Improving the API's quality of use can have a negative impact on other quality aspects of software, which should be properly negotiated. Alternatively, we believe the designer could think of two modes of operation for the API. One operation mode for when the users are building their code and another mode for executing a production code. Having two different source codes, one for encoding and one for executing, may seem strange, but it is already something widely used in software development. An excellent example is the CSS and JavaScript minify mechanisms (47).

3.5 Evaluation of the conceptual framework

3.5.1 Date and Time APIs Classification

This section describes an evaluation of our conceptual framework performed with date and time APIs from Java programming language. This evaluation demonstrates how our conceptual framework can be used and how different categories of conversational APIs can affect the quality of use and, consequently, the final quality of the software, with fewer API-related failures. We will use for this illustration two APIs known from the Java programming language. The API *Calendar* from *Java 7* and the API *DateTime* from *Java 8*. We chose to work with these 2 APIs because they are both from the same programming language and widely used by the Java programming community. Also, they are APIs with recurring problems reported on popular sites, such as *stackoverflow* and *github*. So we want to demonstrate that, although there is a clear evolution between the two versions of the API, there is still a gap of understanding when pragmatic issues arise.

As defined in our conceptual framework, we cannot fit any date and time API from Java programming language into the full conversational API category. So, at the end of this section, we will exemplify how a full conversational API can be instantiated for the context of dates. We will show only a small piece of a proposed API that can support a dialogue committed to meeting the user's needs of the example usage scenario. To simplify and focus on just one aspect of the API conversation, we will define a specific scenario to perform our classification. Thus, we define a real usage scenario for an API and then

present a possible conversation that represents the user interaction with the API to achieve the objective of the scenario described.

Scenario: an experienced programmer, who knows the Java programming language and is accustomed to using APIs, wants to use a library that helps him in some operations involving date type objects. One task he will need to perform is to calculate the due date of billing bills. To do this, he will receive a textual value representing a date, convert it to the date object of the API used, and perform the sum operation according to the business rule (1-month term). After the operation, he converts the date object back into a text and returns the value. This scenario is recurrently found in APIs with the following dialog exchange (U represents the user's speeches and API(D) the designer's speeches):

1: **U:** Convert this text "Jun-15-2020" into a date object.
2: **API(D):** Object successfully created.
3: **U:** Now, I wish to add a month to the date object.
4: **API(D):** OK, I made the calculation here is the result.
5: **U:** Now, convert this object into a text.
6: **API(D):** OK, successfully converted -> "Jul-15-2020".

The proposed scenario and the dialogues described above were intentionally designed to create a pragmatic situation that may conflict between the user's and the designer's understanding of the desired functionality. After all, we may ask ourselves, what is "add a month". Some might say it is add a unit to the month field of the date object. However, this can generate temporarily invalid situations. When doing this operation on the date "Jan-31-2020" for instance, we come across the non-existent date "Feb-31-2020". How should the API react in that situation? That is what we will see in the following three subsections: how the APIs *Calendar* and *DateTime* behave, and how we think a full conversational API should behave.

3.5.1.1 API Calendar - Java 7 - Rudimentary Conversational API

We look for the Java 7 *Calendar* API the signs that somehow refer to the proposed scenario. We can divide the scenario into two core activities: (1) convert date into text (and vice versa) and (2) add some period to a date object. These two core activities apply to the three APIs we are classifying here. In the first activity, the *Calendar* API does not offer any support, or, in words we are using in our work, the API does not provide a conversation on this matter. Converting date to text, an activity that is considered central given the context, is totally out of API scope, forcing the user to use code from

other APIs (SimpleDateFormat API) to assist him in his task. Regarding the add period activity, the API offers the following method:

```
/**Adds or subtracts the specified amount of time to
   the given calendar field, based on the calendar's
   rules. For example, to subtract 5 days from the
   current time of the calendar, you can achieve it
   by calling:
add(Calendar.DAY_OF_MONTH, -5).
Parameters:
field - the calendar field.
amount - the amount of date or time to be added to
   the field.*/
public abstract void add(int field, int amount)
```

Listing 3.6: Java 7 Calendar API - Static and Metalinguistic Signs

Analyzing the signs presented within the classifications of the section 3.4, we classified this API as rudimentary conversational within the proposed scenario for two reasons. First, the API is not able to support all the conversations required to run the scenario. The user needs to use another API to convert text to date and vice versa. The second reason is the inability of static and dynamic signs to establish a coherent conversation with the user. It is not clear what the rationalization of the proposed ADD operation is. In fact, use a mathematical metaphor for date operations is pragmatically wrong. It may still be effective semantically, but when conflict situations occur, the metaphor is not an appropriate choice for the conversation.

3.5.1.2 API DateTime - Java 8 - Metalinguistic Conversational API

The second API we use as an example falls into the category of metalinguistic conversational API. In the first task, to convert date into text, the API presents simple methods and can establish a fluid and effective conversation with the user. However, here we want to focus on the second scenario task, the addition of a period to the date object. For the user to be able to achieve productive dialogs with the API in this task, it is essential that he reads the documentation carefully. There are no static or dynamic signs that indicate the rationalization of the API over conflicting situations. Let us see below the static and metalinguistic elements present in the API that can be useful for the proposed scenario:

```
/** Returns a copy of this LocalDate with the specified
   number of months added.
This method adds the specified amount to the months field
    in three steps:
Add the input months to the month field
Check if the resulting date would be invalid
Adjust the day-of-month to the last valid day if
   necessary
For example, 2007-03-31 plus one month would result in
   the invalid date 2007-04-31. Instead of returning an
   invalid result, the last valid day of the month,
   2007-04-30, is selected instead.
This instance is immutable and unaffected by this method
   call. */
public LocalDate plusMonths(long monthsToAdd)
```

Listing 3.7: Java 8 DateTime API - Static and Metalinguistic Signs

Although it manages to expose all the internal logic through the metalinguistic signs, the API does not reach the maximum level of conversation in our classification. Previous work indicates that programmers have a hostile view of API documentation. Thus, the API designer should bring core issues, which may impact usage, to static and dynamic signs. Although the metalinguistic sign is well established in the API behavior, there is no guarantee that (1) the user will read the documentation and, even worse, (2) the user will agree with the design decision adopted. This takes us to the last level of conversation, which we will see next.

3.5.1.3 Fully Conversational APIs

As we cannot find a date API that meets the full conversation requirements we propose in this work, we will present a plausible API with modifications on top of the Java 8 DateTime API in the method that composes the scenario we are discussing. This hypothetical API will show how it is possible to go beyond the dialogs commonly offered by existing APIs. The following is an example of possible dialogs in this hypothetical API:

1: **U:** Convert this text "31-jan-2020" into a date object.
2: **API(D):** Object successfully created.
3: **U:** Now, I wish to add a month to the date object.
4: **API(D):** An error occurred while performing the operation. The result was an invalid date "Feb-31-2020". You need to explain what decision the

API must to make.

5: **U:** Understood. Perform the operation again, now with the behavior of adding the remaining days by pushing the date to the next month.

6: **API(D):** OK, operation successfully performed.

7: **U:** Now, convert this object into a text.

8: **API(D):** OK, successfully converted -> "Mar-03-2020"

To meet the above dialogs, in coding time, the API needs to present a conversational behavior. One way to do this is to add an additional message to the return object of the API with useful information for the user to understand the internal logic of operation. For example, if the called method is making some internal decision that could be controversial, the designer could add a field in the return object as: "*warningMsg:* Caution, adding periods operation may cause invalid dates. In this case, the API will behave in a rounding manner...". Such a message would already add a portion of the desired full conversation. Another alternative is to trigger an exception, forcing and offering the user the customization of the behavior.

Let us take as an example of the usage scenario involving the billing date. In practice, the calculation of a due payment bill may not obey the logic imposed by the API of rounding to the previous valid date. The user may want the API to round to the next valid date, for example. Thus, it may be consistent for the API to present static signs that indicate the possibility of customizing such behavior. A signature suggestion for the method follows below:

```
/** Returns a copy of this LocalDate with the specified
   number of months added.
This method adds the specified amount to the months field
    in three steps:
(1) Add the input months to the month field
(2) Check if the resulting date would be invalid
(3) adjusts if necessary as indicated in the parameter
   roundType.
*** Use value 0 to throw an exception in case of invalid
   date result
*** Use value 1 to indicate roll previous valid date
*** Use value 2 to indicate roll next valid date
May trigger an exception if you don't set roundtype
Be aware of the information messages on the return object
This instance is immutable and unaffected by this method
   call. */
public LocalDate plusMonths(long monthsToAdd, int
   roundType)
```

Listing 3.8: Fully Conversational APIs - Static and Metalinguistic Signs

A warning message, the triggering of an exception, and the possibility of customization can elevate the API to the fully conversational category. With these three new items in the API, the user can understand the internal logic without consulting the documentation, which is often a laborious and tiring task. Thus, we bring much of the designer's communication, which was only in the documentation, to the real user interaction with the API.

3.5.2 Refactoring API Classification

This section describes an evaluation of our conceptual framework performed with two versions of an refactoring API. The two versions we are going to present here are not yet for use in the public domain and are not yet in production. They are part of an ongoing Doctoral research on the subject of customization of program refactorings. Unlike the previous evaluation, we do not have a rudimentary conversational version of the API, since the designer was already concerned about offering an API with good quality of use. However, the fully conversational version is presented as a solution to improve the original version. As in the previous evaluation, we will show only part of the API contextualized within a usage scenario.

Scenario: An inexperienced Java programmer wants to improve the structural quality of his source code. He does not understand much about

code-smells and refactoring, only what he learned during his college course. He knows some types of refactoring available in a widely-popular refactoring catalog (18), but he cannot always understand the need for refactoring. So he would like to use an API to help him on refactoring his code. He will trust the tool but would like to fully understand what is happening.

3.5.2.1 Refactoring API - Metalinguistic Conversational API

A snippet code from the first version of the API classified as metalinguistic can be seen below. We highlight the method responsible for performing the refactoring operation. In this method, the user gives a list of files for the API to scan and search for refactoring opportunities. At the end of the operation, a list of refactored files is returned to the user.

In this version of the API, the designer had put in the documentation an explanation about the refactorings his API supported, and how the operations were performed. Thus, this conversation, which may be necessary for the full understanding of some users, was left out of the interface and appeared only in the metalinguistic signs. The source code below was created for this operation:

```
/** Search for problems and apply refactorings in the
   list of files. **/
public List<File> scanAndApplyRefactoring(List<File>
   files)
```

Listing 3.9: Refactoring API - Metalinguistic Conversational API - Static and Metalinguistic Signs

3.5.2.2 Refactoring API - Fully Conversational API

In order to give more conversation than the previous version, the designer brought the API interface an explanation about the problems encountered and refactoring needs. In this version, we have two important features of a conversational API. First, we have a conversation flow that determines the user interaction with the API. The designer decomposed the search and refactoring operation in two different operations by realizing the need for more conversation. First, the user searches for problems, and then he can refactor a certain code location containing each problem. Thus, a conversation flow is established between user and API.

Besides, the second feature that makes this case a fully conversational API is the presence of static and dynamic signs that explain the refactoring

decisions. The "applyRefactoring" operation returns, besides a list of refactored files, a list of explanations associated with each refactoring performed. Thus, an inexperienced API user, such as the one in our analyzed scenario, will understand the reason for each of the refactoring operations performed.

```
/** Searches the list of files for existing codesmells
   and returns a list with an explanation and the
   location of each problem. **/
public List<String> scanFiles(List<File> files)

/** Applies refactorings to the list of files according
   to the code-smells found. Returns the list of
   refactored files and a list of explanations justifying
    the need for each refactoring. **/
public Map<String, File> applyRefactoring(List<File>
   files)
```

Listing 3.10: Refactoring API - Fully Conversational API - Static and Metalinguistic Signs

3.5.3 Machine Learning API Classification

This section describes an evaluation of our conceptual framework performed with two versions of a Machine Learning API. The two APIs we are going to present here are not in the public domain. Similar to the last evaluation, we do not have a rudimentary conversational version of the API. However, the fully conversational version is presented with a solution to improve the original metalinguistics version. As in the previous evaluation, we will show only a piece of the API contextualized within a usage scenario.

Scenario: A geologist wants to create predictive models for image research based on geological data. However, they have no interest in learning a programming language for this. They are looking for an API that can abstract Deep Learning concepts, and that can provide a set of ready-made models. He would like to take an existing model and then run his training and prediction on top of his database.

3.5.3.1 Machine Learning API - Metalinguistic Conversational API

A snippet code from the first version of the API classified as metalinguistic can be seen below. We highlight the *"train_model"* and *"run_model"*

operations used by the geologist in his goals. In these two operations, it is necessary to inform which model is desired in the training and execution. However, such information is only present in the API documentation. Thus, either the user knows the type of model in advance, or necessarily the conversation is transferred to the API metalinguistic signs.

```
/** Train the model using the defined data and
   hyperparameters. To see the list of available models,
   check the API documentation. **/
void train_model(data, model, hparams)

/** Executes the model trained to perform predictions **/
void run_model(model, hparams)
```

Listing 3.11: Machine Learning API - Metalinguistic Conversational API - Static and Metalinguistic Signs

3.5.3.2 Machine Learning API - Fully Conversational API

In order to give more conversation than the previous version, the designer brought the API interface an list of available models. In this way, the user would not need to examine the documentation every time he wants to use another existing model. Besides, to further promote conversation, the operation of listing models can be triggered with a filter via the *"model_type"* parameter. Below, a snippet of API code representing this conversation flow.

```
/** Train the model using the defined data and
   hyperparameters. To see the list of available models,
   use the "available_models" available operation. **/
void train_model(data, model, hparams)

/** Executes the model trained to perform predictions **/
void run_model(model, hparams)

/** Lists the available models according to the type. To
   list all models, pass null value to the type. **/
List<Model> available_models(type)
```

Listing 3.12: Machine Learning API - Fully Conversational API - Static and Metalinguistic Signs

3.6 Discussion

Using concepts of communicability and Semiotic Engineering in API interaction has been discussed in the community for some years (1). We believe that this approach has a lot to contribute to the traditional software engineering process. This view on the API design goes beyond non-functional requirements commonly related to designers' concerns, such as robustness, modularity, and security. In this sense, the proposal of using Semiotic Engineering as a background theory is to put communicability as the center of the development process, always thinking about offering the best interaction, using the metaphor of conversation, to the end-user.

Through the examples listed in the previous sections, we realize how urgent the need to have an API design method that takes into account the aspects we are discussing in this work. The absence of conversation in an API leads users to its usage misinterpretations that can cause damage to the software being produced. The quality of use of an API unquestionably improves the resulting software quality. However, what we discuss here goes beyond that. We argue that conversation is a way to help API users recover from an API usage error, even if the user does not notice the existence of such an error.

By classifying the APIs used as examples, we realize some limitations in our conceptual framework. For example, could there be other types of signs that the API designer can use to pass on their communication? Or, would it be possible to divide the conversation levels into more than 3? And would this bring any advantage? As far as we investigate, these questions are open and can be addressed in future studies with different types of APIs.

The proposed conceptual framework can also be extended to other API domains and contexts. An API will always have the same sign structure that we list in this work. Our classification in different levels of conversation, relating the classes of signs proposed by Semiotic Engineering, has the robustness to help the API designer to think which sign is more appropriate to pass the desired message to the user. By establishing that a fully conversational API can pass all communication and internal logic using only the static and dynamic signs, we help the designer realize the need to give autonomy to the user to interact with his API without consulting the documentation at all times.

Finally, we point out that a study found in the literature indicates that API users tend to not consult the official documentation frequently (3). Consequently, it is relevant to call attention to the Software Engineering community that we need to better think about our API designs and how to interact with such APIs, taking into account the practical reality of software

development.

3.7 Related Work

In recent years, several researchers have investigated API usability. Some papers focus on strategies to improve the design process of an API, such as Watson (54) and Mindermann (27). These studies propose approaches to design with a focus on ease of use, using consolidated usability techniques in the HCI area. Eduardo Mosqueira-Reya et al. (28) have compiled a set of guidelines and heuristics to introduce usability in APIs during their project. Yessenov et al. (56) propose DemoMatch, a tool to support programmers in discovering how to use an API based on interactions with software that already uses it. Nguyen et al. (32) present a tool to scan the source code of an Android application in order to find possible security flaws caused by inappropriate APIs usage. Ichinco et al. (22) propose Example Guru, a tool for recommending APIs based on the context of the programmer's code.

The work that comes closest to our study is that of Santos and Myers (46). The study is based on the creation of annotations so that the API designer has one more tool to explain his design decisions. These annotations are attributes of a Java class whose goal is to make possible the declaration of metadata on objects. In this work, they made a small proof of concept to test this form of interaction between designers and users through annotations.

As we saw in this section, there are few studies that bring the API designer to the center of the phenomenon. Also, communicability as a desired quality in a programming interface does not appear in any of the studies found. Compared with the related work, our perspective is remarkably different from the approaches found in the literature. Our focus is on conversation and the two groups of humans involved with it, users and API designers.

3.8 Conclusion and Future Work

In this work, we present and discuss a conceptual framework for the definition and classification of conversational APIs. We present the conversation as a new perspective to capture the interaction between programmers and APIs. Based on the Semiotic Engineering theory, we describe the different classes of signs and how we can adapt them to the context of the communicability of software APIs.

Besides the proposed conceptual framework, we also present and discuss an evaluation of the framework on top of APIs from different domains. We

classify the APIs according to the conversational levels that were proposed, showing an example of how our framework can be used. These examples can guide a designer through the levels of conversation during the process of creating his API. Furthermore, we realized that possibly our conceptual framework needs to be improved in order to address some limitations. We need to investigate whether it is possible to cover more than three categories of conversation and whether there are different types of ways for the designer to send his communication that was not addressed in our framework.

Currently, our research goals are to create a method to assist designers in creating conversational APIs. Such design method can benefit from the conceptual framework we propose and describe in this article. In the long term, we hope to realize our method and be able to test how APIs conceived from the concepts defended here can have a better quality of interaction, resulting in fewer problems of misuse and failures resulting from misinterpretations.

3.9 Summary of Chapter 3

In this chapter, we present the conceptual basis of this doctoral thesis. We define what a conversational API is according to the Semiotic Engineering view (section 3.3). We also demonstrate how the classification of signs proposed by Semiotic Engineering can be applied to the context of software APIs (section 3.4). Furthermore, we demonstrate how our conceptual framework can be applied to assist designers in defining conversations in their APIs. An API designer who has the knowledge proposed by our framework will be better able to decide on the distribution of the conversation between the three different types of signs present in an API.

This chapter and its conceptual definitions serve as a theoretical and methodological guide for the following two chapters. Guided by the concepts proposed in this chapter, we present in the next chapter, a technical action-research study conducted in support of an API design. Chapter 5 presents the method based on the concepts defined in our conceptual framework and on the results that have emerged from the action-research described in chapter 4.

4
On the Support for Designing a Conversational Software API: An Action Research Study

Based on a perspective anchored in Semiotic Engineering theory (13), we consider an API as an artifact mediating the communication process between designers and users (chapter 2). During the development of an API, the designer needs to create dialogues with which the user will participate. In interaction time, the user and designer alternate conversation turns through these pre-created dialogues. As discussed in chapter 3, an API capable of offering effective dialogues to its users is called a conversational API. Thus, a *conversational API* is the one that communicates to its users its form of use and its internal logic of operation, making it clear the design rationale when abstracting concepts and tasks.

Unfortunately, there is limited empirical knowledge about the challenges designers face when creating a conversational API (section 1.1). Thus, in this chapter, we describe an technical action research study that we carried out in an R&D laboratory of a large IT company. Over several months of action research, we followed the design process of a machine learning API. We run three cycles of action research exploring and reveling the advantages and limitations of a set of techniques to support the design of a conversational API (section 4.3). The design of such conversational API was based on an existing API with poor conversationality. The main scientific contribution in this chapter is a set of challenges on the use of techniques to support API design (section 4.4). For each technique used, we also compiled a set of important lessons learnt that can improve the design process of an API.

The study reported in this chapter was submitted to a high reputation conference (*Qualis A3*). The paper is entitled *On the Support for Designing Conversational Software APIs: An Action Research Study* (5). As we discuss in section 4.8, the lessons learnt and the results of the action research were fundamental to the conception of the method to support conversational APIs design, which is presented as the main scientific contribution of this thesis, in chapter 5. The reader may want to skip section 4.2 since the theoretical basis presented is very similar to the section 3.2. The reader may also skip the beginning of section 4.1 since the motivation is the same in chapter 1.

On the Support for Designing a Conversational Software API

An Action Research Study

João Antonio D. M. Bastos
PUC-Rio
Rio de Janeiro - RJ, Brasil
jbastos@inf.puc-rio.br

Rafael Maiani de Mello
CEFET/RJ
Rio de Janeiro - RJ, Brasil
rafael.mello@cefet-rj.br

Alessandro Fabricio Garcia
PUC-Rio
Rio de Janeiro - RJ, Brasil
afgarcia@inf.puc-rio.br

Renato F. G. Cerqueira
IBM Research Brazil
Rio de Janeiro - RJ, Brasil
rcerq@br.ibm.com

4.1 Introduction

Application Programming Interfaces (APIs) are everyday tools in software companies. APIs are software components designed for providing specialized resources for supporting the development of software systems. These components can be developed in several programming languages, offering functions to support the execution of activities in particular domains. However, designing APIs for users' needs is far from trivial (29). In this way, one can see that the difficulties on proper understanding and using APIs interfaces is a recurrent claim (34, 57). These difficulties address the need of providing APIs communicability and usability since the early development stages (21).

Existing work to support API design often focuses on supporting designers in providing APIs' usability (31, 49, 50). However, these approaches lack on addressing communicability aspects (13), which is essential to assure that API users will properly understand the API interfaces. Usability focuses almost exclusively on users, with aspects such as their ability to learn how to use an artifact and what level of ease and efficiency of use (36). On the other hand, communicability is defined as an interactive digital artifact's ability to communicate to users, in an effective and efficient way, its designers' intent (which facilitates users' understanding and decisions about how to use the API) (13). Communicability addresses different, sometimes complementary, aspects addressed by usability approaches. Communicability goes beyond usability when pragmatic conflicts of interest between designer and user occur. When designer and user do not have the same understanding of a particular operation, an API with low communicability will not be able to pass its design rationale to the user. As consequence, it is very common reports of bugs coming from poor communicability in many popular APIs, such as Java Reflection (42) and automated refactoring APIs (38). For instance, let us consider the following interface offered by the Calendar API of the Java programming language:

public abstract void add (int field, int amount)

At *stackoverflow.com*, one of the most popular discussion forums for programmers, it is not hard to find postings with wrong explanations about the operation and behavior of the API above. In general, these posts are associated with bugs caused by misuse of the API. An example is a post which can be found at this link[1]. In this example, a user reports an issue addressing the result of the operation of adding a month to January 31 by using the *"add"* operation. Among the several answers provided by the other users, we observed that several users also misinterpreted the logic of the API. For instance, one of the answers to this issue argue that adding one-month is the same that adding add 30 days in the API, which would explain the unexpected result.

The case reported above does not address difficulties for using the interface, but it addresses the poor communication between the API designer and the APIs user on how to properly using the interface to support the users' needs. The users know how to call the API operation, set parameters, and which data should be passed. However, it is noted that users do not understand the API's internal operating logic, based on inappropriate assumptions about its use. Once they are convinced that they know how to use the interface, they will not resort to its original documentation. For them, they will consider the interface has some *bug* as it did not work as they expected.

Thus, we say that the conversation between designers and users was not well established in this API. Problems like this could be solved by implementing conversational APIs. A conversational API is an API capable of indicating the appropriate use to its users, solving pragmatic conflicts during the API use. The lack of the conversational characteristics hampers API users on understanding how to properly use the API's interfaces (3). The concept of conversational API is anchored in the Semiotic Engineering (13), a theory of human-computer interaction. Semiotic Engineering establishes that a computational artifact can be seen as a mediator of communication between two interlocutors: the one who created the artifact, that is, its designer, and the one using the artifact, that is, its user (13).

Although the potential benefits of establishing a proper conversation between API designers and users (15), we could not find in the technical literature approaches for supporting API designers on identifying and parameterizing the necessary conversations between APIs and users. In particular, methods focused on usability (28, 27, 31, 54) do not cover communicability issues. These methods focus exclusively on user interests, leaving out an in-depth analysis of the design logic used by the one who created the API.

In this paper, we report the challenges and lessons learnt from a tech-

[1]http://stackoverflow.com/questions/14618608

nical action research (52) conducted for redesigning an API in order to make it conversational. This API was designed to be widely used in a large scale industrial settings. The redesign of the API was required due to the considerable challenges perceived on its use. The action research was conducted in the context of a project in a research and development lab at IBM. During six months, one researcher worked with the development team for redesigning a machine learning API for seismic imaging. During the action research, we proposed and assessed interventions for redesigning the API based on theories of human-computer interaction, especially Semiotic Engineering. The conduction of the action research allowed us to be continuously using the lessons learnt for emerging innovative solutions while solving a real problem from industry.

In the following sections we summarize our theoretical background (section 4.2); describe the action research performed and its main challenges (section 4.3); report the results of the action research and the lessons learnt (section 4.4) discuss the contributions of the new API from the perspective of the developers involved in the action research (section 4.5); present and discuss some related work (section 4.6); discuss the limitations and threats to validity (section 4.7) and present our conclusions and perspectives of future work (section 4.8).

4.2 Theoretical Background

Our research aims at supporting API designers in designing conversational APIs. From a practical perspective, we expect that this support would lead to the development of APIs easier to understand and manipulate, independent from the programming experience of their users. For reaching this objective, we should go beyond traditional usability concerns, typically insufficient to establish an effective communication between users and designers. In this way, we opted by grounding our research in the Theory of Semiotic Engineering. This theory understands the process of Human Computer Interaction (HCI) as a particular case of metacommunication between humans mediated by computers (13). In this view, metacommunication is the communication taken between users and designers through a software artifact (see Fig. 4.1). Thus, the metacommunication process has three interlocutors: the designers who encode their intentions into software; the users, who express their own intent and interpretations by interacting with the software, and the technology itself, which represents the designer at interaction time.

For the Theory of Semiotic Engineering, software is a tool used to exchange messages between two groups of individuals: users and designers. One

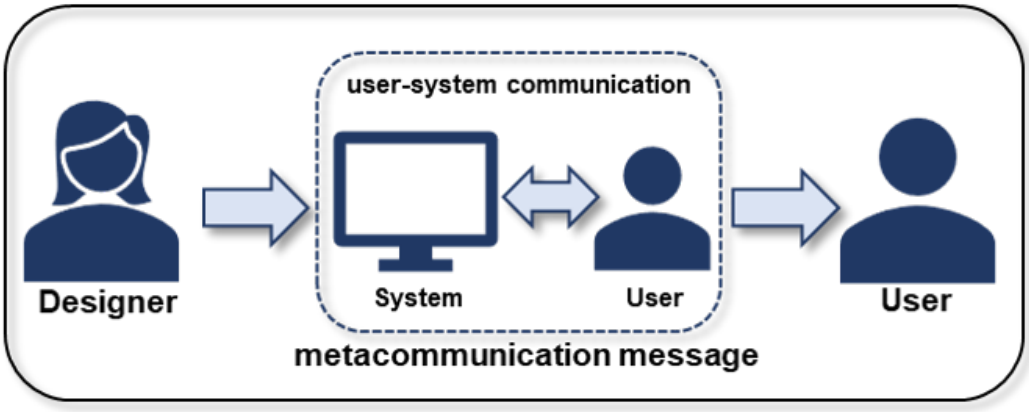

Figure 4.1: Semiotic Engineering and Metacommunication

of the main principles of Semiotic Engineering is the communicability. Unlike usability, which definition is strongly centered in the users, communicability is defined as the ability of an interactive artifact to effectively communicate its designers' intent to their users. Therefore, it is important to have in mind both designers' and users' aims when setting and evaluating the quality of the communication provided by the API's interfaces.

In regular software interaction, interfaces with inadequate communicability may lead the user to mistakes during the interaction. Semiotic Engineering calls those mistakes as communicability failures. A conversational software interface is one that can mitigate the occurrence of communicability failures, through error recovery dialogues designed by developers. For instance, if the user of text editor accidentally clicks in the "close button", the designer had already prepared a conversation with the user to find out whether he actually intends to close the application. Therefore, although the communicability failure happened, a conversational interface was able to bring the user back to a fruitful interaction. In this sense, we understand that a similar concern should be applied to the design of conversational APIs.

Based on a Semiotic Engineering perspective, we may define an API as a particular type of software designed for mediating the communication between two groups of programmers: the API designers and the API users. API designers communicate their intentions using expressions defined through method signatures, protocols, return values, and textual description (documentation). API users express their interpretation of the designers' intent while using APIs in their software systems. If the communication is not adequate, issues related to the incorrect use of APIs may arise, resulting in bugs (34).

Thus, we can apply the principles of Semiotic Engineering to identify opportunities for improving the metacommunication of APIs. In our research, this application is represented by the concept of conversational APIs. A

conversational API is an API designed to offer mechanisms for improving the effectiveness of the conversation between its designers and users. API designers should predict the different types of communicability failures that can occur, preparing the API to bring the user back to a fruitful interaction. For this purpose, the API interface should carry out its internal logic of operation to the API users. These users should be able to identify all the API's functionalities, understanding how to properly use them. Besides, the users should be able to adapt API internal behavior selecting through different pre-programmed design choices.

From a pragmatic perspective, conversational APIs should attend the four principles (maxims) of cooperation proposed by Grice (20): maximum of quality, maximum of quantity, maximum of relevance, and maximum of mode. The *maximum of quality* requires that the interlocutors provide only correct information. The *maximum of quantity* refers to the interlocutors present all the necessary information objectively, avoiding unnecessary information. The *maximum of relevance* requires that the interlocutors must stay focused on providing only relevant information for the conversation. Finally, *the maximum of mode* requires that the interlocutors should avoid the use of ambiguous expressions when transmitting the information.

4.3 The Action Research

In this section, we present the settings of the action research conducted at IBM. We present the goals of our study. Then, we describe the research context, characterizing the study participants, and the project involved. Finally, we report how we collected data during the cycles of the action research.

4.3.1 Research Objectives

We opted by conducting a technical action research due to our intention of developing a feasible approach for supporting API designers emerged from their practice. In this way, we planned to not only observing but also collaborating with a development team on redesigning a real API. From this, we expect to get an in-depth view of the challenges involved in the development of a conversational API, while proposing and validating innovative interventions to solve the problem of the practice during the development process.

Goals. the main goal of our research is to support designers on conceiving and implementing conversationality in new or already existing APIs. For this purpose, we want to identify effective strategies for improving

API's interfaces, once they are responsible for establishing the conversations between designers and users. Thus, our first goal with this action research is to *redesign the interface of a real API, making it conversational*. In this way, we intend to cover from the early development steps until the implementation of the API interfaces. Our second goal with this study is to use the rich experience obtained in the action research *to compile the lessons learnt for supporting the development of a future method for supporting the introduction of conversationality in APIs*.

The goals of our study do not include the implementation of the API's internal source code. We also do not intend to review the architectural aspects of the original project.

4.3.2 Research Context

The action research was carried out in the context of an industrial project for redesigning a Machine Learning API for supporting systems for seismic image investigation. We choose this project due to the valuable opportunity of performing our investigation in a challenging context, which includes the high complexity of its domain and the high levels of quality expected from the API by the stakeholders.

Team. The project was conducted by a development team from a research and development (R&D) department of IBM. This department develops software solutions for supporting research and practice on using natural resources. More specifically, this R&D department has been investing effort in developing software solutions supported by machine learning, such as web services and APIs, to assist decision making in areas such as agriculture and geology. For this purpose, the department's professionals have an outstanding theoretical and practical background in developing software solutions based in machine learning. Besides, most of them also have a extensive research background in Computer Science.

The development team of this project was composed of three API designers, including the technical leader. Among others, the technical leader was responsible for defining the API design and its interfaces. Besides being experienced programmers, all team members are also researchers, including a Ph.D. in Engineering and two Ph.D. candidates in computer science. They also have a considerable background on using the technologies used in the project, which includes the programming language (Pyton) and deep learning.

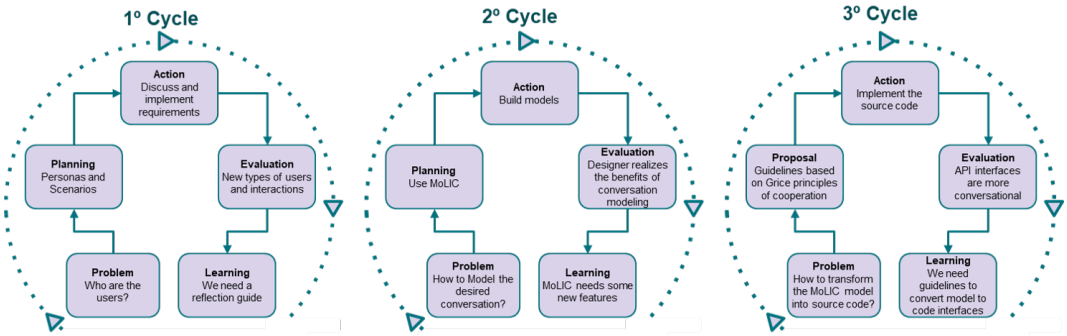

Figure 4.2: The Action Research Cycles

4.3.3 Execution

Action research usually requires long-term execution. In general, an action research study involves the execution of several interaction cycles along weeks or months. The central idea is to use the lessons learnt in the previous cycle to plan the subsequent action that will be performed in the following cycle, evaluating its results. Each cycle of an action research is commonly composed of five steps. They are [1] diagnosis [2] planning; [3] action; [4] evaluation of the intervention performed; and [5] learning (52).

Three complete action research cycles.The main author of this paper carried out the action research with the development team by six months. During this period, he performed three complete action research cycles. During each cycle, the author could perceive a receptive environment for research. At the end of the third cycle, the action research team concluded that the appropriate resources and settings to redesign the API were reached out. In other words, they concluded that the design of the API reached the desired level of conversation needed. Besides, we also concluded that there was sufficient knowledge to report practical lessons learnt on designing the conversational API. In the next subsection, we report the research procedures carried out throughout each cycle and how data collections were made.

4.3.4 The Action Research Cycles

Action research is characterized by its iterative nature, having complete cycles that start from the identification of a problem to the evaluation of a proposed solution (52). As already mentioned, our action research involved three complete cycles. Figure 4.2 presents an overview of the three cycles of our action research. Each cycle started with a problem to be solved addressing the API design, which can be seen in the lower-left box. After planning, executing,

and evaluating a solution, we close a cycle with a learning process. In this subsection, we describe in detail how the five steps were performed in each cycle from the perspective of the individual that played the role of the researcher, i.e., the main author of this paper.

4.3.4.1 First Cycle

At the beginning of the project, our goal was to design the conversational API. Since the early stages, we realized that the API design team was not sufficiently aware of the API users. At the same time, we also perceived that the application domain of the API addresses different user profiles. Thus, characterizing these profiles would not be a trivial task. This challenge led us to raise the following questions: *what are these users? What are their goals and needs?* To answer these questions, we carried out the first cycle of the action research (see figure 4.2, first box). The first cycle lasted around six weeks.

First, we explored the technical literature to better understand the problem and look for solutions. We search for existing tools and techniques to support the identification of the API users' profiles and their needs (planning step). We finished this step concluding that characterizing the API users through personas (11) combined with the description of interaction scenarios (7) would be a feasible alternative to define the users' needs. Both techniques have been shown useful in the field, especially in the interaction design of software interfaces. (43).

In the third step, we conducted daily meetings with the development team to characterize the personas and interaction scenarios. In these meetings, which lasted about one hour, all team members actively participated in the discussion for characterizing the API's personas and their corresponding interaction scenarios. AAfter concluding the characterization tasks, the team participated in a meeting for interactively evaluating the correctness, the comprehensiveness, and the usefulness of the artifacts generated. After the meeting, the development team concluded that the API should attend two personas in two scenarios composed of four sub-scenarios. Besides, the team also concluded that the characterization of personas and interaction scenarios was a very positive experience. They found that these techniques allowed them to identify new user's profiles and alternative ways for using the API not identified in the API's original design.

Finally, we conducted a meeting for compiling the lessons learnt in the first cycle of the action research. In this meeting, we reflected on which extent

the action performed helped us in reaching the goals of the action research, as well as on which extent this action could be improved. After this reflection, we found that the team had considerable difficulty in identifying the API personas and interaction scenarios. This finding led us to compile a checklist for guiding API designers on performing these activities (subsection 4.4.1).

4.3.4.2 Second Cycle

We started the second cycle by assuming that the whole set of API's user profiles and their corresponding needs of use were properly characterized. However, we diagnosed that the characterization of personas and scenarios are too informal and subjective for supporting the future implementation of the API. Thus, we concluded that we should model the conversations and the interaction paths corresponding to the scenarios established to each persona. Therefore, we started our second cycle motivated by the following question (figure 4.2): *How to model the desired conversations?*

The second cycle lasted around ten weeks. We searched by the technical literature aiming to find the most appropriate technique to assist us in modeling the API conversations. Once we did not find a proper technique to support our needs, we found some closer and adaptable alternatives, such as MoLIC (12), UML Interaction Diagram (48) and AlaDIM (33). While UML Interaction Diagram and AlaDIM are specific to software interface interaction design, MoLIC is more agnostic to the artifact type being designed. In addition, we can highlight that MoLIC has the characteristic of modeling interaction as a metaphor of the conversation established between designers and users (12). Besides, one can see that MoLIC has been frequently used in software projects to design human-computer interaction (44). Thus, we opted by using this MoLIC to redesign the API.

After selecting MoLIC, the team started to perform the action, i.e., to model the interaction scenarios. We originally planned to build one model for each interaction scenario obtained (first cycle). However, we realized during the action that certain scenarios are very similar. It allowed us to cluster some scenarios, reducing the number of models built. As a result, we created different models for representing completely distinct conversations.

Our evaluation of the action stated by discussing the main challenges faced by the team on using MoLIC. Due to its flexible nature to model conversations in general, MoLIC did not support the type of modeling they considered suitable for APIs. Thus, the developed team missed a more formal definition of the API dialogues. While conventional software dialogues with

the users may occur in several ways (images, links, text, menus, and others), user-API dialogues are necessarily performed by the exchanging of restricted messages through parameters and returns of operations. For this, MoLIC could not meet this need of designers of formal definition. Besides, MoLIC allows verbose and subjective dialogues, which is frequently undesirable in the context of software APIs. Thus, the development team learnt that some adaptations should be made to MoLIC better serving the propose of the project. These adaptations are described in subsection 4.4.2.

4.3.4.3 Third Cycle

After designing the whole interactions and modeling the API conversations, the development team faced the need to convert these models into source code by implementing the API interfaces. Thus, our challenge in this cycle was to implementing the API's operations signatures, return values, and function parameters. The third and last cycle of the action research lasted around ten weeks.

For overcoming this challenge, we provided the design team with a set of best practices and guidelines for building and structuring the API's source code. Those guidelines and practices focused on realizing the conversations designed in the previous modeling cycle. However, we could not find in the technical literature guidelines to support the API implementation based on MoLIC models. Thus, the development promoted discussions for depicting their own guidelines for converting each element of the MoLIC models to the API's interface elements. Our goal here was to emerge mapping rules in such a level that the interface elements could systematically be extracted from the content of the MoLIC models. These guidelines are described in subsection 4.4.3.

In the third step of this cycle, we implemented the API interfaces by following the proposed guidelines. The implementation was all carried out by the API development team, including the researcher. At the end of this step, all the API interfaces were successfully implemented. Then, the researcher conducted a structured interview with the team's technical leader. We use the interview to gather the feedback of the technical leader regarding not only the current cycle but also regarding the previous cycles of the action research. The findings of this interview is discussed in section 4.5.

4.4 Lessons Learnt

During the execution of the action research, We learnt a set of valuable lessons for designing conversational APIs. In this section, we present and discuss these lessons, distributed in three different subsections. The first subsection addresses the characterization of the API users. The second subsection addresses modeling the possible conversations between the API and its users. The third subsection addresses the definition of the API interfaces.

4.4.1 Who are the Users

4.4.1.1 Challenge: Making Designers Aware of the Users' Needs

API designers should be aware of its users for properly identifying the API's functionalities and interfaces. For introducing conversationality, this knowledge is also required. However, in this case, we need to reach a deeper level of knowledge regarding the whole set of the API users' profiles for properly identifying the set of conversations that could be established between the API and its users.

For example, let us consider the design of an API for deep learning. Its users range from experts in deep learning to novice programmers. For the experts, the API designer would feel comfortable on specifying interfaces using more specific terms addressing deep learning, such as "back-propagation" or "convex optimization." On the other hand, the API designer should abstract these concepts from professionals, such as medical doctors and geologists, which would use the API in their programs.

4.4.1.2 Solution: to Adopt Consolidated HCI Techniques

In our action research, the first step we identify as necessary to answer is how to properly identify which user profiles the API should support. As described in the previous section, we have used traditional HCI artifacts - personas and interaction scenarios - to try to map these users and their needs.

Personas: The concept of persona was first used by Alan Cooper (11) in the context of software design interaction. Personas are fictional characters used to characterize and combine the different roles played by software users. The characterization of each persona involves characterizing its domain knowledge, its motivation for adopting the developed technology, and its tech-

nological background. Besides, it also involves characterizing the beliefs, values, and behaviors surrounding this character.

Scenarios: After establishing the personas, the next task consists of composing the interaction scenarios. An interaction scenario (43) aims at specifying in detail the user's actions and the corresponding system responses. In our understanding, the system is the API under development. Interaction scenarios are composed of textual and concrete narrative, rich in contextual details. This narrative reports API's situations of use by characterizing the corresponding users, processes, and (potentially) real data. We use interaction scenarios for describing all the possibilities of interaction that a persona may have with the API.

4.4.1.3 Lesson Learnt: API Designers Have Difficulties on Establishing Personas and Scenarios

During the personas' identification and creation phases, the API designers reported difficulties on identifying and properly characterizing the API users. Truthfully, personas and interactions scenarios are not a method of requirements gathering, but a method of requirements registering. The lesson learnt at this stage was the need to give additional support to the designers on how to identify the API' personas. In this sense, we had to investigate what would be the important aspects to consider when describing the personas for an API.

API metacommunication template: The Semiotic Engineering (13), as a theory to support HCI, offers a technique that can help in the identification of users, the metacommunication template (13). Based on this template, Afonso (1) proposes the *API metacommunication template*. This template is composed of generic questions that may be used for characterizing users in the context of any software. Table 4.1 presents the template questions adapted for characterizing API users. It is important to note that some aspects addressed by the template may not be applicable in all contexts. For example, in the development of a general-purpose API – such as date APIs, or database access APIs – certain aspects such as "academic background" may be irrelevant. However, in developing an API to support scientific research, such information becomes eminent. It is part of the designer's work to identify which aspects are relevant in its API domain.

Personas were crucial to create empathy with users and to start developing their API thinking about another person's existence on the other side. By using the adapted metacommunication template, the API designers were able

Table 4.1: API Metacommunication Template

Question	**Aspects to be covered in the answers**
Who are the API users?	Culture / Language Professional or End-User Programmer Ursers' values Relevant demographic data Programming Experience Academic background Knowledge of programming languages and paradigms Types of API interaction patterns API domain knowledge
What do they need or want to do?	Intended use cases for the API Use case preconditions and restrictions User needs
What are their API preferences?	Programming conventions and culture Language specific conventions and culture Parameter styles and return types Appointment styles Productivity Accuracy in activity Use of auto-complete and other shortcuts Consult documentation or learn by doing
Why do they have these preferences?	Lack of experience or professionalism Personal values Naming preferences Programming culture Knowledge of other languages and APIs Academic training Programming environment requirements

to define two different personas (table 4.2 and table 4.3): a machine learning expert and a geologist having little background in programming and machine learning algorithms. We observed the template was useful for supporting the identification of the personas‘ needs.

The description of these personas were fundamental to support the creation of the conversations. When modelling the API interfaces, the designer needs to take into account the characteristics of each persona. This empathy created will generate new and adequate functionalities for the API, as we will see in the next subsection.

Table 4.2: Persona 1 - Machine Learning Expert

Who are the user?	They are professional programmers, with experience in the Python programming language, with academic backgound in the area of artificial intelligence, who know the terms and notations used in the area of DL. They are accustomed to python language patterns and object oriented. Intermediate knowledge of the API domain. They can "read" geological data, but have difficulty interpreting them.
What do they need or want to do?	They would like an API that offers an abstraction on geological data. They want to build the neural networks of their DL model. They don't have much knowledge about geological data, and would like the API to support them in reading and standardizing that data.
What are their API preferences?	Like most of professional programmers, they value productivity when creating their programs. They want an API that follows the Python conventions and has the object- orientation as a paradigm. An organized API with clear and objective documentation is what they expect from third-party software. They also want an API that abstracts the input data.
Why do they have these preferences?	Once they are professional programmers, they focus on the creation of the machine learning model. For them, the input data is not something to worry about. Therefore, they expect the API should give them maximum support on abstracting such input data, allowing them to focus on creating predictive models.

Table 4.3: Persona 2 - Geologist

Who are the user?	They are professionals in geology. Their main skill is to interpret geological data. They have no academic background in programming or computing. They are used to follow end-user programming paradigms.
What do they need or want to do?	They need to train predictive models based on geological data. However, they have no interest in learning a programming language for this. They are looking for an API that can abstract DL concepts and that can provide even a set of ready-made models.
What are their API preferences?	They prefer simplified APIs that are easy to use, having few configuration options. They also prefer APIs that follow the basic style of scripting with direct and simplified function calls. Although the are not used with programming, they understand the basics of the Python programming language.
Why do they have these preferences?	Since they are not professional programmers, they don't understand some aspects of programming and DL, such as how to build a neural network or how to convert data into tensor. Therefore, they need the API to be able to abstract these concepts, providing methods for data conversion as well as ready-made models.

4.4.2 How to Model the API Conversations

4.4.2.1 Challenge: to Help Designers to Model API Conversations

The API designer should model the different conversations that may be established between the API and its users. However, we observed the API designers were not used to build models during the API design process. On the other hand, it allowed us to propose and to assess the use of MoLIC (Modeling Language for Interaction as Conversation) (12), a traditional modeling language form the HCI field focused on the creation of interaction models.

4.4.2.2 Solution: Use MoLIC to Think About API Dialogues

MoLIC is a popular resource for HCI modeling, especially in the context of projects following the principles of Semiotic Engineering. MoLIC was created for modeling system interactions as conversations between the system and its users. In these conversations, the user and the designer (represented by his proxy) alternate turns of speeches and dialogues within scenes of interaction. MoLIC has two main elements: conversation scenes and conversation flows. *Conversation scenes* are represented through rectangular labeled boxes. The title (label) indicates what the scene is about. The box is composed of the set of speeches exchanged between designer and user. *Conversation flows* are represented by directed lines. These lines may be continuous for indicating a progressive flow in the conversation or dotted for indicating a regressive flow in the conversation. Regressive flows usually happen due to communication failure.

4.4.2.3 Lesson Learnt: MoLIC Should be Adapted

Although we observed that MoLIC is potentially useful to model conversations of an API, we also observed the opportunity to perform the following adaptations in its original composition, aiming at better supporting the modeling of the API designed during the action research. As future work, we will conduct new investigations to establish a comprehensive modeling language for supporting the design of conversational APIs in general.

Asynchronous flow: The first adaptation in MoLIC was the addition of the resource of asynchronous flow for modeling the conversations between APIs and their users. Asynchronous flow may be needed, especially when the API

requires long data processing to answer users, as was the case with the API we were modeling. In this case, the conversation flow is temporarily interrupted, returning to where it left off after receiving a proper signal from the API. It applies when the API call takes too long. For instance, in deep learning, the training of models and predictions may take a long time, leading to a temporary suspension of the conversation. The conversation is then only returned when the API emits a signal if the API user had programmed the source code to understand it.

Formalization of the dialogues: The flexibility of representation in MoLIC may be a problem for API modeling. Unlike traditional software interfaces, APIs are formal and rigid interfaces. The API interfaces need to be defined with parameters, data types and returns. Thus, other necessary adaptation in MoLIC was the formalization of the dialogues that take place within the modeled scenes. To this end, we used OpenAPI, a modeling language designed for specifying RESTFul services (23). OpenAPI offers a powerful tool for specifying the data flow of each method call. It is composed of two main features: a tool to support formal API specification tool and a tool for automatically generating source code.

In our adapted version of MoLIC, we used the OpenAPI resources to represent the alternation of dialogues between API users and API designers. The input attributes describe the user's speeches, and the return attributes describe the designer's speeches. Through using a set of reserved words and structure rules, the API designers were able to describe API interfaces, input data, output data, and documentation.

Figure 4.3 shows a model designed by using our adapted version of MoLIC. This model was build in the context of the API designed during the action research. This figure shows a conversation performed between the user and the API about the user's goal of build and training a particular model. When the user goes to the model training scene, he can go straight to talking about model training. Or, alternatively, the user can pass before for a scene to discover the available models. During the model training scene, the conversation between user and API is suspended by an asynchronous flow, returning only after the user gets a positive response from the API indicating the model's end-of-training. If the user wants to train another model type, the conversation with the API may continue by releasing a new training set or returning to the scene addressing the discovery of the available models.

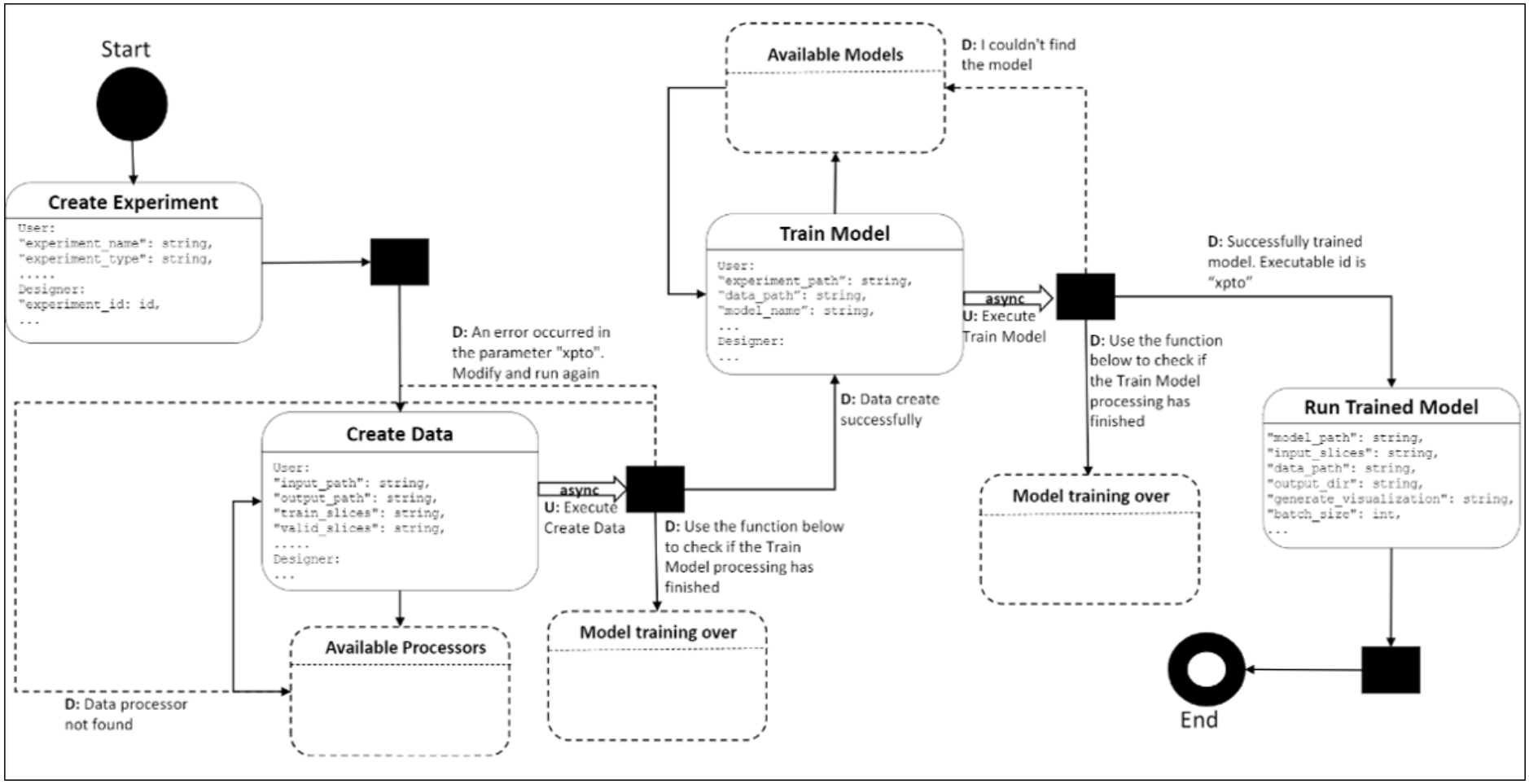

Figure 4.3: MoLIC and Adaptations - DL API Modelling

The exemplified conversation from the MoLIC diagram can be seen from two different points of view, depending on the persona involved in the conversation. The geologist, who does not have much programming knowledge and will not create own models, may opt by going through the scene of discovering the available models and identify which of them would fit the user need. On the other hand, for the DL expert persona, going through this scene is optional, once he may be training his own model. Thus and so he already knows which model he has chosen and what parameters and settings he has. This kind of conversation is modeled in the figure with the scenes of "Train Model", "Available Models" and "Run Model".

From this modeling presented, the API development team was able to think about new implementation needs in the API. The API has a dynamic set of ready-made models. However, the developers had not thought of a way to expose to users which models would be available. By creating a method for this, the conversation is improved, and there is a more fruitful interaction. Moreover, once the user already knows the model he wants to train, he does not need to go through the model selection dialog, which is optional. So, the API offers two conversational paths.

4.4.3 How to Implement the API Interfaces

4.4.3.1 Challenge: to Help Designers on Choosing the Appropriate Signs for API Interfaces

Unlike the traditional computer human interaction process, in which several types of signs (graphic, textual, sound, engines, etc.) may be used, the interaction with APIs and programming languages are limited to textual signs. These signs include function signatures, documentation, error messages, among others. MoLIC offers a set of guidelines to convert the model into user interfaces (graphic, textual, sound, engines, etc.). However, these guidelines are insufficient to model the communication between APIs and their users. Therefore, one challenge was to propose guidelines for supporting API designers on creating API interfaces based in MoLIC diagrams, which include identifying the appropriate signs.

4.4.3.2 Solution: A Set of Guidelines to Structuring the API Interfaces

Based on the results of our action research, we propose a set of guidelines to support the designer when naming and structuring API interfaces.

These guidelines are grounded in the Grice's principles of cooperation (20). Following, we list the guidelines with their corresponding principles.

API interfaces must focus on relevant information

The design of an API should be accurate in characterizing the expected conversations between API designers and API users. In the same way, an API design should also let clear which conversations are not allowed. That is, the API designer may choose to abstract some concepts as well as prioritizing others. This guideline addresses the Grice's principle of maximum relevance. For instance, considering the example of the API designed during the action research, one can see the conversation sometimes abstracted DL concepts. In other cases concepts from Geology were abstracted.

API documentation must be simple and concise

API documentation is a key opportunity for API designers to improve the conversation level with the API users. However, this relationship cannot become abusive. Once API interfaces frequently have a weak conversation, API designers are prone to embed most of the conversation in the API documentation. Consequently, it is not rare to find APIs composed of few methods but with extensive documentation. As previous work has already discussed (3), API users are not encouraged to search for the API documentation to solve their problems. Instead, they prefer to find examples in forums, which go straight to the point. Based on the principle of maximum quantity, we recommend that API designers should avoid establishing a tiring and laborious communication when documenting APIs. In other words, the API designer should be careful about introducing an overhead of information for the API users. In this sense, the API designer may compose more detailed and more synthesized versions of the API documentation for attending different types of users.

API interfaces must be cautious with metaphors and idiomatic expressions

The appropriate choice of names for the interfaces' identifiers is decisive for the quality of the API conversation. As stated by the principle of maximum mode, if the API designer adopts ambiguous expressions, they may hamper users' understanding of how to use the API. In particular, we recommend that metaphors and idiomatic expressions should be avoided. For instance, during the action research, we found that a metaphoric function name - "ZooModel" - was confusing for non-DL professionals. In this way, we suggested renaming the function to "AvailableModels."

API internal behavior must not contradict API interfaces or documentation

The API must be consistent between the different user interactions, assuring an efficient conversation flow. Based on the principle of the maximum quality, API designers should assure the consistency between the API documentation, interfaces, and internal behavior. Consequently, all the possible conversations should be correct and true. This correctness includes avoiding tricking the API user. In this way, it is considered a severe failure when the conversation results in a wrong understanding of the API internal behavior.

4.4.3.3 Lesson Learnt: We Need More than Just Guidelines

As a lesson learnt at this stage of our action research, we realized that we would need more than guidelines to support the designer in choosing the appropriate signs and establishing an effective conversation. In this way, we propose a conceptual framework to support the construction of conversational APIs. Due to its complexity and broad theoretical foundation in Semiotic Engineering, this framework is presented in another paper (4). In this conceptual framework, we borrow Semiotic Engineering concepts to establish different conversation levels according to the signs used by the API: static, dynamic, and metalinguistic. We discuss how the API designers may use each sign to reach better conversation levels with the API users.

4.5 Follow-up and Discussion

Researching in the context of real projects is frequently claimed as a challenge to the field. It could be even more challenging whether the research requires taking part in a long term project. Therefore, opportunities for conducting action research in industrial settings are not common in software engineering. Besides, recording and reporting data gathered in long term studies also prove to be quite complex activities. The work presented in this paper reports the results of an action research that required the collaboration of a researcher in an industrial project for six months. The action research was carried out aiming at investigating the API design process from the perspective of the API designers. As far as we are aware, this is the first study in the field with this purpose. Thus, despite the strict context involved, we believe that the lessons learnt and the proposed guidelines for designing conversational

APIs represents an important scientific contribution. This study also led us to apply the concept of conversational APIs for the first time. Another important contribution of the action research is the conversational API developed. The cycles of the action research covered the API development, from its conception until the implementation of its interfaces.

After concluding the action research, the development team finished the implementation of the API. In the sequence, we interviewed the technical leader of the API project, actively involved in all the development steps. The main goal of the interview was to characterize the perceived contributions of the interventions made during the action research to the API development, evolved to the guidelines presented in section 4.4. The feedback provided by the technical leader suggests that our approach brought practical benefits to the development process, positively influencing the designers' attitudes and decisions. The technical leader stated that "the study was essential," followed by his positive opinion about the modeling solution: "I would say that the part of analyzing how users receive messages is very important to know which message the API should send. It adds a new point of view for the API developer". Besides, he also claimed that "...as much as the API developer knows requirements, thinking about communication with the user is very important."

The technical leader also perceived our results as an important resource to support API designers on thinking about the different levels of conversation that should be offered by an API according to the user experience. For him, they were useful to realize whether an API should provide a verbose and explanatory behavior, or a practical and objective one. Regarding the perceived importance of providing conversation, the technical leader used another API as an example: "The Python {anonymous}[2]API, for example, is fantastic. However, the use of the API is awful. The designer did fantastic work, but it was based on an unusual way of use. The way to call the functions is similar to {anonymous}[2]. It maybe makes sense for someone. For the vast majority of people, it does not. The order in which you implement the functions or how you have access to them hinders or helps a lot its use."

Despite the positive feedback, it is important to also consider possible limitations of our study. One can see that the proposed solutions can be challenging for designers unaware of HCI theories. Thus, the presence of an HCI specialist may be required. Indeed, the researcher that conducted the study is an HCI specialist. He used his background to give to support the other team members. Regarding this issue, the technical leader interviewed

[2]The participant cited a third-party API. We think it is better not to expose which API

pointed out that "the benefits certainly outweigh the costs. In a company with several development teams, this cost would be even lower, diluting the presence of a single specialist in more than one project." In this sense, we see it as an open research question that we intend to address in future work in order to create a method suitable for API designers without HCI expertise.

The research on conversational APIs still has a long way to go. We believe the presented work is an important step towards a solid design method for supporting the software industry. In this sense, we intend to assess the resulting guidelines in the development of APIs from other domains.

4.6 Related Work

The reader may want to skip the first paragraph since the papers have already been presented in section 3.7.

Different studies have investigated the usability of APIs. Some investigations focus on approaches for improving the API design process, such as Watson (54), Mindermann (27), and (28). Watson and Mindermann introduced approaches focused on the API' easiness of use, grounded on consolidated usability concepts and techniques from HCI. Eduardo Mosqueira-Reya et al. generates a compilation of guidelines and heuristics that should be applied along the design process to achieve good usability in APIs. Although these studies are concerned with the API's quality of use, they lack addressing communicability aspects, including pragmatic issues and promoting the understanding between users and designers.

The lack of conversational APIs lead API users to have more difficult for applying them in their projects. Thus, one of the main goals of our research is to support the redesigning of already existing APIs to conversational ones. Alternatively, technical literature presents tools for assisting developers in properly using APIs. Yessenov et al. (56) propose *DemoMatch*, a tool to support programmers in discovering how to use an API based on interactions with software already using it. Ichinco et al. (22) propose *Example Guru*, a tool for recommending APIs based on the context of the programmer's code. Addressing code verification, Nguyen et al. (32), present a tool for scanning the source code of Android applications to find possible security flaws resulting from the inappropriate use of APIs.

More recently, some studies investigated the pragmatic issue of APIs misunderstanding (34). Although they did not propose methods or techniques for improving the API communication, these studies may represent resources for supporting understanding the limitations of the communication among

designers and users. Nielebock et al. investigated the misunderstandings on using APIs, leading to their misuse and even to the incidence of bugs. To mitigate this risk, the authors introduce a tool for identifying API misuses, offering rules for fixing its use. The work of Lamothe and Shang aims at understanding the appropriations made by API users. The authors found three workaround patterns followed by API users. These patterns can help API designers to understand the possible bypasses made by the API users (24).

4.7 Limitations and Threats to Validity

A limitation of our study is related to the lack of large-scale empirical evidence regarding our solution and guidelines for introducing conversation in APIs. However, it is important to note that we opted by first obtaining solid knowledge of the research object to then propose a mature technology promptly do be empirically evaluated in industrial settings. In this sense, we invested months of research effort for emerging a feasible technology from the perspective of experienced API designers. One can see this is a key benefit of conducting studies based on action research. This method has unique characteristics that facilitate cover the bridge gaps between academy and industry.

As threats to validity, we have the fact that the researcher has actively contributed to the API discussion. However, the development team members did not have the required skills for the interaction design of the API conversation. In theses cases, the researcher had to often interference along the creation of the API dialogues and conversation flows. One can note that the researcher was part of the development team given the nature of action research. Thus, despite this decision was a threat to validity, we believe that it was treated within our cycles of action research, where we had the constant validation of the designer team on the decisions made.

4.8 Conclusion and Future Work

In this article, we present the lessons learnt from a technical action research conducted to explore techniques to support the design of conversational APIs. At each cycle of our study, we used and adapted techniques aiming at prioritizing the user-API interaction in the API design for empowering its capacity of conversation. We believe that these techniques can be applied in other contexts, including for redesigning popular APIs from other domains

whose conversation is problematic, such as Java Reflection (42) and APIs for refactoring (38).

In order to mitigate the limitations and threats to the validity of this paper, we plan to carry out some future work with the design of different APIs with different characteristics and contexts. In the short term, we have already planned a case study for evaluating the API design method resulting from the lessons learnt in this study. This case study involves the design of an API for performing customized refactoring activities in the source code. In this study, we intend to evaluate in the practice whether our findings are adaptable to other contexts. Besides, we also expect to characterize the main difficulties the programming team will have when working with our approach. After the case study, we expect to acquire sufficient knowledge to report the first version of a method to support conversational API design.

4.9 Summary of Chapter 4

In this chapter, we present the technical action research performed during this Doctoral research (section 4.3). In this research, which lasted about six months, we had the opportunity to apply the concept of conversational API to develop a real API, within the context of deep learning in a professional software development use. To apply the concept, we followed the API design and searched for techniques and tools that could help the design team to create a conversational API.

From this study, we generated a set of lessons learnt (section 4.4). At each cycle of our study, we used and adapted techniques aiming at prioritizing the user-API interaction in the API design for empowering its capacity of conversation. With this result, we were able to propose a method to assist the design of conversational APIs. The method, detailed in the following chapter, defines a set of steps and techniques that must be followed by the designer to achieve the creation of a conversational API. 7

5
Colloquy: A Method for Conversational API Design

As discussed in chapter 1, we characterize an API as a mediating artifact of a conversation between two parts involved: the designer, and the user. Therefore, for the conversation to occur effectively and efficiently, the designer must be able to think about the necessary dialogues that his API needs to have with the user. So that way, in interaction time, user and designer alternate conversation turns through these pre-created dialogues.

Therefore, as we discussed in section 3.8, the API designer needs methodological support to create his APIs with these conversations. Different works proposed approaches for improving the design of APIs, but only from the perspective of usability. Watson (54) and Mindermann (27) introduced approaches focused on the API' easiness of use, grounded on consolidated usability concepts and techniques from human-computer interaction (HCI). Mosqueira et al. (28) compiled a set of guidelines and heuristics for enhancing usability on designing APIs. However, none of these methods help the designer to reflect on the dialogues he should create for API to have with his users.

In this chapter, we present a method to assist the designer in this task. The method, called Colloquy, is composed in 3 steps and aims to help the designer explore his epistemic power and think about his users needs, and, consequently, the conversations that the API should offer (section 5.4). This chapter presents an extended version of a paper accepted at *Brazilian Symposium on Software Engineering (Qualis A3) in the innovative ideas and emerging results track*, entitled *"Colloquy - A Method for Conversational API Design"*. Besides presenting the method, in this chapter, we present a case study where we put the method in a real scenario of API development. We present the results of this study and discuss potential improvements needed in our method. The reader may want to skip section 5.2 since the theoretical basis presented is very similar to the previous section 3.2. The reader may also skip the section 5.3 since the related work is very similar to the previous section 4.6.

Colloquy: A Method for Conversational API Design

João Antonio D. M. Bastos
PUC-Rio
Rio de Janeiro - RJ, Brasil
jbastos@inf.puc-rio.br

Rafael Maiani de Mello
CEFET/RJ
Rio de Janeiro - RJ, Brasil
rafael.mello@cefet-rj.br

Alessandro Fabricio Garcia
PUC-Rio
Rio de Janeiro - RJ, Brasil
afgarcia@inf.puc-rio.br

5.1 Introduction

The reader may want to skip the first seven paragraphs since this introduction has already been presented in chapter 1.

APIs are programming interfaces created to define the rules for using software components. These software artifacts are commonly implemented using modern programming languages. They are composed of rules establishing the set of operations available and the corresponding inputs and outputs required. Besides, an API should also guide the use of its operations. Recent work states that API interfaces play an important role in the communication between two types of programmers: the API user and the API designer (3). An API developed for supporting a particular domain reflects the designers' beliefs and understandings about this domain. On the other hand, the API user needs to interpret the designer messages as best as possible for achieving his goals.

When observing APIs as artifacts for mediating communication, we should have in mind an important aspect: the communication mediated by a computational artifact does not have the same dynamics of human communication. When creating the API interfaces, its designer should fix his communication in the source code and its documentation. Thus, if the designer assumes that the user already knows a specific concept, it may end up sending a message that has no meaning for the user. The user, in turn, will not have the opportunity to be promptly clarified.

The concept of conversational API is anchored in the theory of Semiotic Engineering (13), a theory of human-computer interaction. Semiotic Engineering establishes that a computational artifact can be seen as a mediator of communication between two interlocutors: the one who created the artifact, that is, its designer, and the one who is using the artifact, that is, its user (13). The communicability, central characteristic of quality for the Semiotic Engineering, concerns the designer's ability to expose the logic of the use of the artifact created.

Here, it is important to differentiate the concept of communicability from the concept of usability, widely explored in the modeling of software artifacts,

such as APIs (28). (17) (54). While usability focuses on ease of use and learning, communicability focuses on the ability to send the right information to the user about the best and most efficient ways to use the artifact created. For example, let us consider the Calendar API of the Java programming language:

public abstract void ***add****(int field,int amount)*

At stackoverflow.com, one of the most popular discussion forums for programmers, it is not hard to find postings with wrong explanations about the operation and behavior of the API above. In general, these posts are associated with bugs caused by API misuse. An example is a post that can be found at [1]. In it, a user questions the result of the operation of adding a month to January 31 using the "add" operation of the Calendar API. The user asking the question clearly did not understand the API's internal operating logic. When we checked several answers to the question, we noticed that several users also misinterpreted the API logic of operation. For example, one of the answers suggests that the "add one-month" operation is the same as "add 30 days", which would explain the unexpected result.

Clearly, in the example above, the problem is not the difficulty of using the interface, but poor communication. Users know how to execute the call, how to set parameters, and what data should be passed. However, it is noted that users do not understand the API's internal operating logic, based on inappropriate assumptions about the correct way to use it. Once they are convinced that they know how to use the interface, they will not resort the API original documentation. For them, if the interface did not work as they expected, it is probably due some *bug*. So, we say that the conversation between designer and user cannot be properly established in this API.

Problems such as the one described above could be solved by adding conversational features to the API. However, we did not find approaches for supporting the API designer to perform this activity. In particular, methods focused on usability, such as those composed of heuristics and guidelines, are not able to identify capture communicability problems during the design of the API. These methods are focused in the user interests, leaving out an in-depth analysis of the design logic used by the API designers. Thus, pragmatic conflicts related to the understanding the behavior of the API functionalities are not captured by approaches focused only on usability.

In this scenario, popular APIs for supporting software engineering activities, such as those available at IDEs for automating code refactorings, also

[1] http://stackoverflow.com/questions/14618608

lack establishing proper conversations with its users (38). Even simple refactorings, such as an Extract Method 5.1, may be performed in several ways. When developing APIs for supporting refactorings, the designer influences the API behavior based on his particular beliefs and understandings of the operations. Thus, the users may meet or not the implicit expectations of the API designers.

Figure 5.1 exemplifies the misuse of a refactoring API. The left side of the picture presents the original source code. The right side presents two alternatives for refactoring this code through extracting methods. Thus, when creating a refactoring API, the designer will choose the best alternatives from his point of view. However, the API user may not understand or even may not agree with the designer. To avoid situations such this, the designer should design a conversational API. Through pre-established dialogues, the user may interact with the API and understand its design rationale, or even modify its internal behavior, indicating his personal strategy for refactoring.

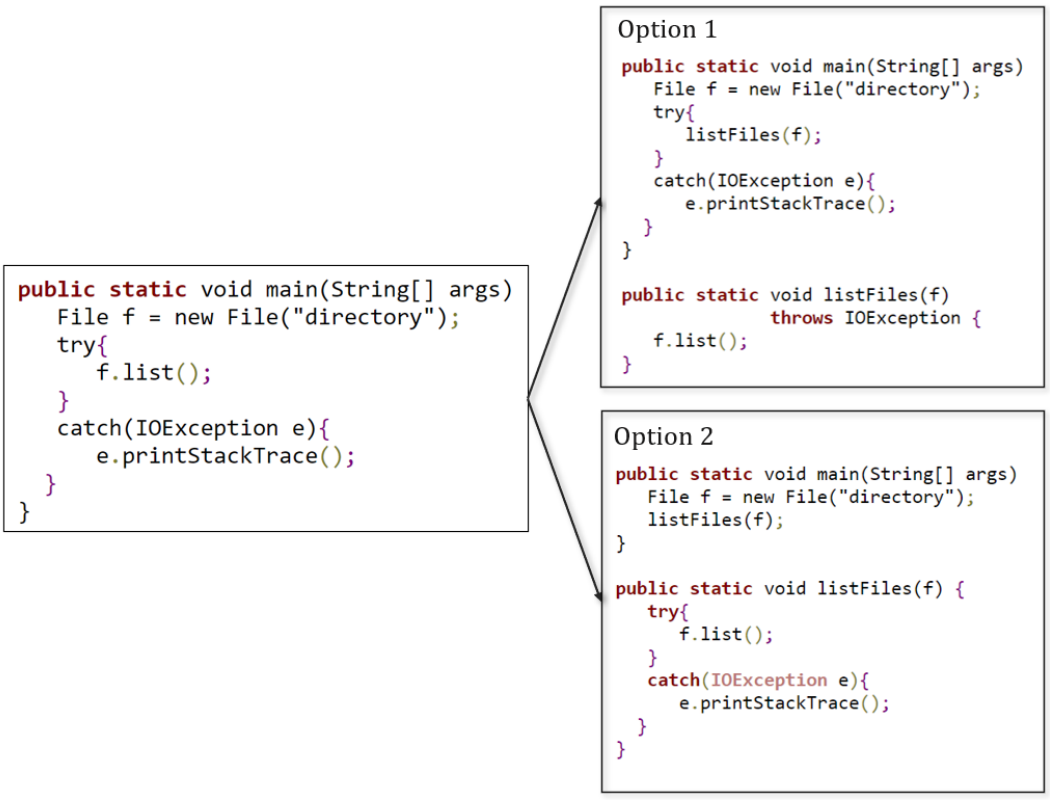

Figure 5.1: Two Alternatives to Extract Method Refactoring

As we saw in the examples above, sometimes it is not possible to make users and designers understand each other about the expected results of some operations. Moreover, when we look at the literature, we do not find studies that point to API design solutions that address such a problem. Thus, we

believe that a design process that has as central objective the development of the conversation in APIs can help in solving this problem.

In this paper, we present Colloquy, a method for supporting designers on introducing conversations in their APIs. This method is inspired by the lessons learnt on a recent technical action research (5). During six months, we worked with a research and development team from IBM Brazil for redesigning an API from a complex domain. Colloquy consists of three steps. The first step assists designers in identifying the API users and their conversation needs. The second step assists the designer in modeling the set of possible conversations with the different mapped users so that they achieve their goals. The third step provides a set of guidelines implementing the API interfaces.

The use of Colloquy was submitted to an in-depth case study, also presented in this paper. In this case study, Colloquy was applied for conceiving and designing a API for refactoring source code written in the Java programming language. As discussed in the above example about refactoring, conflicts between users and designers of such APIs are frequent since they do not always have the same understanding about how to perform an operation. Thus, an API that will deal with source code refactoring can significantly benefit from the concept of conversational API, and the method we propose for the design process. In this case study, we follow the design process of an API using our method. We then compare the API generated with the support of our method to a version previously created using guidance available in the literature. The findings of the case study indicate that Colloquy was essential for identifying several API properties.

Section 5.2 presents the theoretical background to our research. Section 5.3 discusses the main related work. Section 5.4 describes the proposed method. Section 5.5 describes the methodology of the case study conducted. Section 5.6 shows the study results and the differences in the API created with our method. Section 5.7 discusses our findings and threats to validity. Section 5.8 presents our conclusion and future work.

5.2 Theoretical Background

In this section, we introduce the theoretical background of the proposed method to support the introduction of conversation in the design of APIs.

5.2.1 Semiotic Engineering

Semiotic Engineering understands the process of human-computer interaction as a particular case of metacommunication between humans mediated by computers (13). From this perspective, metacommunication consists of communication established between users and designers through a software artifact (see Figure 5.2). We can identify three distinct interlocutors in the process of metacommunication: the designers, who encode their intentions in software; the users, who express their own intentions and interpretations through interaction with the software, and the technology itself (called proxy or designer's deputy), which represents the designer at interaction time.

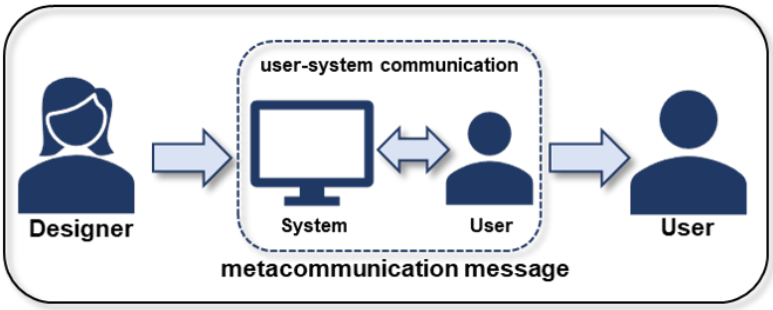

Figure 5.2: Metacommunication Process

For Semiotic Engineering, the software is a resource used to exchange messages between two groups of individuals: their users and their designers (13). The principles of Semiotic Engineering can be used to identify opportunities to improve the communicability of APIs (15). Unlike usability, whose definition is strongly user-centered (31), communicability is defined as the ability of an interactive artifact to effectively communicate the intent of its designers to their users (13). Thus, it is important to consider both designers' and users' objectives when defining and evaluating the quality of communication provided by API interfaces.

In the interaction performed on a software potentially provided with usability, but scarce in communicability, the interfaces can lead the user to make mistakes during the interaction, resulting in communication failures (14). On the other hand, a conversational software interface can mitigate the occurrence of communicability failures through dialogues that help users notice errors and correct them. In this research, we understand that this capability should be explored and applied to the context of API design, resulting in the development of conversational APIs (2).

5.2.2 Conversational API

Based on the Semiotic Engineering perspective, we can define an API as a particular type of software designed for intermediate communication between API designers and users. Designers communicate their intentions through expressions found in the signatures of operations, protocols, return values, and API documentation. In turn, users express their particular interpretation of designers' intentions throughout the use of APIs to achieve the goals of their software systems. If this communication is not appropriate, problems may arise related to incorrect use of the APIs, resulting, for example, in the incidence of bugs (3).

Thus, we can apply the principles of semiotic engineering to identify opportunities to improve the metacommunication of APIs (15). In this sense, the concept of conversational APIs was defined. A conversational API is an API designed to provide mechanisms to improve the effectiveness of conversation between designers and users (4), given the pragmatic contexts of use of its users.

Besides, a conversational API must meet the Grice's principles of cooperation in its four conversational maxims (20): the maxim of quality, the maxim of quantity, the maxim of relevance, and the maxim of mode. The *maximum of quality* says that the interlocutors must present only true information, avoiding false or doubtful statements. The *maximum quantity* says that interlocutors must present all necessary information, but without exaggerating with information irrelevant to the context of the conversation. The *maximum of relevance* says that interlocutors must maintain focus and relevance, that is, present only information focused on the context of the conversation. And finally, the *maximum of the mode* text says that interlocutors should avoid the use of ambiguous expressions when transmitting the information.

In the development of a Conversational API, the creation of scenarios is adopted as a design principle, which allows the designer to predict the most varied cases of use of an API and its pragmatic contexts of use. It is up to the designer to prepare the API to maintain a productive interaction with the different types of users. In turn, users should be able to identify all the functionalities of the API, understanding how to use them properly. Additionally, the API designer must predict the different types of communication failures that can occur, leaving pre-created dialogues for users to recover from these possible failures.

5.3 Related Work

The reader may want to skip the first, third and fourth paragraph since the papers have already been presented in section 4.6.

Different studies proposed approaches for improving the design of APIs, especially from the perspective of usability. However, they lack on addressing the communicability perspective, including pragmatic issues as promoting the understanding between users and designers. Watson (54) and Mindermann (27) introduced approaches focused on the API' easiness of use, grounded on consolidated usability concepts and techniques from human-computer interaction (HCI). Mosqueira et al. (28) compiled a set of guidelines and heuristics for enhancing usability on designing APIs.

Some other studies also offer API design methods. Each of these methods aims to improve different characteristics of an API. Watson's work presents a case study where a text analysis technique was applied to improve APIs' usability (53). The author presents a method to analyze the API interfaces to identify inconsistencies that could reduce usability. In this method, he decomposes the elements and analyzes whether the name is compatible with the type of data, analyzes if the name of the methods is compatible with what the method does, and finally analyzes the methods composed of get and set. The work of Stylos et al. formalizes a method for the process of API redesign (51). The method is composed of steps involving the presence of potential users. First, the participants go through interviews to get requirements and create personas. Then, the participants are called again to perform tests on top of pseudocodes. Thus, after these steps, a user-centric API is designed. Both studies, although they formalize a method, do not touch on the central issue of conversation, which is the lack of understanding between designers and users regarding controversial decision-making.

The lack of conversation in existing APIs lead API users to have more difficult for applying them in their projects. Thus, one of the main goals of our research is to support the redesigning of already existing APIs to conversational ones. Some studies investigated the pragmatic issue of APIs' misunderstanding addressing the motivation of our research by revealing several limitations of the communication among designers and users. However, different from us, they do not propose alternatives for redesigning the APIs. Nielebock et al. investigated the misunderstandings on using APIs, leading to their misuse and even to the incidence of bugs (34). To mitigate this risk, the authors introduce a tool for identifying API misuses, offering rules for fixing its use. Lamothe and Shang (24) aims at understanding the appropriations made by API users.

The authors found three workaround patterns followed by API users. These patterns can help API designers to understand the possible bypasses made by the API users.

Technical literature also presents studies introducing tools for assisting developers in properly selecting and using already existing APIs despite eventual communication issues. Yessenov et al. (56) propose *DemoMatch*, a tool to support programmers in discovering how to use an API based on interactions with software already using it. Ichinco et al. (22) propose *Example Guru*, a tool for recommending APIs based on the context of the programmer's code. Addressing code verification, Nguyen et al. (32), present a tool for scanning the source code of Android applications to find possible security flaws resulting from the inappropriate use of APIs.

5.4 Colloquy

In this work, we propose Colloquy, a method to establish conversation in APIs. Our method consists of a set of iterative steps. Each step consists of a set of guidelines that will support the API designer to reflect on the conversations that the API should make possible with its users.

Figure 5.3 illustrates the Colloquy steps, including its main techniques and the resulting artifacts. In the following subsections, we introduce in detail these steps. The first step of Colloquy requires the identification and characterization of the API users and their requirements. The second step consists of modeling the possible conversations between the API and its users, including error recovery conversations. The third step consists of defining the API interfaces. For this purpose, the third step consists of guidelines for naming functions, establishing parameters and their names, and defining appropriate return messages. Appendix A illustrates the step-by-step execution of Colloquy for the design of a date and time API.

Colloquy's main objective is to guide the designer through the process of developing his API, offering techniques and tools for exploring and creating user-API conversations. Thus, Colloquy does not predict the participation of real users in the development process. However, Colloquy does not prohibit such participation, and it is up to the designer the option to adopt or not users in the evaluations of the artifacts created.

Thus, we consider that a designer who has the availability to include real users in the API's development process can perform studies with these users in three specific Colloquy points. First, after creating the personas and interaction scenarios, the designer could validate the created documents by

performing workshops with real users of his APIs. With this kind of empirical study, the designer could identify the need or not create new personas or interaction scenarios.

Another point that can be useful to employ real users is after modeling the interaction with MoLIC4APIs. A designer could benefit from asking users to navigate through the created diagram, using simulation and making the user exchange dialogues with the API through the model. This type of study would be useful to verify if the modeling really meets the objectives and needs of real users. Such objectives would already be outlined and validated in the previous step with personas and scenarios. With this empirical study, the designer would only validate if the conversation flow is adequate.

Finally, the study with real users could come on top of the final API interface. The designer could design mock behaviors in his API and perform empirical studies to identify if his interfaces fulfill the previous steps' conversation. By doing this study before the final implementation, the designer could identify new requirements and eventual communication failures, correcting them before the final release of his API.

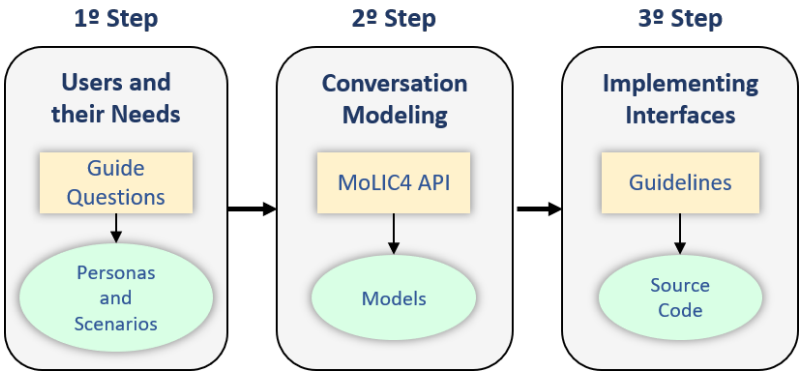

Figure 5.3: Colloquy Steps

5.4.1 Personas and Interaction Scenarios

The first step of Colloquy is aimed at discovering of the API's potential users and their corresponding requirements. This step is critical for identifying who are the API users, what they need, and what level of detail should the API conversation have for them. For this propose, Colloquy recommends adopting two technologies often adopted by human-computer interaction professionals: *personas and interaction scenarios.*

5.4.1.1 Personas

Alan Cooper (11) first used the concept of personas in the context of software interaction design. Personas are fictional characters used to characterize and combine the different roles played by the software user. The characterization of each persona involves establishing their domain knowledge, motivation to adopt the developed technology, and their academic and professional training. Furthermore, characterizing a persona also involves characterizing their beliefs, values, and behaviors. Thus, the characterization of characters will give the API's designer the knowledge to enable the most appropriate conversations for API users. For example, an experienced programmer may require a more detailed interface, while an occasional programmer may require a more simplified interface.

5.4.1.2 Interaction Scenarios

After establishing the personas, the next task is to compose the interaction scenarios of these personas with the API. An interaction scenario aims to specify in detail the actions of the user and the corresponding system responses (43), in this case, the API in development. The interaction scenarios are composed of a textual and concrete narrative, rich in contextual details. This narrative reports situation of use of the API, describing the processes involved, and (potentially) real data. Thus, we use interaction scenarios at this stage to describe all possibilities of interaction that a user may have with the API.

5.4.1.3 Guidelines for the Characterization of APIs Personas and Interaction Scenarios

To create a conversational API, it is especially critical that the designer can think of the full set of possible personas and interaction scenarios for its API. It is not uncommon for users of an API to make different appropriations than those they are designed. However, creating personas and interaction scenarios may not be a simple task. When we put conversation as a central element in the design process, we need to emphasize a set of special characteristics of the personas. In this way, our method offers a set of guiding questions (Table 5.1) to help the designer to make this reflection. These questions are inspired by the API metacommunication model proposed by Afonso (1).

Table 5.1: API Metacommunication Template

Question	Aspects to be covered in the answers
Who are the API users?	Culture / Language Professional or End-User Programmer Users' values Relevant demographic data Programming Experience Academic background Knowledge of programming languages and paradigms Types of API interaction patterns API domain knowledge
What do they need or want to do?	Intended use cases for the API Use case preconditions and restrictions User needs
What are their API preferences?	Programming conventions and culture Language specific conventions and culture Parameter styles and return types Appointment styles Productivity Accuracy in activity Use of auto-complete and other shortcuts Consult documentation or learn by doing
Why do they have these preferences?	Lack of experience or professionalism Personal values Naming preferences Programming culture Knowledge of other languages and APIs Academic training Programming environment requirements

To answer each question, the designer should have in mind the set of aspects presented in the right column. For example, when answering *"Who are the API users?"*, the designer must consider the user experience and culture, among others. For example, in the design of a date and time API, culture can impact the API's conversation about the time zone and the date input and display format.

5.4.2 Conversation Modeling

Once the personas and the corresponding interaction scenarios are defined, the API designer needs to model the different conversations that the user can establish with the API. For this purpose, we designed MoLIC4API (MoLIC for APIs), a language resulting from the adaptation and combination of two modeling languages often used in HCI and Software Engineering: MoLIC (12) and OpenAPI (23).

MoLIC (Modeling Language for Interaction as Conversation) (12) is a popular resource for human-computer interaction modeling, especially in

the context of projects having communicability as a key factor. MoLIC was created to allow the designers of systems to model the interactions as conversations between user and system. In these conversations, dialogues between users and designers are represented through scenes of interaction. The other language we use as a basis is OpenAPI (23), which is a modeling language originally designed for RESTFul service specification. OpenAPI has a formal API specification language and an automatic source code generation feature from this language. Through a set of reserved words and structure rules, the designer can describe API interfaces, input data, return data, and documentation.

5.4.2.1 MoLIC4API

MoLIC4API is composed of two main elements of MoLIC: conversation scenes and conversation flows. A conversation scene is represented by a rectangular box, with a title describing the scene and a set of designer and user speeches. The conversation flows are represented by directed lines, which can be continuous, indicating a progressive flow in the conversation or dotted, indicating a regressive flow in the conversation, usually due to some communicability failure. Besides, MoLIC4API also includes a new element: the representation of asynchronous flows, which serves to signal possible operations that may take time and need some return function to have the result achieved. Also, MoLIC4API uses the formalities already existing in OpenAPI for a detailed description of the user and API conversation. Through this description, the designer will explain what the user's speeches will look like and what the API (your proxy) will look like, in terms of parameters, data types and structures, and object names.

Thus, MoLIC4API combines the flexibility of MoLIC to represent conversation flows and the formality of OpenAPI to represent data exchange during conversations. The application of MoLIC4API is independent of programming language. The designer should use MoLIC4API to model all interaction scenarios that were thought of in the previous phase. When modeling these scenarios, the designer must reflect on the conversation flow and call the API operations' sequencing to achieve the users' goals. It is essential that the designer can also model the backward flow at this stage, predicting possible communication failures and offering fruitful dialogues for the user to recover from.

Figure 5.4 presents, through an illustrative example, the elements of the proposed language. In the figure, we model a possible conversation between a user and a Deep Learning API. In this model, we illustrate a conversation where

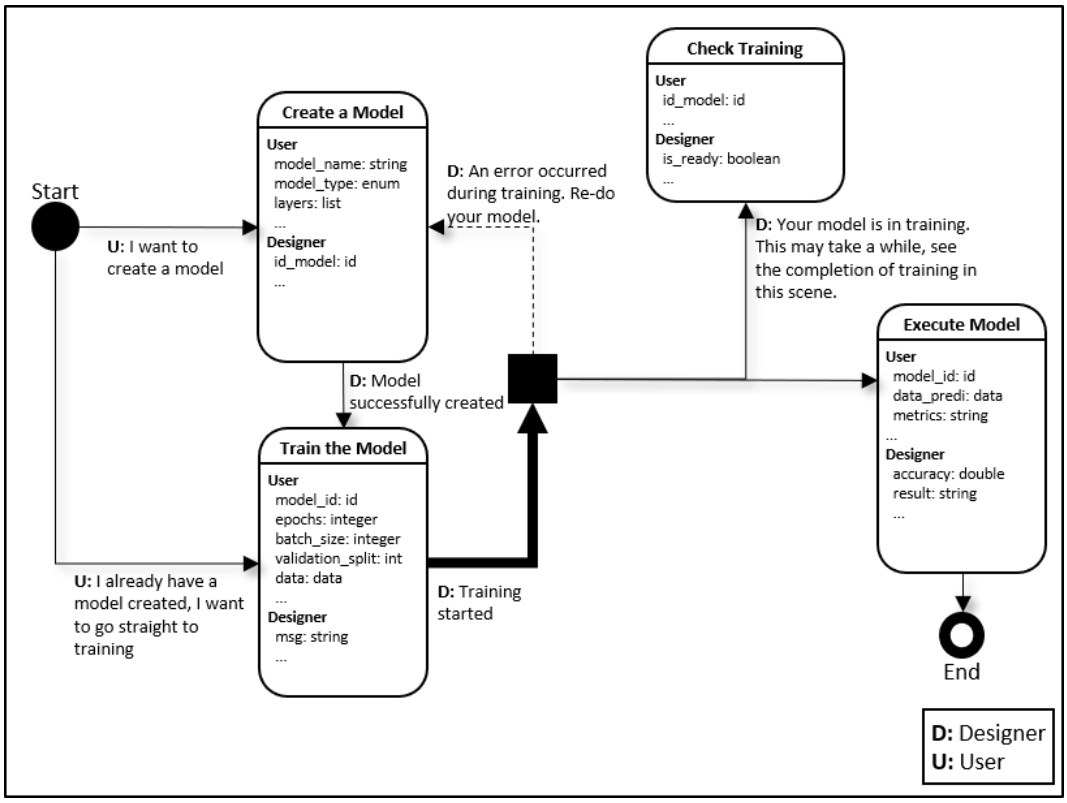

Figure 5.4: Example of MoLIC4API modeling

the user wants to create, train, and execute a machine learning model. The API must support these dialogues and guide the user through the call flow needed to achieve the proposed goal. The main elements of MoLIC4API can be seen in the picture. The conversation scenes are represented by the rectangular box, with the title above and the user and designer speeches below, represented through OpenAPI syntax. The arrows indicate the possible flows of the conversation followed by messages from the designer or the user indicating the conclusion of the scene it is leaving. The flows can be progressive (continuous arrows) or regressive (dotted arrows).

5.4.3 Interfaces Implementation

The primary goal of Colloquy is to help designers on modeling the conversations that can be established between the API and the user. In this sense, our method also aims to promote traceability between the mapped conversations and the API interfaces. To this end, we have developed recommendations for designers to convert MoLIC4API diagrams into source code. We describe this recommendations in subsection 5.4.3.1. Besides, the method also offers a set of guidelines to support the designer when naming and structuring API interfaces. These guidelines are grounded in the Grice's principles of coopera-

tion (20). In subsection 5.4.3.2, we list the guidelines with their corresponding principles.

5.4.3.1 Recommendations for Composing Source Code from MoLIC4API Diagrams

R01. Represent each scene through an API operation: Each scene in a MoLIC4API diagram describes an exchange of messages between the designer and the user. Although the designer has different mechanisms to send his message to the user, the API user can only send his message in one way: by executing API operations. Therefore, we recommend that an API operation should mediate each scene.

R02. Represent each attribute of the user's speech in a scene as an operation parameter: Within the conversation scenes, the user's speech, represented by the OpenAPI syntax, should become parameters in the call of the operation that originated from the scene.

R03. Represent each attribute of the designer's speech as a return of the operation: The same goes for the designer's lines, which must be converted into variables of the return object of the operation that originated from the scene.

R04. Include messages to the user in the call back operations: The main message following the conversation flow should appear in the return of each operation. If the flow is regressive, the operation return must contain be represented as an error message. In this way, the API designer should be careful on writing these messages. An error message must be comprehensive enough for the user recovering from the error and shift to a productive interaction. If the flow is progressive, the return of the operation may report a message indicating the success of the operation. Besides, messages indicating the next step the user should take in the interaction may be beneficial in this case.

It is important to note that the recommendations presented here for implementing interfaces should not be followed as a mandatory rule. This mapping can be adapted to suit other API requirements that consider features such as efficiency or safety. We intend to refine these recommendations in future studies. Another future improvement is the development of automated tools for this step.

5.4.3.2 Naming and Structuring Guidelines

G01. API interfaces must focus on relevant information: The design of an API should be accurate in characterizing the expected conversations between API designers and API users. In the same way, an API design should also let clear which conversations are not allowed. That is, the API designer may choose to abstract some concepts as well as prioritizing others. This guideline addresses the Grice's principle of maximum relevance.

G02. The API documentation must be simple and concise: The API documentation is a key opportunity for API designers improving the conversation with the API users. However, this relationship cannot become abusive. Once API interfaces frequently have a weak conversation, API designers are prone to embed most of the conversation in the API documentation. Consequently, it is not rare to find APIs composed of few methods but with extensive documentation. API users are not encouraged to search for the API documentation to solve their problems. Instead, they prefer to find examples in forums, which go straight to the point (3, 24). Based on the principle of maximum quantity, we recommend that API designers should avoid establishing a tiring and laborious communication when documenting APIs. In other words, the API designer should be careful about introducing an overhead of information for the API users. Alternatively, the API designer may compose more detailed and more synthesized versions of the API documentation for attending different types of users

G03. Avoid metaphors and idiomatic expressions when naming interfaces: The appropriate choice of names for the interfaces' identifiers is decisive for the quality of the API conversation. As stated by the principle of maximum mode, if the API designer adopts ambiguous expressions, they may hamper users' understanding of how to use the API. In particular, we recommend that metaphors and idiomatic expressions should be avoided.

G04. The API internal behavior must not contradict the API interfaces and the API documentation: The API must be consistent between the different user interactions, assuring efficient conversation flows. Based on the principle of the maximum quality, API designers should assure the consistency between the API documentation, interfaces, and its internal behavior. Consequently, all the possible flows should result in correct conversations. This correctness includes avoiding tricking the API user, resulting in a wrong understanding of the API internal behavior.

5.5 Study Design

5.5.1 Goal and Research Questions

The case study presented in this chapter aims to test the advantages that Colloquy brings when used in developing a real API. To do this, we establish a context in which we could get a API designer interested in participating in our study. We defined that we would use Colloquy to design an API aimed at refactoring source code in the Java programming language. This API was designed by a Doctoral researcher focused on refactoring and code smells. By relying on the guidelines provided by Wohlin et al. (55), we proposed the following study goal.

- **Analyze** an API design process supported by Colloquy,
- **For the purpose of** characterizing the Colloquy advantages,
- **With respect to** the capability to generate a conversational API,
- **From the viewpoint of** API designers,
- **In the context of** real API design for program refactoring.

The study goal led us to design our research question a follow.

> **RQ1. By using Colloquy, an API designer will be able to design APIs with proper conversations?**

The first research question aims to understand whether Colloquy will really deliver a conversational API as we planned. Thus, to address RQ1, we established that the API generated needed to contain an effective and efficient conversation flow that the user could navigate to complete their API usage goals. Thus, the API designer should be able to model and define its interfaces. We discuss about this in the results (section 5.6).

> **RQ2. What other advantages could Colloquy bring to the designer?**

RQ2 aims to understand whether, in addition to the conversational API, Colloquy could bring more advantages to the design process from the designer's point of view. Therefore, we conducted a set of interviews with our participant to collect his opinion about each Colloquy phase. In these interviews, we addressed questions such as ease or difficulty of executing the Colloquy, effects on API design, and need for additional support.

5.5.2 API Context

As we discussed in the introduction, conflicts between users and designers of refactoring APIs are frequent since they do not always have the same understanding about how to perform the operation. Thus, an API that will deal with source code refactoring can significantly benefit from the concept of conversational API, and the method we propose for the design process. This study was conducted in the context of a project for building an API for refactoring source code written in the Java programming language. The API was designed by experienced Java programmer who has developed software in both academic and industrial environments. The API designer, whom we will refer to as a participant from now on, has extensive experience developing and designing reusable APIs and software. Besides, the participant has extensive experience in developing refactoring and code smells analysis tools.

Reported work investigating APIs for refactoring in software IDEs faces a recurrent problem of misuse. There are well-defined and widely studied catalogs on how to perform various refactoring types in source code (18). Still, we find cases where users disagree about the operations performed by APIs (38). Even simple refactoring as an Extract Method (18) can find several ways to be performed among API users (38). Thus, we believe that a conversational API may be the appropriate approach to decrease misuse recurrence for the refactoring context.

5.5.3 Data Sources

To conduct our data analysis, we collected experimental data from different data sources: *interviews with the participant*, and the *generated artifacts* for the API design. We combined the data obtained via these data sources to compensate their strengths and limitations. We describe each data source as follows.

- **Interviews with the participant:** We conducted a total of 4 interviews with the participant of our study. They were all scripted interviews, intending to understand the participant's opinions and perceptions about that phase of the study. Each interview lasted about 20 minutes. In these interviews, the participant was asked about the utility, and the ease and difficulty of using the Colloquy method. He was also asked how his previous skills might have influenced certain design decisions. Each interview was contextualized with the phase that had just happened.

- **Generated artifacts:** At each stage of our study, a set of specific artifacts were created, including personas, scenarios, models and API interfaces. These data were collected so that they could later be analyzed and discussed with the participant. We required the participant about each artifact to discover how Colloquy might have helped him in the creation.

5.5.4 Data Analysis Procedures

Qualitative Data Analysis: Our study was qualitative as our analysis was very based on the interviews we conducted. We tried to cross the data from the interviews with the artifacts generated to understand if the participant was making proper use of the method. All the interviews were transcribed for an in-depth analysis of the participant's statements.

After the transcription, we attempted to find and group statements into the following categories: advantages, disadvantages, and improvements related to Colloquy. Our objective was to see what the participant could point out as positive and negative points of our method, showing how useful it is and also opening ways for future improvements. The results of this in-depth analysis of the participant's statements are present in section 5.7.

5.5.5 Phases of the Study Execution

5.5.5.1 Phase 1 - API Design Following Another Method

In the first stage of the case study, we searched in the technical literature for methods and guidelines for supporting the API design process. Among the papers found, we selected the approach proposed by Henning (21) due to the fact that it was one of the few that offered a sequence of steps and guidelines for API design. Moreover, this work was the one with highest number of citations. The approach is composed of a set of guidelines and steps for successful API design. Besides, the approach also lists problems of a poorly designed API and the advantages of relying on the guidelines he has established. Below, the set of guidelines proposed by Henning:

- An API must provide sufficient functionality for the caller to achieve its task.
- An API should be minimal.

- APIs cannot be designed without an understanding of their context.
- General-purpose APIs should be policy-free, special-purpose APIs should be policy-rich.
- APIs should be designed from the perspective of the caller.
- Good APIs don't pass the buck.
- APIs should be documented before they are implemented.
- Good APIs are ergonomic.

Before executing the first step, we performed a quick training on the Henning's method with the API designer. In this training, he was guided to use his skills and follow Henning's work for creating the API. After two months, the participant delivered the class diagram of the API, and a set of sequence diagrams representing the user interaction with the API. It is worth noting here that the participant was not exclusively working on our study. Then, we conducted a second meeting with the API designer for collecting the participant's opinions on the method used in the first step and on the quality of the API generated. This phase took the participant six weeks.

5.5.5.2 Phase 2 - API Design Following Colloquy

At the last hour of the second meeting, the participant was introduced to the concept of conversational API and Colloquy. In this meeting the participant was trained to perform the first phase of our method. More specifically, we introduced the participant to the concept of personas and scenarios, giving examples based on API for other contexts. We also have shown him the table with the guiding questions to assist him in the task.

After the first meeting, the participant was guided to create the personas and interaction scenarios for his API. After four weeks, the participant delivered the personas and scenarios created. We then interviewed the API designer to collect the his experience regarding the use of Colloquy. The same happened for the next two phases. First, we would have a meeting with the training and presentation of examples of how to use the method in that phase. Then, the participant carried out the method execution, and later we would have another meeting to collect the participant's opinions and experiences. In phase 2, it took the participant four weeks to perform the modeling, and in phase 3, it took the participant two weeks.

After all three phases of the method, the participant delivered a redesigned API, with significant differences from the original version. In the following section, we will show the API's snippets pointing to the main features

that were modified or created. We will highlight those features that prompted the API conversation, establishing a more fruitful interaction between user and API. In section 5.7.3 we discuss about how the execution of Colloquy after a previous API design may have impacted the validity of the study, and how we face this threat.

5.6 Results

In this section, we will show the results achieved from our case study. We will show the artifacts generated by the participant, step by step, while he or she performed the three steps proposed by Colloquy.

5.6.1 Personas and Interaction Scenarios Created

In the first stage of the method, the participant conceived three different personas who would use his API. For each persona, the participant was able to reflect on two different interaction scenarios.

5.6.1.1 Persona 1: John - Expert Software Engineer

The first person created was John, an expert software engineer with several years of programming experience. John, who leads a development team, is very skilled with the Java programming language and in using IDE for refactoring source code. However, he leads a team composed of novice developers. In this way, he needs a tool for assuring the quality of the code generated by his team. To do so, this tool should provide an API where John can define his quality criteria through custom code refactoring. The participant thought of two scenarios to assist this persona.

- *Refactoring in the code review process*
- *Recommend refactoring for your team*

5.6.1.2 Persona 2: Philip - Experienced Freelance Programmer

Philip has a degree in computer science and is an experienced freelance programmer. During his five years working as a freelancer, Philip has accumulated code elements to be reused. However, Philip increased his concern with quality in his last project, in which he was required to follow design patterns. This way, to reuse his old work, Philip will need to refactor it to suit his new

way of programming. Thus, Philip needs a tool that can support him on refactoring programs in his old code bases. The participant thought of two scenarios to assist this persona.

- *Automating refactoring*
- *Refactor old source code*

5.6.1.3 Persona 3: Katarina - Inexperienced Programmer

Katarina is an inexperienced young programmer studying systems analysis. Katarina is not yet familiar with design standards and has great difficulty for improving the structural quality of her code. Therefore, Katarina needs a tool for helping her to overcome this challenge. As Katarina is still studying software design, she wants a tool that would work as tutor, explaining the need for each suggested code modification. The participant thought of two scenarios to assist this persona.

- *Refactoring recommendation*
- *Learn refactoring-driven programming*

5.6.1.4 Discussion about Personas and Interaction Scenarios

One can see the participant was able to think of different personas and different interaction scenarios. Experienced and novice programmers were described. Consequently, the API designer need to design different dialogues to support the needs of these personas. For example, Philip, the more experienced persona will need a more efficient and direct API. The persona who is learning how to program, Katarina, will need a more verbose API, with more explanations and more sophisticated dialogues. These needs will be reflected in the final interfaces, as we will see in the following subsections.

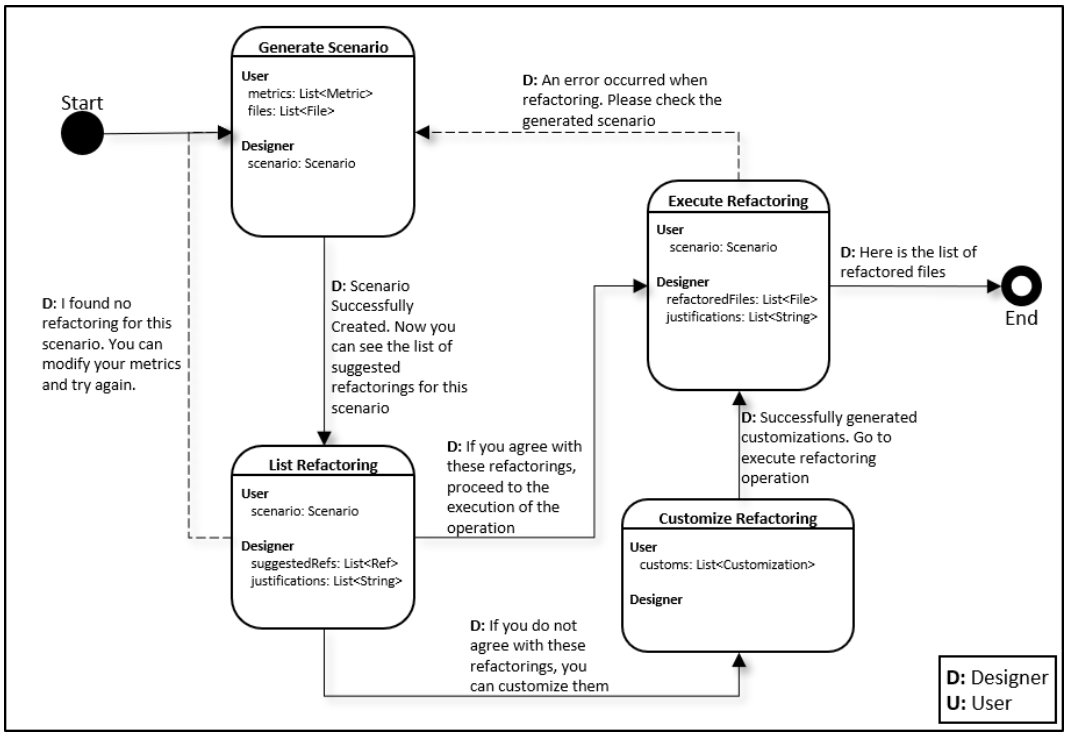

Figure 5.5: Modeling the Refactoring API Interaction

5.6.2 Diagrams

After creating the personas and defining the interaction scenarios, the participant proceeded to Colloquy's next stage. Following the method's procedures, the participant modeled the interaction defined in the previous scenarios using the MoLIC4APIs modeling language. The final model generated is shown in figure 5.5 below.

During the API modeling, the participant was able to think in detail about the interaction flows (or conversations as we call) that the user should perform with its API. In the participant's own words, modeling helped him think about the regressive flows, that is, the possible problems the user would face and how they could be solved. In Figure 5.5, we see the conversation scenes, conversation flows, and the dialogues exchanged between designers and users, represented in the OpenAPI language (section 5.4.2).

5.6.3 API Interfaces

The last stage of Colloquy is the definition of the API interfaces. It is important to note that we will not show the complete API. Instead, we focus on presenting and discussing the API parts that show how Colloquy was relevant in this stage. The listing 5.1 shows the source code of the interface and a short

documentation above them. This documentations aims to explain what each operation does. Besides, the documentation also explains about the parameters and return values. In the following subsection, we will discuss how the method has impacted the creation of these interfaces from the designer's point of view.

Note that four different operations were generated in the proposed API. Each operation was the result of an interaction scene from the modeling of the previous step. Thus, for each operation, the user speeches became parameters, and the designer speeches were grouped into return objects, including the transition speeches between scenes. Below, we present the four public methods created in the final stage of Colloquy. In section 5.6.4, we discuss which of these elements represent improvements concerning the API generated using Henning's guidelines (21). In section 5.6.5, we discuss which aspects of conversation are present in the API and its distribution among static, dynamic, and metalinguistic signs.

```
/** Generate a refactoring scenario by defining a set of
   metrics and selecting a set of files to be analyzed.
   Return the generated scenario. */
public Scenario generateScenario(List<Metric> metrics,
   List<File> files)


/** List the suggested refactorings and the
   justifications. The result of the list comes in the
   format: Map<Refactoring, String> */
public ListRefactoringObject listRefactoring(Scenario
   scenario)


/** Add a set of refactoring customizations to the
   scenario. Returns a message with the result of the
   operation. This message can indicate the operation's
   success, indicating the user which next operation he
   should perform. Alternatively, it can indicate an
   error message explaining to the user how to recover
   from the error and restart the operation.*/
public String customizeRefactoring(Scenario scenario,
   List<Customization> customizations)


/** Execute the refactoring in the scenario, following
   the defined metrics. Return a set of newly modified
   files and an explanation of the modification made. The
    return has the following format: Map<File, String> */
public ExecuteRefactoringObject executeRefactoring(
   Scenario scenario)
```

Listing 5.1: Generated API for Refactoring Java Programs Language

5.6.4 Improvements After Using Colloquy

When the participant started development using our method, he already had an original version of his API, based on the use of Hening's guidelines (21). In this section, we highlight the improvements that came with the method and their respective reasons.

The first improvement was the reflection the participant made about the

need to give his user a reason for each refactoring his API was suggesting. Inspired by the persona Katarina (section 5.6.1.3), which only emerged after using Colloquy, the participant realized the need for a better explanation of this feature. In the original version of the API, the return of the *"listRefactoring"* operation was just a set of *"Refactoring"* objects that listed the changes required to be applied in the program. However, after considering the scenarios provided for persona Katarina, and its primary purpose of learning about refactoring, the participant modified the return to add a *"justification"* field for each refactoring offered by the API.

The second significant modification that came up with Colloquy was the *"Refactoring Scenario"* concept incorporated into the API. According to the participant, this idea arose during modeling using MoLIC4APIs. When realizing the interaction flow created, the participant identified the need to create an object that was a connecting factor between the parts of the API. Consequently, the concept of refactoring scenario emerged, with which the user would interact throughout the navigation until the completion of its goal. One can see that a *"Scenario"* object is present in all API methods.

The last significant change we highlight here is the creation of operations *"listRefactoring"* and *"executeRefactoring"* as a result of the split of the previous method *"doRefactoring"*. This division was a consequence of modeling using MoLIC4APIs. When invited to think of modeling as a conversation, the participant could realize that the original method should be split in two to improve the API conversations. Moreover, by analyzing the current interfaces, we can realize that, in fact, the conversation flow has become more natural between users and designers. Moreover, this spliting of the original method into two methods brought a new feature that the participant had not thought of before: the possibility of one user just wanting to discover the refactorings offered without necessarily performing them. A user with a learning interest in refactoring could benefit from such a feature.

5.6.5 Interface Conversations Aspects

This subsection highlights the conversational elements of the generated API and how they enhance the communication of the design rationale to the users. In the previous subsection, we have already discussed some new API elements that emerged from Colloquy's execution. Now we will talk again about these elements but from the perspective of the contribution to the conversation improvements.

Detecting structural problems in software may be not a cut-and-dried

task. Even with some ascertained metrics, there will always be margins for interpretation of the possibility of a code smell existence. Thus, in cases like these, the creation of resources in the interface that captures the user's attention to possible disagreements about an operation is fundamental. Therefore, our approach made the participant realize this need and added a field of *"justification"* to the return of the *"listRefactoring"* operation. By adding this feature, the designer is exposing his rationale to the user. Thus, the user can understand the application of the metrics that have been selected, and the refactorings offered by the API as a consequence. This feature represents an advance in API conversation over the first version.

Another important conversation characteristic that we highlight in the API is the creation of the *"generateScenario"* method and the detachment of two methods that were previously grouped into a single operation, *"listRefactoring"* and *"executeRefactoring"*. In the previous API version, the user did not have the option to view all the refactorings that would be executed before actually executing them. Using Colloquy, the participant realized that splitting this operation in two could enhance the conversation with the user. Thus, the API exposes the user to its internal behavior, i.e., which refactorings were selected, so the user can execute them if he desired. Besides, it is important to note that all this conversation happens in the interface, even if a more detailed description of each operation ends up happening in the complete API documentation.

Finally, we highlight the flow of conversation that the API began to promote with the user. The API users will be able to interact with the API in continuous dialogues until they achieve their goals. This API global feature is also a characteristic that defines it as a conversational API. For instance, if an experienced user already familiar with the API wants to perform a refactoring, he can follow an efficient conversation flow. He can create a refactoring scenario and then go straight to the refactoring execution, without going through the list of selected refactorings. On the other hand, an inexperienced user who wants to understand the refactoring applied can follow the whole flow of operations step by step, through the dialogues pre-made by the designer.

5.7 Discussion

In this section, we discuss the results found from the participant's point of view, i.e., the designer of the API generated in our case study. We show the benefits of using Colloquy listed by the participant during the interviews we conducted. Besides, we also discuss some opportunities to improve Colloquy

that has emerged from participant opinions and experiences. Finally, we discuss some threats to validity of our case study.

5.7.1 Colloquy Method Benefits

Developing Empathy with Users: throughout the interview, the designer emphasized that he developed his empathy for the possible users of the API. His report indicates that such improvement is mainly a reflection of the execution of the first stage of the method. When asked about the main influences of creating personas and scenarios, the designer stated that *"When you are building an API, you have the concept to you. What the API offers to you. When you start creating characters, you will think about what my API will offer that persona, in that situation, in that specific scenario. Then you start to expand the API to serve a larger set of people"*. So, we may notice that the participant put the user in the foreground, emphasizing the quality of use of the API interfaces. This is a consequence of creating empathy for the user, i.e., trying to understand the needs and how to help the user on solving problems. This consequence did not arise when using the Henning approach since its guidelines are not directly related to the users' profile and needs.

Modeling Conversation with Users: The designer founds advantage on modeling the API interaction as a conversation. The MoLIC4API language helped him thinking of the API operations: *"By using the modeling language, you can have the vision of the whole interaction. That kind of vision I don't think I would have if I didn't make the models. Modeling is related to what the user will ask for API and how it will answer. And you can associate that to method calling, parameter passing, values and so on"*. This report also indicates that Colloquy influenced the designer to think about the conversational capabilities that should be offered by the refactoring API. Since Henning does not offer an API modeling step in his approach, it was not possible for the participant to reflect on the conversations and the interaction flow of his API.

Identification of New Requirements: Another point that the design highlighted as relevant when using Colloquy was developing the ability to extend the requirements met by the API. According to him, thinking about the conversations that the API should provide to the users allowed perceiving that different users might need different conversations, leading to identifying new requirements for the API: *"It was noticeable that what I was offering in the API was something limited and focused on a type of persona. By making more*

personas, I could realize that I can provide more details and more requirements for users. Thinking about the personas made me expand what the API could offer. It made me think things that I had not thought about at first."

Created Models as API Documentation: Another significant contribution of Colloquy was using the generated models as part of the API documentation. It was the participant who pointed to this possibility in the last the interview. Asked about the advantages and disadvantages of using the modeling language to design the API interaction flow, the participant replied: *"Let us say I have a development team, and we are in the modeling phase of this API. If I showed a UML diagram, the team might not be able to see an interaction flow. So what I think would be more practical is that the moment I was presenting my functionalities, I would show this interaction through MoLIC4APIs. Even for the API user. If I show UML the user may not understand quickly, but it is much more practical and clear to understand all the flow if I show the MoLIC4APIs diagram"*. Thus, we believe that the generated models also contributed for increasing the quality of the API documentation.

These aforementioned benefits indicate that Colloquy is feasible, bringing effective contributions for designers developing better APIs. Such contributions perceived by the designer encompass the three steps of the method, resulting into improvements in the designer's conversation with the API users. It enhances the quality of the API from the standpoint of its completeness and structural quality. At the moment, the API is in the implementation phase of the internal functionalities. After the conclusion of this stage, we intend to conduct experiments with different API users. In these studies, we will also seek to analyze the perception of the API's quality of use from the users point of view.

5.7.2 Colloquy Drawbacks

The execution of Colloquy was more complex and time-consuming: as Colloquy consists of 3 laborious steps, the time taken to execute the method was considerably higher than the time spent on the previous design. Furthermore, for a designer who lacks knowledge of the HCI techniques we use, the method can be even more costly as it will have a disadvantageous learning curve in the short term. However, we believe that Colloquy can bring superior advantages that would pay off in cost benefit after learning, since the API generated after the execution of the method had a great gain in quality

and conversation, as discussed in the section 5.6.4.

Prioritize which guidelines are most important: During the interviews, we noticed that the API designer faced some difficulties in applying Colloquy. The first major difficulty for the participant was in creating the API personas and scenarios. Although he reported that the guiding questions significantly helped him to think and write about personas and scenarios, the participant pointed out that there are too many aspects that should be considered in each question. Several aspects that were listed in the guiding questions did not make sense for the participant API's context. For example, for the participant, questions about "culture" or "demographic data" are not relevant for characterizing the set of API personas. In contrast, questions such as academic background are extremely important, once he believe that academics are more prone to better understand code refactoring than practitioners. Thus, possible opportunities of improving Colloquy addresses reducing the number of guidelines or even the prioritization of which aspects should be more relevant.

Colloquy needs computational support to be more viable: Another great difficulty was the creation of the MoLIC4APIs models. The participant complained that the method does not offer a tool to support the creation of the models. Additionally, memorizing exactly how to design each element of the model was considered inefficient by the participant. In his words, he would have significantly benefited from a tool to assist him in the construction of the models. Besides accelerating the modeling process, this tool would reduce the cognitive load of who is modeling the API. The participant reported much difficulty in creating the models since there was no tool that offered the elements ready for him just to drag and drop. Other approaches, such as UML-based modeling, already have such computational support. However, we will address this challenge in future work.

5.7.3 Threats to Validity

We planned first to perform the API design with an existing approach (21) and then redesigning the API with our approach. Our intention was to see what new improvements our approach could bring to the design process. However, we are aware that this methodology has a learning bias as the designer have already acquired previous knowledge with the use of other approach. In this way, we have instructed the designer to throw away the API

design produced in the first phase and redesign the API with Colloquy from the scratch. Moreover, we plan conducting new studies following alternative designs. Besides, we intend to evaluate our method without a comparison with existing methods, ensuring that all API design emerged from the use of Colloquy.

Another threat to validity of this study addresses its restricted context: a single API designed by a single professional. We know that more studies are needed so that we can generalize our approach to be applicable on a large scale industrial setting. However, case studies such as the one we design and performed are very complex and require a lot of research effort. We are reporting this our first result and will continue to conduct more studies on evaluating and improving our proposed method.

Our study participant had a well-defined research goal and precise ideas about the API needs. We believe this is another threat to the validity of our study. It brings a positive bias on our method, since the participant may already have a vision about which API he wanted to build. However, we seek to reduce this threat by always making it clear to the participant that he should make transparent his perceptions about Colloquy. Moreover, he always explains what the method helped him think and improve in the API design, highlighting what would not have been possible without the method.

5.7.4 Colloquy and the Software Development Process

We believe that our method can be applied to any software development process. In an agile method, our method could be used, at each sprint, to generate a desired piece of API. The MOLIC4API model itself is flexible to accommodate and evolve with each sprint as new API requirements are implemented. As with any other process of documentation of requirements and functionalities, if the whole team is in tune and able to execute the method, the discussions about personas and modeling can be carried out as a team without problems. Diagrams like MOLIC4API are excellent tools for collaborative discussions in software development processes. Thus, we believe that Colloquy can be inserted in a development process under any kind of methodology. However, we highlight here that no study has been conducted within a more extensive software development context. Our study focused on qualitative analysis and could not detect our method's entry restrictions in a software development team.

5.8 Conclusion

In this paper, we present a method to support the design of conversational APIs. Our method places user interaction with the API as a priority in the process of designing and modeling an API. In this sense, it is important to highlight that the evidence from our case study points out that our method's benefits extrapolate the conversational aspect. However, we understand that other quality aspects of APIs are also important and should be appropriately balanced and prioritized by their designers. In this sense, we understand that our method can be combined with software engineering efforts that traditionally focus on other quality aspects, such as structural quality and performance.

The evidence from our case study suggests that the method has the potential to contribute to improving user interaction with the API. As future work, in the short term, we plan to verify that the API created with our method actually has a better quality of use from the users' point of view by conducting observations studies and interviews with potential users of the generated API. Next, we plan to conduct case studies with API projects in other domains. Throughout these studies, we intend to refine the method, developing a methodology that is instantiable through a computational tool. Thus, we aim to disseminate and evaluate its use on a large scale.

5.9 Summary of Chapter 5

In this chapter, we present Colloquy, a method to support the design of conversational APIs (section 5.4). In addition to the method, we also present a case study that we performed, applying the method to a real case of API development (section 5.5). In this case study, we worked with an API for code refactoring in Java. One participant applied our method and obtained satisfactory results in designing his API.

In this chapter, we discussed the method usefulness and innovation in comparison with state of the art (section 5.3). Given the need for an API to be conversational, the designer of such API can benefit from the method to enhance his API conversation. Thus, improving its quality of use and reducing the problems of misunderstanding by the user. We also show how Colloquy can positively impact the designer himself, making him reflect on his users and their needs better (section 5.7). Another point that we listed in this chapter was the need for further application studies of the method, as well as the necessary improvements that have already emerged from the case study results. Finally,

we point out as future work the need for computational support to the designer in applying the method (section 5.8).

6
Conclusion

As we have seen in this thesis, there is an urgent need to address the problem of lack of API conversation (3). Despite their relevance to software development, APIs still represent a significant challenge of use and learning for programmers. Pragmatic conflicts between users and designers require APIs designers to explain their design rationale beyond documentation to properly help users learn how to use their APIs effectively and efficiently (4). For that, the definition of API conversations is a promising approach in dealing with the problem.

We believe that through the work done in the context of the doctorate defended here, we were able to provide a systematic way for conversational software APIs design. We show that putting all design rationale only into the API documentation can cause serious misunderstandings by the user (3). Furthermore, our conceptual framework provides the appropriate theoretical support (4), while our design method provides the necessary technical and methodological support (6). Moreover, we conducted two major empirical studies indicating that our proposed solutions can help the designer in the design process (6, 5).

Using a semiotic approach, and focusing on communicability (13), our work went beyond those anchored in usability (49, 31, 28), in two-points. First, we take a different look to a case that usability does not properly address: the problems of disagreement in pragmatic usage situations. Second, we offer support for API design, in opposition to studies that offer alternative proposals to help the use of poorly designed APIs (24). We believe that the high differential of our thesis, in opposition to the existing literature, is to have focused on the root of the problem, i.e., the API design process.

The main contributions and their possible impact on the state-of-art and the state-of-practice are described as follows.

- *Characterization of an API as a mediating artifact of conversation between users and designers*

  In this thesis, we discussed the fact that an API plays a role in the mediation of conversation between users and designers. We also discuss

the understanding that APIs should have adequate communicability for better pragmatic adequacy (section 2.5). We show that putting all design rationale only into the API documentation can cause serious misunderstandings by the user. From these understandings, we define that an API needs to be conversational to be adequate in different contexts of use.

- *Characterization and classification of conversational APIs*

  We present and discuss in this thesis a conceptual framework for the characterization and classification of conversational APIs. We present the conversation as a new perspective to see the interaction between programmers and APIs. Based on the Semiotic Engineering theory (13), we describe the different classes of signs and how we can adapt them to the context of the communicability of software APIs. Furthermore, we demonstrate how our conceptual framework can be applied to assist designers in defining conversations in their APIs. An API designer who has the knowledge proposed by our framework will be better able to decide on the distribution of the conversation between the three different types of signs present in an API.

- *A method for conversational APIs Design*

  We also present in this thesis a method for designing conversational APIs. We demonstrate in detail how this method should be used in practice and how it can positively impact API design processes in industry and academia. Moreover, we exhibit and discuss the results of a case study conducted with our method on designing a source code refactoring API. Although we have no results from the API users' point of view, we have satisfactory results from the designer's point of view. These results reveal to us that this is a promising method that can benefit the software engineering community in creating better and more usable APIs.

- *Report of two empirical studies conducted in supporting the design of two different conversational APIs*

  We have reported two long-term empirical studies in the conversational API design process. These studies are innovative because technical literature lacks in these types of work. The results and discussions presented in these studies may inspire other researchers and practitioners' decisions about software APIs design processes.s

Despite the contributions cited above, our work has some limitations that need to be addressed in future work:

- *Our approach can be complex and costly for some API designers profiles*

  Our entire approach is anchored in Semiotic Engineering. We dedicated to building a method and a conceptual framework that would abstract much of the theory's concepts. Still, we consider it necessary that a designer interested in using Colloquy has the basic knowledge of Semiotic Engineering. Therefore, a designer without any knowledge of HCI and Semiotic Engineering may have a disadvantageous learning curve. However, we believe that with adequate computational support, this limitation can be smoothed.

- *Our approach lacks computational support*

  As mentioned in the previous item, adequate computational support can help an inexperienced designer with the difficulties he would face. However, in the current state of our research, we do not have any tools that can fulfill this role. In the paragraphs below, we will discuss this need as future work.

6.1 Overall Studies Reflection and Threats to Validity

This thesis presents four studies, where three are empirical studies, and one is an analytical study. All studies converged to the conclusion we need to prioritize the conversation as an essential quality in designing an API. In chapters 2 and 3, the first two studies point to problems that would be a consequence of inappropriate conversation. APIs with low conversations cause misunderstandings in their users and, consequently, bugs in the final software. Through these studies, we show how a process of API development based on the concepts that we defend here could improve the quality of use.

In chapters 4 and 5, the studies were searching for design solutions for the problems we identified. We chose to perform two qualitative empirical studies, with different natures and domains, to explore in-depth possible solutions to support designers in creating conversational APIs. Besides the method evaluation, which we discussed in section 5.6, these studies also showed us how urgent it is to offer techniques and tools to support the API designer. In both studies, we realized that API designers do not usually follow traditional Software Engineering methods to improve their APIs' quality. Thus, APIs are usually produced with low quality and do not meet the needs of conversation with their users. As a consequence, this can lead to misuse and even disuse of the API.

Regarding the general threats to validity of this thesis, we can list two that are considered more important: the absence of an empirical study with

the Colloquy method being performed within a broader context of Software Engineering, and the lack of quantitative studies.

The two studies carried out with Colloquy, the one on conception (chapter 4), and the one on evaluation (chapter 5) were not in a software development scenario that already had some Software Engineering method or process in the development team. However, according to what we discussed in section 5.7.4, we defend that Colloquy can be added in an organic way to any software development process. We also state that Colloquy's techniques are already often used in other stages of software development. For example, the modeling language as a conversation (MoLIC) is already employed in Human-Computer Interaction processes. So, we know that more studies need to be done with Colloquy in an environment with well-defined Software Engineering methods and processes. However, this does not disqualify all the contributions that Colloquy brings to the conversational API design process.

The lack of quantitative and large-scale studies in the industry is also a threat to validity. However, we argued that in this doctoral thesis, the focus was on understanding how to help the software API designer on improving the conversation of his interfaces. Therefore, we designed only qualitative studies, which give us better in-depth results about the designer's views and perceptions when using our method. As discussed in section 5.7, we believe that to test Colloquy on a large scale will require some computational support. In the following section, we will discuss this computational support as future work.

6.2 Future Work

As future work, we divide our research interests in three terms: short-term, mid-term, and long-term.

In the short-term, we want to explore the reception of potential users of the APIs created using our approach. We have the Java code refactoring API that was designed following the conversational API concept proposed in this thesis. The designer followed step by step our proposed method during the API design process. Currently, the API is in the final phase of implementation, and soon we will be able to conduct studies with users. In these studies, we intend to compare the API generated with those already available to users. We believe that with our conversational API approach, where the user will be able to understand and even act on the refactoring performed, we will achieve greater success in the appropriate use and the user satisfaction with the operations performed.

In the mid-term, we intend to develop a computational tool to support

the method execution. To do this, we need to define which tasks of the method need more support and would benefit from a computational tool. We believe that the modeling step is the one that most needs this help nowadays. We intend to go in search of tools that can help the designer in the construction of models using the language proposed by Colloquy. We will need to adapt some existing tool or create our own.

Another future work that we plan on this same topic is the availability of the models created in phase 2 of Colloquy as a sort of documentation add-on. The same tool that will help the designer could provide a navigation functionality on the model created. This would help the user on discovering the conversation flows that he must follow to achieve the desired results. Thus, the modeling would serve as an interactive API documentation.

In the long-term, we believe that the concepts and tools we use in our thesis can be explored in the context of artificial intelligence algorithms. Recently, a new field of research has emerged in this area, the explainable artificial intelligence (XAI). Some artificial intelligence techniques, such as machine learning (ML) models, for example, are usually seen as "black boxes" by their users (25). They are essentially mathematical models, operating through a pattern search on a vast amount of data, which does not necessarily have a human-relevant relationship. Consequently, understanding the internal logic of these models is a complex task, both for designers and users.

With APIs playing a central role in the creation of ML models, we want to investigate how communication takes place between designers and users of ML APIs. As the main result of this future work, we hope to offer, through our method (with adaptations), a methodological tool that allows API designers to explore and understand the true meaning of explainability from the user's point of view. Therefore, we can offer a way to ML API designers to expose their design rationale about the explainability of ML models.

A
Colloquy Execution Example

In this appendix, we illustrates the step-by-step execution of Colloquy for the design of a date and time API. First, we start by creating personas and interaction scenarios. We describe one persona and two possible interaction scenarios. Second, we present a conversation model using the MoLIC4APIs language proposed by Colloquy. Finally, we show an interface proposal for the API designed.

A.1
Personas and Interaction Scenarios

Persona: Daniel (Java programmer)

Daniel is a systems analyst and an experienced computer programmer. Daniel is Brazilian but is used to programming in APIs and languages written in English, even though he is not fluent. Daniel needs an API to work with "date" type objects. For Daniel, the date is synonymous with the Gregorian Calendar, those whose months go from January to December, divided in weeks of 7 days. Daniel hopes that the API can help him calculate the differences between dates and perform operations such as adding dates and periods. Since the system will be used in various parts of the world, Daniel hopes that the API can help him with time zone calculations. Daniel loves programming in Java, a language he has over ten years of professional experience. Besides Java, Daniel is also as experienced as in SQL for database queries and JavaScript scripting language. He has worked with other scripting and object-oriented languages and has also contacted other date APIs such as JavaScript, MySQL, and Java7.

Interaction Scenario 1: Bill of Exchange Registration

Daniel needs to implement new functionality for generating billing tickets in his company's new sales system development. A client can pay up to 3 installments of their purchase amount in the company's sales policy without interest by generating three billing tickets. Daniel then needs to calculate the date generated on the bills at the time of the purchase's finalization. To do

this, Daniel will rely on the date API of his preferred programming language. Daniel's input data is the purchase date, and he needs to calculate the billing deadlines, the first being one month after the purchase date, the second two months after the purchase date, and the third three months after the purchase date. When interacting with the API, it can create a date object and use a method to calculate the expected date adding the necessary months. After operating, he generates the bills and saves the due date in the database.

Interaction Scenario 2: Calculating the product delivery time
Now, Daniel needs to implement the delivery time calculation in the system. According to the company's business rules, Daniel should perform the delivery time calculation counting seven working days after the purchase date. Besides, a special rule must be applied if the purchase was made by noon, the day of purchase counts as the first working day. Otherwise, the account should start only on the next working day. Daniel will use a date API from his programming language to perform the calculations. Daniel will create a date object with the help of the API. Then he will check if the time of purchase was before noon or after, to apply the correct sum of days on time. As the company Daniel works for is a multinational, he needs the API to deal with the time zone issue. The database time zone is not the same as the customer time zone. The API he will use needs to understand these differences when performing the calculation. In the end, Daniel will ask the API to add 6 or 7 working days to the purchase date and will save the result to the database.

A.2 Conversation Modeling

In the diagram below, there are the main elements of the modeling language we propose. The conversation scenes are represented by the rectangles, where we have the title of the scene, the dialogues, and the arrows indicating the conversation flow. Straight arrows are indicating progressive flow, and dotted arrows are indicating regressive flow. Within each scene, the dialogues are represented following the OpenAPI syntax: *[name_parameter] : [type_parameter].*

In this diagram, we propose a conversation modeling that covers the proposed scenarios for the persona Daniel. Certainly, date and time API can and should have many other conversations. However, the purpose of this appendix is to illustrate and exemplify how Colloquy can be used to improve the API conversation during the design process.

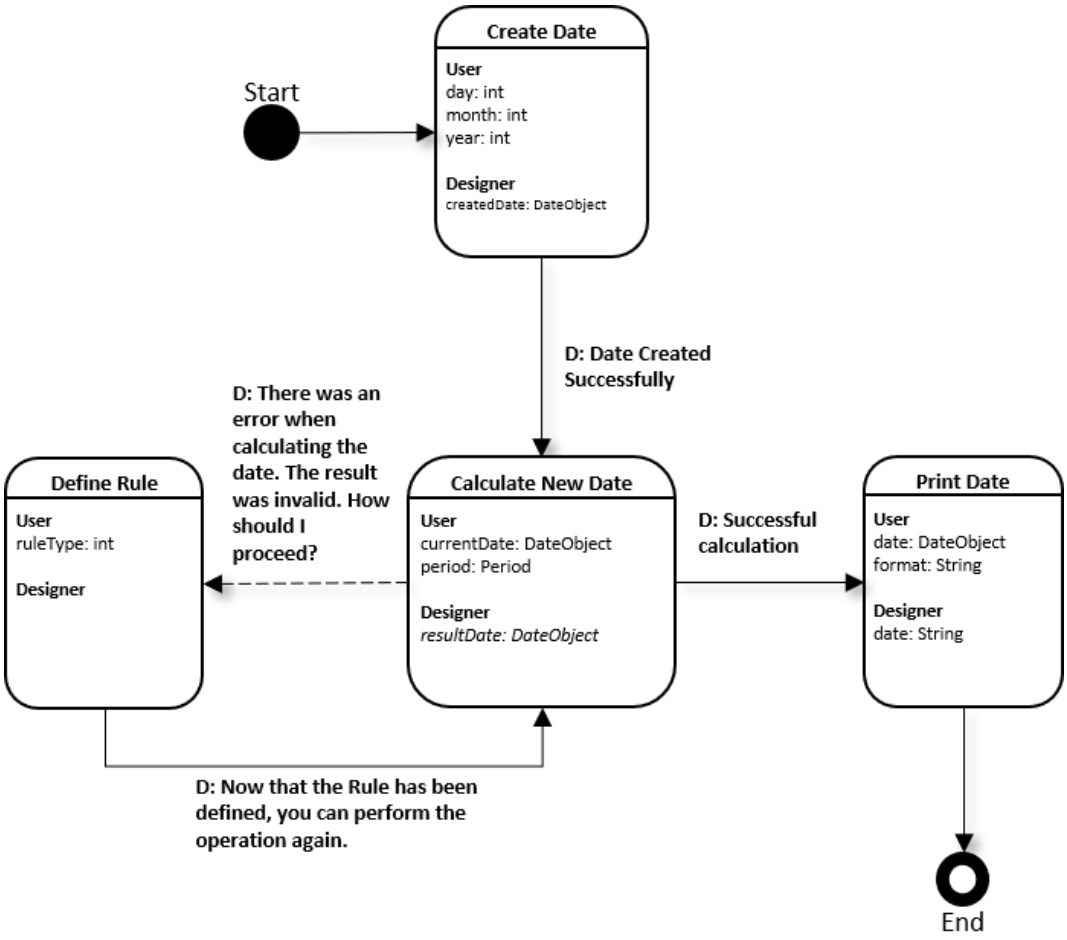

Figure A.1: Modeling the Date and Time API Conversations

A.3 Interfaces Implementation

In this section, we present a proposal for a possible interface for the generated API. Each API method is derived from a scene modeled in the previous step, following Colloquy's recommended. Each conversation has been converted to parameters or method returns. Below are the proposed interfaces:

```
//Creating a New Date Scene
DateObject createNewDate(int day, int month, int year)

//Calculate a New Date Scene
DateObject calculateNewDate(DateObject currentDate,
   Period period)

//Define Exception Rule Scene
Void defineExceptionRule(int ruleType)
```

```
//Print Date Scene
String printData(DateObject date, string format)
```

Listing A.1: Generated API for Date and Time Operations

Bibliography

[1] AFONSO, L. M.. **Communicative dimensions of application programming interfaces (APIs)**. PhD Thesis, Programa de Pós-Graduaçao em Informática of the Departamento de Informática . . . , 2015.

[2] AFONSO, L. M.; BASTOS, J. A. M. D.; SOUZA, C. S. D. ; CERQUEIRA, R. F. D. G.. **The Case for API Communicability Evaluation: Introducing API-SI with Examples from Keras**. CoRR, abs/1808.05891, 2018. _eprint: 1808.05891.

[3] BASTOS, J. A.; AFONSO, L. M. ; DE SOUZA, C. S.. **Metacommunication between programmers through an application programming interface: A semiotic analysis of date and time APIs**. In: 2017 IEEE Symposium ON Visual Languages AND Human-Centric Computing (VL/HCC), p. 213–221, Oct. 2017. ISSN: 1943-6106.

[4] BASTOS, J. A.; MELLO, R. M. ; GARCIA, A. F.. **A conceptual framework for conversational apis**. 2020.

[5] BASTOS, J. A.; MELLO, R. M.; GARCIA, A. F. ; CERQUEIRA, R. F. G.. **On the support for (re)designing conversational software apis: An action research study**. 2020.

[6] BASTOS, J. A.; MELLO, R. M. ; GARCIA, A. F.. **A method for conversational api design**. 2020.

[7] CARROLL, J. M.. **Making use: scenario-based design of human-computer interactions**. MIT press, 2000.

[8] CETINSOY, A.; MARTIN, F. J.; ORTEGA, J. A. ; PETERSEN, P.. **The past, present, and future of machine learning APIs**. In: CONFERENCE ON Predictive APIs AND Apps, p. 43–49, 2016.

[9] CHING, T.; HIMMELSTEIN, D. S.; BEAULIEU-JONES, B. K.; KALININ, A. A.; DO, B. T.; WAY, G. P.; FERRERO, E.; AGAPOW, P.-M.; ZIETZ, M.; HOFFMAN, M. M.; XIE, W.; ROSEN, G. L.; LENGERICH, B. J.; ISRAELI, J.; LANCHANTIN, J.; WOLOSZYNEK, S.; CARPENTER, A. E.; SHRIKUMAR, A.; XU, J.; COFER, E. M.; LAVENDER, C. A.; TURAGA,

S. C.; ALEXANDARI, A. M.; LU, Z.; HARRIS, D. J.; DECAPRIO, D.; QI, Y.; KUNDAJE, A.; PENG, Y.; WILEY, L. K.; SEGLER, M. H. S.; BOCA, S. M.; SWAMIDASS, S. J.; HUANG, A.; GITTER, A. ; GREENE, C. S.. **Opportunities and obstacles for deep learning in biology and medicine**. Journal of The Royal Society Interface, 15(141):20170387, Apr. 2018.

[10] CLARKE, S.; BECKER, C.. **Using the Cognitive Dimensions Framework to evaluate the usability of a class library**. In: IN Proceedings OF THE 15H Workshop OF THE Psychology OF Programming Interest Group (PPIG 2003, 2003.

[11] COOPER, A.. **The inmates are running the asylum:[Why high-tech products drive us crazy and how to restore the sanity]**, volumen 2. Sams Indianapolis, 2004.

[12] DE PAULA, M. G.; BARBOSA, S. D. J.. **Designing and Evaluating Interaction as Conversation: A Modeling Language Based on Semiotic Engineering**. In: Jorge, J. A.; Jardim Nunes, N. ; Falcão e Cunha, J., editors, INTERACTIVE Systems. Design, Specification, AND Verification, Lecture Notes in Computer Science, p. 16–33, Berlin, Heidelberg, 2003. Springer.

[13] DE SOUZA, C. S.. **The semiotic engineering of human-computer interaction**. MIT press, 2005.

[14] DE SOUZA, C. S.; LEITÃO, C. F.. **Semiotic Engineering Methods for Scientific Research in HCI**. Morgan & Claypool Publishers, 2009.

[15] DE SOUZA, C. S.; CERQUEIRA, R. F. D. G.; AFONSO, L. M.; BRANDÃO, R. R. D. M. ; FERREIRA, J. S. J.. **Software Developers as Users : Semiotic Investigations in Human-Centered Software Development**. Springer International Publishing, 2016.

[16] DEHAGHANI, S. M. H.; HAJRAHIMI, N.. **Which Factors Affect Software Projects Maintenance Cost More?** Acta Informatica Medica, 21(1):63–66, Mar. 2013.

[17] 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, CHI '10, p. 2327–2336, Atlanta, Georgia, USA, Apr. 2010. Association for Computing Machinery.

[18] FOWLER, M.. **Refatoração: Aperfeiçoamento e Projeto**. Bookman, 2004. Google-Books-ID: xV2_wAEACAAJ.

[19] GREEN, T. R.. **Cognitive dimensions of notations**. People and computers V, p. 443–460, 1989.

[20] GRICE, H. P.. **Logic and Conversation**. In: SPEECH ACTS, p. 41–58. Brill, 1975.

[21] HENNING, M.. **API Design Matters**. Queue, 5(4):24–36, May 2007.

[22] ICHINCO, M.; HNIN, W. Y. ; KELLEHER, C. L.. **Suggesting API Usage to Novice Programmers with the Example Guru**. In: PROCEEDINGS OF THE 2017 CHI Conference ON Human Factors IN Computing Systems, CHI '17, p. 1105–1117, Denver, Colorado, USA, May 2017. Association for Computing Machinery.

[23] INITIATIVE, O. A.. **OpenAPI Specification**, May 2020.

[24] LAMOTHE, M.; SHANG, W.. **When APIs are Intentionally Bypassed: An Exploratory Study of API Workarounds**. p. 13, 2020.

[25] LECUN, Y.; BENGIO, Y. ; HINTON, G.. **Deep learning**. nature, 521(7553):436–444, 2015. Publisher: Nature Publishing Group.

[26] MCLELLAN, S.; ROESLER, A.; TEMPEST, J. ; SPINUZZI, C.. **Building more usable APIs**. IEEE Software, 15(3):78–86, May 1998. Conference Name: IEEE Software.

[27] MINDERMANN, K.. **Are easily usable security libraries possible and how should experts work together to create them?** In: PROCEEDINGS OF THE 9TH International Workshop ON Cooperative AND Human Aspects OF Software Engineering, CHASE '16, p. 62–63, Austin, Texas, May 2016. Association for Computing Machinery.

[28] MOSQUEIRA-REY, E.; ALONSO-RÍOS, D.; MORET-BONILLO, V.; FERNÁNDEZ-VARELA, I. ; ÁLVAREZ ESTÉVEZ, D.. **A systematic approach to API usability: Taxonomy-derived criteria and a case study**. Information and Software Technology, 97:46–63, May 2018.

[29] MURPHY, L.; KERY, M. B.; ALLIYU, O.; MACVEAN, A. ; MYERS, B. A.. **API Designers in the Field: Design Practices and Challenges for Creating Usable APIs**. In: 2018 IEEE Symposium ON Visual Languages AND Human-Centric Computing (VL/HCC), p. 249–258, Oct. 2018. ISSN: 1943-6106.

[30] MYERS, B. A.; KO, A. J.. **The Past, Present and Future of Programming in HCI**. p. 2, 2009.

[31] MYERS, B. A.; STYLOS, J.. **Improving API usability**. Communications of the ACM, 59(6):62–69, 2016. Publisher: ACM New York, NY, USA.

[32] NGUYEN, D. C.; WERMKE, D.; ACAR, Y.; BACKES, M.; WEIR, C. ; FAHL, S.. **A Stitch in Time: Supporting Android Developers in WritingSecure Code**. In: PROCEEDINGS OF THE 2017 ACM SIGSAC Conference ON Computer AND Communications Security, CCS '17, p. 1065–1077, Dallas, Texas, USA, Oct. 2017. Association for Computing Machinery.

[33] NETO, M. A. C.. **Uma linguagem de modelagem da interação para auxiliar a comunicação designer-usuário**. p. 134.

[34] NIELEBOCK, S.; HEUMÜLLER, R.; KRÜGER, J. ; ORTMEIER, F.. **Cooperative API Misuse Detection Using Correction Rules**. p. 4, 2020.

[35] NIELSEN, J.; MOLICH, R.. **Heuristic evaluation of user interfaces**. In: PROCEEDINGS OF THE SIGCHI CONFERENCE ON Human FACTORS IN COMPUTING SYSTEMS, p. 249–256, 1990.

[36] NIELSEN, J.. **Usability Engineering**. Morgan Kaufmann, Oct. 1994. Google-Books-ID: 95As2OF67f0C.

[37] NORMAN, D. A.; DRAPER, S. W.. **User centered system design; new perspectives on human-computer interaction**. L. Erlbaum Associates Inc., 1986.

[38] OLIVEIRA, J.; GHEYI, R.; MONGIOVI, M.; SOARES, G.; RIBEIRO, M. ; GARCIA, A.. **Revisiting the refactoring mechanics**. Information and Software Technology, 110:136–138, June 2019.

[39] PARNAS, D. L.. **A technique for software module specification with examples**. Communications of the ACM, 15(5):330–336, May 1972.

[40] PEIRCE, C. S.. **Reasoning and the logic of things: The Cambridge conferences lectures of 1898**. Harvard University Press, 1992.

[41] PICCIONI, M.; FURIA, C. A. ; MEYER, B.. **An Empirical Study of API Usability**. In: 2013 ACM / IEEE International Symposium ON Empirical Software Engineering AND Measurement, p. 5–14, Oct. 2013. ISSN: 1949-3789.

[42] PONTES, F.; GHEYI, R.; SOUTO, S.; GARCIA, A. ; RIBEIRO, M.. **Java Reflection API: Revealing the Dark Side of the Mirror**. In: PROCEEDINGS OF THE 2019 27TH ACM Joint Meeting ON European Software Engineering Conference AND Symposium ON THE Foundations OF Software Engineering, ESEC/FSE 2019, p. 636–646, New York, NY, USA, 2019. Association for Computing Machinery. event-place: Tallinn, Estonia.

[43] ROSSON, M. B.; CARROLL, J. M.. **Usability Engineering: Scenario-Based Development of Human-Computer Interaction**. Morgan Kaufmann, 2002. Google-Books-ID: sRPg0IYhYFYC.

[44] SANGIORGI, U. B.; BARBOSA, S. D.. **MoLIC designer: towards computational support to hci design with MoLIC**. In: PROCEEDINGS OF THE 1ST ACM SIGCHI SYMPOSIUM ON Engineering INTERACTIVE COMPUTING SYSTEMS, EICS '09, p. 303–308, Pittsburgh, PA, USA, July 2009. Association for Computing Machinery.

[45] SANTAELLA, L.. **O método anticartesiano de C. S. Peirce**.

[46] SANTOS, A. L.; MYERS, B. A.. **Design annotations to improve API discoverability**. Journal of Systems and Software, 126:17–33, 2017. Publisher: Elsevier.

[47] SOUDERS, S.. **High Performance Web Sites: Essential Knowledge for Front-End Engineers. sl**. O'Reilly Media, 2007.

[48] SPECIFICATION, U.. **About the Unified Modeling Language Specification Version 2.5.1**.

[49] STYLOS, J.; CLARKE, S.. **Usability Implications of Requiring Parameters in Objects' Constructors**. In: PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON Software Engineering, ICSE '07, p. 529–539, USA, May 2007. IEEE Computer Society.

[50] 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, SIGSOFT '08/FSE-16, p. 105–112, Atlanta, Georgia, Nov. 2008. Association for Computing Machinery.

[51] STYLOS, J.; GRAF, B.; BUSSE, D. K.; ZIEGLER, C.; EHRET, R. ; KARSTENS, J.. **A case study of API redesign for improved usability**. In: 2008 IEEE Symposium ON Visual Languages AND Human-Centric Computing, p. 189–192, Sept. 2008. ISSN: 1943-6106.

[52] THIOLLENT, M.. **Metodologia da pesquisa–ação (7ª edição)**. São Paulo-SP, 1996.

[53] WATSON, R. B.. **Improving software API usability through text analysis: A case study**. In: 2009 IEEE International Professional Communication Conference, p. 1–7, July 2009. ISSN: 2158-1002.

[54] WATSON, R.. **Applying the Cognitive Dimensions of API Usability to Improve API Documentation Planning**. In: PROCEEDINGS OF THE 32ND ACM International Conference ON The Design OF Communication CD-ROM, SIGDOC '14, p. 1–2, Colorado Springs, CO, USA, Sept. 2014. Association for Computing Machinery.

[55] WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B. ; WESSLÉN, A.. **Experimentation in Software Engineering**. Springer-Verlag, Berlin Heidelberg, 2012.

[56] YESSENOV, K.; KURAJ, I. ; SOLAR-LEZAMA, A.. **DemoMatch: API discovery from demonstrations**. ACM SIGPLAN Notices, 52(6):64–78, June 2017.

[57] ZHANG, J.; JIANG, H.; REN, Z.; ZHANG, T. ; HUANG, Z.. **Enriching API Documentation with Code Samples and Usage Scenarios from Crowd Knowledge**. IEEE Transactions on Software Engineering, p. 1–1, 2019. Conference Name: IEEE Transactions on Software Engineering.