Journal of Software Engineering Research and Development, 2024, 12:18 doi: 10.5753/jserd.2024.4130

Colloquy: Evidence-Based Method for Supporting the Design of Conversational APIs

João Antonio D. M. Bastos [**Pontifical Catholic University of Rio de Janeiro** | ***jbastos@inf.puc-rio.br***]
Rafael Maiani de Mello [**Federal University of Rio de Janeiro** | ***rafaelmello@ic.ufrj.br***]
Alessandro Fabricio Garcia [**Pontifical Catholic University of Rio de Janeiro** | ***afgarcia@inf.puc-rio.br***]

Abstract Application Programming Interfaces (APIs) are everyday tools for every software professional. When creating an API, a designer typically abstracts the tasks the API intends to perform. The API user, on the other hand, should use the API to perform specific-propose tasks. During the development of an API, the designer needs to write dialogues with which the user will interact with the API, performing a conversation between them. In this way, an API capable of offering effective dialogues to its users is called a conversational API. A *conversational API* is the one that is capable of communicating to its users its form of use and its internal logic of operation, making it clear the design decisions when abstracting concepts and tasks. In this paper, we present Colloquy, an evidence-based method to support the design of conversational APIs. Colloquy was conceived based on the lessons learned from an action research we conducted to identify effective strategies for designing conversational APIs. For six months, we followed the redesign of an existing API developed at an R&D laboratory of a large IT company. In this paper, we also present the first empirical evaluation of Colloquy. We conducted a case study in which the method was employed to design an API for refactoring source code in the Java programming language. Among other benefits, we found that Colloquy was effective in developing empathy with users and modeling conversations with them. Besides, Colloquy contributed to identifying new requirements and created models as API documentation.

Keywords: *API, Conversation, Action Research, Case Study, Semiotic Engineering*

1 Introduction

Application Programming Interfaces (APIs) are everyday tools in software companies. APIs are software components designed to provide specialized resources to support 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 in compliance with the users' needs is far from trivial [Murphy et al., 2018]. In this way, one can see that the difficulties of API users in properly understanding API interfaces are a recurrent claim [Nielebock et al., 2020; Zhang et al., 2019]. These difficulties indicate the need to improve the design of APIs since the early stages of development to ensure communicability and usability [Henning, 2007].

Existing research on API design often focuses on supporting its designers to deliver usability [Myers and Stylos, 2016; Stylos and Clarke, 2007; Stylos and Myers, 2008]. However, these works commonly lack in addressing communicability [De Souza, 2005]. Providing communicability is essential to ensure that API users properly understand the API interfaces. Usability focuses almost exclusively on the user's aspects, such as the perceived ease to use an interactive artifact [Nielsen, 1994]. On the other hand, we may define communicability as the artifacts' ability to communicate the designers' intent to their users effectively and efficiently. Consequently, communicability enhances users' understanding, guiding them in properly using the interactive artifact [De Souza, 2005]. Thus, one may see that communicability addresses different, sometimes complementary, aspects addressed by usability approaches. However, according to Semiotic Engineering, communicability goes beyond usability by preventing potential pragmatic conflicts of interest between artifact designers and their users [De Souza, 2005]. In this way, if the users of a certain API are prone to not reaching the same understanding of a particular operation as the API designers, we may say that this API suffers from low communicability once the API is failing on passing its design rationale to the users.

Several bug reports that address many popular APIs, such as Java Reflection [Pontes et al., 2019] and automated refactoring APIs [Oliveira et al., 2019] reveal low communicability issues. 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 the report of questions caused by misinterpretations of the operation and behavior of the Calendar API. These questions are commonly associated with bugs caused by API misuse, such as in the following link[1]. In this question, a user reports an issue addressing the result of the operation of adding a month to January 31st by using the *"add"* operation. Among the several answers provided by the other users, we observed that several also misinterpreted the logic of the operation. For instance, one of the answers to this issue argues that adding

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

a month through the Calendar API is the same as adding 30 days, which would explain the unexpected result.

One may see that the example reported above does not address difficulties in using the API interface. It addresses a lack of effective communication between the API designer and the API users on how to properly use the "add" operation to support development needs. In other words, it may be easy for the API users to call the add operation and to identify which data types should be passed as parameters. However, the questions reported at Stack Overflow show that several users are unaware of the operation's internal behavior, making incorrect assumptions. Therefore, in cases like that, the easiness of use may be tricky, once API users are convinced that they know how to correctly use the API interface without resorting to the API documentation. Consequently, they tend to believe that the undesired behavior of the API operation is caused by non-existing *bugs* that would lead the API to behave differently from expected.

In such cases, we may say that the communicability between API designers and API users is insufficient. To solve this problem, we understand that APIs should be redesigned to reach a conversational level. The lack of conversational characteristics in APIs hampers the understanding of how properly use the API interfaces. In this way, Bastos et al. [2017] highlights the lack of support for designing Conversational APIs. Conversational APIs are APIs capable of indicating their appropriate use by solving pragmatic conflicts during their use, i.e., without resorting to external documentation or asking other developers Bastos et al. [2017]. The concept of conversational API is anchored in Semiotic Engineering [De Souza, 2005], human-computer interaction theory. Semiotic Engineering establishes that a computational artifact, such as an API, may 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 [De Souza, 2005].

Although the potential benefits of establishing a proper conversation between API designers and users [Souza et al., 2016], we could not find in the technical literature approaches for supporting API designers in identifying and parameterizing the necessary conversations between APIs and users. In particular, methods focused on usability [Mosqueira-Rey et al., 2018; Mindermann, 2016; Myers and Stylos, 2016; Watson, 2014] do not cover communicability issues. These methods focus exclusively on user interests, leaving out the interpretation of the internal logic implemented by the API designers.

Aiming to build an evidence-based method to support the design of conversational APIs [Bastos et al., 2020a,b], we identified the opportunity of conducting our investigation in collaboration with practitioners. In this paper, we report the challenges and lessons learned from a technical action research [Thiollent, 1996] conducted with a Brazilian team from IBM for redesigning a real API to make it conversational. This API was conceived to be used at a large scale in industrial settings, employing machine learning algorithms to support seismic imaging. The redesign of the API was required due to the considerable challenges perceived in its use. The main author of the paper worked in loco for six months with the R &D team that originally conceived the API. During the action research, we proposed and assessed interventions (actions) for redesigning the API based on theories of human-computer interaction in general and Semiotic Engineering.

This paper extends the article originally presented at SBES 2023 [Bastos et al., 2023] through presenting Colloquy, the method we designed to support the modeling of Conversational APIs based on the lessons learned in the action research. We also report the case study conducted to evaluate the feasibility of Colloquy.

In the following sections, we summarize our theoretical background (Section 2); describe the action research performed and its main challenges (Section 3); report the results of the action research and the lessons learned (Section 4) discuss the contributions of the new API from the perspective of the developers involved in the action research (Section 5); present and discuss some related works (Section 10); discuss the limitations and threats to validity (Section 3.5) and present our conclusions and perspectives of future work (Section 11).

2 Theoretical background

Our research aims at supporting API designers in conceiving conversational APIs. From a practical perspective, we expect that this support would lead to the development of APIs capable of communicating their internal behavior to users, independent of their previous background. Thus, we should go beyond traditional usability concerns, which are typically insufficient to effectively communicate the API's internal behavior to its users. For this purpose, we opted to ground our research in the Theory of Semiotic Engineering. This theory understands the HCI process as a particular case of metacommunication between humans mediated by computers [De Souza, 2005]. In this view, metacommunication is the communication between designers and users through a software artifact (see Fig. 1). Thus, the metacommunication process has three interlocutors: the *designers*, who encode their intentions into software; the *users*, who express their intent and interpretations by interacting with the software; and the *technology itself*, which represents the designer at interaction time.

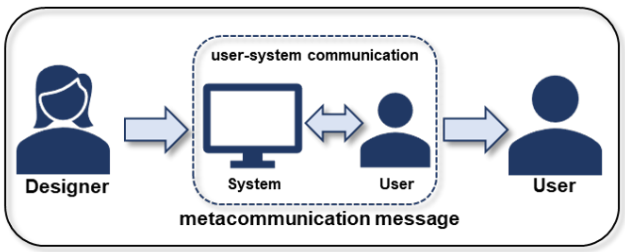

Figure 1. Semiotic Engineering and metacommunication.

For Semiotic Engineering, a software system is a tool used to exchange messages between two groups of individuals: users and designers. One of the main principles of Semiotic Engineering is *communicability*. Unlike *usability*, whose definition is strongly centered on the users, communicability is defined as the ability of an interactive artifact to effectively

communicate its designers' intent to its users. Therefore, it is important to have in mind both designers' and users' perspectives when setting and evaluating the quality of the communication provided by the API interfaces.

Based on the 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 descriptions (documentation). API users express their interpretation of the designers' intent while using APIs in their software systems. If this communication fails, issues related to the incorrect use of APIs may arise, resulting in bug reporting [Nielebock et al., 2020].

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 [Bastos et al., 2017]. API designers should predict the different types of communicability failures that may occur, preparing the API to bring the user back to effective interaction. For this purpose, the API interface should carry out its internal operation logic to the API users. These users should be able to identify all the API functionalities and understand how properly use them, even without resorting to the API documentation. Besides, the API users should be able to adapt API internal behavior by selecting different pre-programmed design choices.

From a pragmatic perspective, conversational APIs should attend to the four principles (maxims) of cooperation proposed by Grice [Grice, 1975]: quality, quantity, relevance, and mode. The *maximum of quality* requires that the interlocutors provide only correct information. The *maximum of quantity* refers to the interlocutors presenting all the necessary information objectively, avoiding unnecessary information. The *maximum of relevance* requires that the interlocutors keep focused on providing only relevant information for the conversation. Finally, *the maximum of mode* requires that the interlocutors avoid ambiguous expressions when transmitting the information. Despite the cooperation principles addressing common concerns of software verification activities, it remains an open question how to address them in the design of APIs.

2.1 Conceptual Framework for Conversational APIs

Based on these aforementioned concepts, we proposed a conceptual framework for designing Conversational APIs [Bastos et al., 2020b], reporting a detailed example of applying the framework concepts to compose a conversational version of APIs from the Java programming language. Our conceptual framework defines and classifies the conversation levels that an API can achieve. For that, we employ the concept of *sign* defined by semiotics and its separation into three classes as Semiotic Engineering defends it [Bastos et al., 2020b].

A *static sign* is one that the user can infer meaning without needing an interaction. In the case of software APIs, we can consider as 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 that the user does not need to interact with in order to infer any meaning. When faced with the method signature below, a user experienced in Java would be able to infer that this function serves to add the number of months passed by the parameter *"monthsToAdd"* to the date object used as a reference in the function call.

```
```

Listing 1: Java 8 DateTime API

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 on the correct path. Alternatively, the dynamic signs can go against the meaning inferred by the static sign, opening the user's way to create a new understanding of how to use the API. In the example listed above, we can imagine the API user creating the following code snippet.

```
/* Jan-01-2020 */

/* Add 1 month */

/* Print Jan-01-2020 */
```

Listing 2: 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 */

/* Add 1 month */

/* Print Feb-01-2020 */
```

Listing 3: Java 8 DateTime API

In the previous example, the user's interpretation is more easily inferred from analyzing the third class of signs, the *metalinguistic*. Metalinguistic signs are the ones that speak about other signs. That is, through them, the user can explicitly communicate 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, bug report tools, and official API forums.

Based on the efficient or non-efficient presence of the different sign types in the APIs, we established a level-classification of conversational APIs [Bastos et al., 2020b]. The first (and more basic) level is the *Rudimentary Conversational APIs* in which every interaction can be seen as an exchange of messages between designer and user. 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 the inefficiency of the signs used in communication. 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 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 a date (or vice versa). The absence of such functionality (or dialogue) impacts the quality of use and interrupts the conversation flow between the user and the API.

Metalinguistic Conversational APIs is the level at which the interfaces and behaviors of the API are not enough to pass the designer's entire metacommunication message, and 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. 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.

Finally, 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. It is the conversation level 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. The 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 the designer must add some signs in the interaction (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, adding too much information in static and dynamic signs may lead the API to become very verbose and less efficient during execution. Improving the API 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 is for when users build their code, and another is 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 [Souders, 2008].

3 The Action Research

In this section, we present the settings of the action research conducted. 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.

3.1 Research Objectives

We opted to conduct technical action research due to our intention of working together with API designers to identify feasible alternatives for supporting the design of conversational APIs. In this way, we intend to collaborate with a development team on redesigning an 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 API designers in conceiving conversational APIs. For this purpose, we want to identify effective strategies for improving API interfaces once they are responsible for establishing the conversations between designers and users. In particular, the first goal of the action research reported in this paper is to *redesign the interface of a real API, making it conversational*. In this way, we intend to cover the development cycle from the early steps until the specification of the API interfaces. Thus, the scope of the action research does not cover the API coding activities. We also do not intend to review the architectural aspects of the original project. Our second goal is to use the experience obtained through the action research *to build the first version of an approach for supporting the design of conversational APIs*.

3.2 Research Context

We conducted the action research in the context of an industrial project for redesigning a Machine Learning API for supporting systems for seismic image investigation. We chose 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 Brazilian team from the research and development (R&D) department of IBM. This department develops software solutions for supporting research and practice on using natural resources. More specifically, the department has been investing effort in developing software solutions supported by machine learning, such as web services and APIs, to assist decision-making in domains such as agriculture and geology. For this purpose, the department's professionals have an expressive theoretical and practical background in developing software solutions

based on machine learning. Besides, most of them also have an extensive research background in Computer Science.

The project team 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 considerable background in using the technologies used in the project, which includes the programming language (Python) and deep learning. The first author carried out the action research with the development team, but he was also an original member of the R&D department. Like the other team members, the first author has also experience designing APIs.

3.3 Execution

Action research usually requires long-term execution. In general, an action research study involves the execution of several interaction cycles over weeks or months. The central idea is to use the lessons learned in the previous cycle to plan the subsequent actions to be performed in the following cycle, evaluating its results. Each action research cycle is commonly composed of five steps. They are (1) diagnosis (2) planning; (3) action; (4) evaluation of the intervention performed; and (5) learning [Thiollent, 1996].

Three complete action research cycles. The first author of this paper carried out the action research with the development team for six months. During this period, the team performed three complete action research cycles. The first author mainly worked on guiding colleagues in performing the action research cycles' tasks and discussing the API requirements and design through regular meetings. Besides, the first author did not work composing software artifacts, such as source code and requirements specifications. At the end of the third cycle, the action research team concluded they had reached the appropriate resources and settings to redesign the API. 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 learned on designing the conversational API. In the following subsection, we report the research procedures carried out throughout each cycle.

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 [Thiollent, 1996]. As already mentioned, our action research involved three complete cycles. Figure 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 is presented 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 of each cycle were performed from the perspective of the individual that played the role of the researcher, i.e., the main author of this paper.

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 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 by concluding that characterizing the API users through personas [Cooper, 2004] combined with the description of interaction scenarios [Carroll, 2000] 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. [Rosson and Carroll, 2002].

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 discussions conducted to characterize the API personas and their corresponding interaction scenarios. After concluding the characterization tasks, the team participated in a meeting to interactively evaluate the correctness, comprehensiveness, and 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 characterizing personas and interaction scenarios was a very positive experience. They found that these techniques allowed them to identify new user profiles and alternative ways for using the API that were not identified in the API´s original design.

Finally, we conducted a meeting to compile the lessons learned in the first cycle of the action research. In this meeting, we reflected on to which extent the action performed helped us reach the action research goals, as well as how to improve the action. 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 in performing these activities (subsection 4.1).

3.4.2 Second Cycle:

We started the second cycle by assuming that the whole set of the API's user profiles and their corresponding use needs were properly characterized. However, we diagnosed that the characterization of personas and scenarios is too informal and subjective for supporting future API implementation. Thus, we concluded that we should model the conversations and the interaction paths corresponding to the scenarios established for each persona. Therefore, we started our second

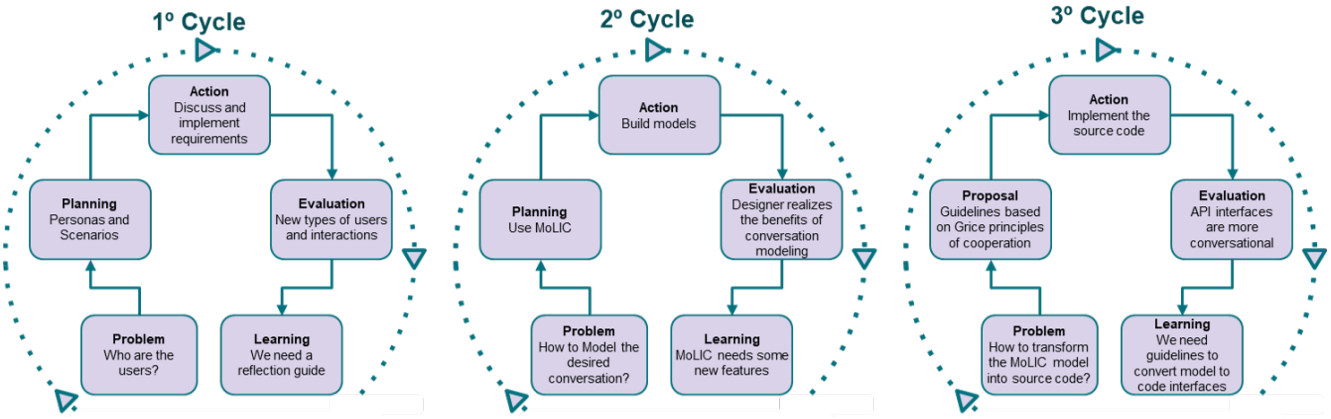

Figure 2. The Action Research Cycles

cycle motivated by the following question (Figure 2): *How to model the desired conversations?*

The second cycle lasted around ten weeks. We searched the technical literature to identify 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 (Modeling Language for Interaction as Conversation) [de Paula and Barbosa, 2003], UML Interaction Diagram [OMG, 2016] and AlaDIM [Costa Neto, 2013]. While UML Interaction Diagram and AlaDIM are specific to software interface interaction design, MoLIC is more agnostic to the type of artifact designed. In addition, we can highlight that MoLIC has the characteristic of modeling interaction as a metaphor for the conversation established between designers and users [de Paula and Barbosa, 2003]. Besides, one can see that MoLIC has been frequently used in software projects to design human-computer interaction [Sangiorgi and Barbosa, 2009]. Thus, we opted by using MoLIC to redesign the API.

After selecting MoLIC, the team started to act, 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 started by discussing the main challenges faced by 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 exchanging restricted messages through parameters and returns of operations. For this, MoLIC could not meet the 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 learned that some adaptations should be made to MoLIC better serve the project purpose. These adaptations are described in subsection 4.2.

3.4.3 Third Cycle:

After designing the whole set of 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 implementing the API operations signatures, return values, and function parameters. The third and last cycle of the action research lasted around ten weeks.

To overcome this challenge, we provided the design team with a set of best practices and guidelines for building and structuring the API source code, including the Grice's principles of cooperation [Grice, 1975]. Those guidelines and practices focused on realizing the conversations designed in the previous modeling cycle. However, we could not find guidelines to support the API implementation based on MoLIC models available in the technical literature. Thus, the development team promoted discussions for depicting their own guidelines for converting each element of the MoLIC models to the API interface elements. Our goal here was to emerge mapping rules at such a level that the interface elements could be systematically extracted from the content of the MoLIC models. These guidelines are described in subsection 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 main author of this paper. At the end of the third cycle, 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 feedback from the technical leader regarding all the action research cycles. The findings of this interview are presented and discussed in Section 5.

3.5 Limitations and Threats to Validity

One limitation of our study addresses 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 to first reach a solid knowledge of the research object for then propose a mature technology promptly to 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 that emerging feasible innovation from real settings is a key benefit of action research studies. Besides, we understand that the unique characteristics of this research paradigm are key for bridging gaps between research and practice.

Given the nature of the action research, the researcher has actively contributed to the API discussion. The study involved the participation of a Ph.D. and two Ph.D. candidates (Section 3.2) working on redesigning an API from a highly complex domain. This context probably contributed to the perceived engagement in the action research. During each cycle, the author could perceive a receptive environment for research. In this way, the first author did not identify any negative feedback about the action research. However, he identified two more relevant and continuous challenges to overcome. The first challenge was to keep the team focused on the API redesign rather than running other projects. The second challenge was dealing with the complexity of the API domain, including the eventual need to change the API scope.

Despite the development team members had experience with API design, they did not have previous experience in designing the interaction of the API conversations. Thus, the researcher had to frequently lead the design of the API dialogues and the conversation flows. However, all the team members were engaged in validating all the design decisions taken and artifacts produced during the API development cycles.

4 Lessons learned

During the execution of the action research, We learned a set of valuable lessons for designing conversational APIs. We present and discuss these lessons in this section. 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.

4.1 Who are the Users?

4.1.1 Challenge: To make designers aware of the users' needs

API designers should be aware of the potential API users to properly identify its functionalities and interfaces. Thus, this knowledge is required to introduce conversationality. However, we need to reach a deeper level of knowledge regarding the API users' profiles to properly identify 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 specifying interfaces using more specific terms addressing deep learning, such as "backpropagation" or "convex optimization." However, the API designer should abstract these concepts from professionals, such as medical doctors and geologists, who would use the API in their programs.

4.1.2 Solution: To adopt consolidated HCI techniques

During the action research, the first challenge faced by the team was how to properly identify the profiles of users that 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 [Cooper, 2004] in the context of software design interaction. Personas are fictional characters used to characterize and combine the different roles played by software users. Characterizing a persona requires characterizing its domain knowledge, its motivation for adopting the developed technology, and its technological background. Besides, it also requires 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 [Rosson and Carroll, 2002] aims to specify the user's actions and the corresponding system responses in detail. In our understanding, the system is the API under development. Interaction scenarios are composed of textual and concrete narratives, rich in contextual details. This narrative reports API usage situations by characterizing the corresponding users, processes, and (potentially) real data. We use interaction scenarios to describe all the possibilities of interaction that a persona may have with the API.

4.1.3 Lesson Learned: API designers have difficulties on establishing personas and scenarios

During the personas' identification and creation phases, the API designers reported difficulties identifying and characterizing the API users. Truthfully, personas and interaction scenarios are not methods for requirements gathering, but requirements registering. We learned that API designers need additional support to identify and characterize personas. In this sense, we had to investigate relevant aspects to consider when describing the personas for an API.

API metacommunication template: The Semiotic Engineering [De Souza, 2005] offers the meta-communication template, a technique that can help in the identification of users, [De Souza, 2005]. Based on this template, Afonso [Afonso, 2015a] proposes the *API metacommunication template*. This template is composed of generic questions for characterizing users in the context of any software. Table 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 instance, certain aspects such as academic background may be irrelevant for developing general-purpose APIs – such as date manipulation APIs, or database access APIs –. However, such information becomes important in developing APIs to support scientific research. Thus, API designers should work to identify which aspects are relevant in its API domain.

Through the action research, we learned that personas were crucial to create empathy with users and to start developing their API thinking about another person's existence

Table 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

on the other side. By using the adapted metacommunication template, the API designers were able to define two different personas (2 and 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 was fundamental to supporting the creation of the conversations. When modeling the API interfaces, the designer should take into account the characteristics of each persona. This empathy will generate new and adequate functionalities for the API, as presented in the following subsection.

4.2 How to Model the API Conversations

4.2.1 Challenge: To help designers to model API conversations

The API designer should model the possible conversations between the API and its users. However, we observed that API designers were not used to building models during the API design process. On the other hand, it allowed us to propose and assess the use of MoLIC (Modeling Language for Interaction as Conversation) [de Paula and Barbosa, 2003], a traditional modeling language from the HCI field focused on the creation of interaction models.

4.2.2 Solution: Use MoLIC to think about API dialogues

MoLIC is a popular resource for HCI modeling, especially

Table 2. Persona 1 - Machine Learning Expert

Who are the users?	They are professional programmers with an academic background in artificial intelligence and experience in the Python programming language. They know the terms and concepts of deep learning. They are used to work with programming patterns in Python and object-oriented programming. 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 of geological data. They want to build the neural networks of their DL model. They don't have sufficient 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 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?	As professional programmers, they focus on creating 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 in abstracting such input data, allowing them to focus on creating predictive models.

Table 3. Persona 2 - Geologist

Who are the users?	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 able to abstract deep learning concepts and provide sets of ready-made models.
What are their API preferences?	They prefer simplified APIs that are easy to use, and have few configuration options. They also prefer APIs that follow the basic scripting style with direct and simplified function calls. Although they are not used to 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 deep learning, such as how to build neural networks and how to convert data into tensors. Therefore, they need the API ready to abstract these concepts, providing methods for data conversion and ready-made models.

in 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 with interaction scenes. 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 a set of speeches exchanged between the designer and the user. *Conversation flows* are represented by directed lines. These lines may be continuous to indicate a progressive flow in the conversation or dotted to indicate a regressive flow in the conversation. Regressive flows usually happen due to communication failure.

4.2.3 Lesson Learned: To adapt MoLIC

Although we observed that MoLIC is potentially useful for modeling API conversations, 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. In future work, we will conduct new investigations to establish a comprehensive modeling language to support 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. An asynchronous flow may be needed, especially when the API requires large-scale 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, training models and predictions may take a long time, leading to a temporary suspension of the conversation. In this case, the conversation will be only resumed after an API signal sending, which should be properly implemented in the source code. *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 fixed. The API interfaces are defined with parameters, data types, and returns. Thus, another necessary adaptation in MoLIC was providing support for specifying the dialogues taking place in the modeled scenes. To this end, we used OpenAPI, a modeling language designed for specifying RESTFul services [Initiative, 2020]. 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 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 a set of reserved words and structure rules, API designers can specify API interfaces, input data, output data, and documentation.

Figure 3 shows a snippet of a model designed through our adapted version of MoLIC. The model was built in the context of the API designed during the action research. This figure shows a conversation between the user and the API about the user's goal of training a particular model. The user can start right at the training scene if he already knows which model he wants to train. Alternatively, the user can pass before a scene to discover the available models. During the model training scene, the conversation between the user and API is suspended by an asynchronous flow, returning just 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.

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 their own models, may opt by going through the scene to discover the available models and identify which model would better fit the user's needs. On the other hand, for the persona expert in deep learning, going through this scene is optional, as he may be training his own model. Thus, 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".

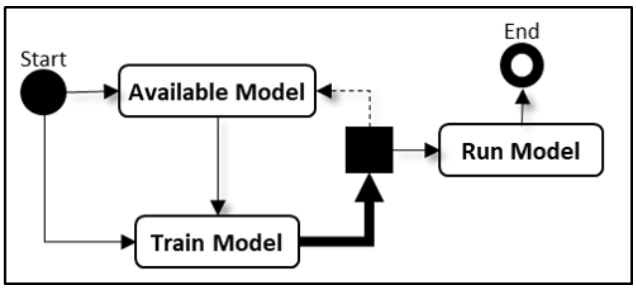

Figure 3. MoLIC and Adaptations - DL API Modelling

With the API model, API designers were able to think about new implementation needs and improvements. 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. Through creating a method for this purpose, the API conversation was improved and the API designers experienced 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.

4.3 How to Implement the API Interfaces

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 pro-

gramming languages is limited to textual signs. These signs include function signatures, documentation, and 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 in creating API interfaces based on MoLIC diagrams, which include identifying the appropriate signs.

4.3.2 Solution: To compose a set of guidelines for 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 Grice's principles of cooperation [Grice, 1975]. In the 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 make clear which conversations are not allowed. That is, the API designer may choose to abstract some concepts as well as prioritize others. This guideline addresses 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 deep learning concepts. Alternatively, concepts from the 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 been discussed [Bastos et al., 2017], 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 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 to 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 the users' understanding of API use. In particular, we recommend avoiding metaphors and idiomatic expressions. For instance, during the action research, we found that a metaphoric function name- "ZooModel"- was confusing for non-deep learning 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 among user interactions, assuring efficient conversation flow. Based on the *maximum quality* principle, API designers should assure consistency between the API documentation, interfaces, and internal behavior. Consequently, all the possible conversations will be effective, avoiding tricking the API user with wrong understandings about the API internal behavior.

4.3.3 Lesson Learnt: We need more than just guidelines

We realized that we would need more than guidelines to support the designer in choosing the appropriate signs and establishing an effective conversation. We should go beyond by conceiving an evidence-based method to support the construction of conversational APIs. Further, this method should be supported by a recommendation tool that is able to combine semiotic engineering concepts to establish different conversation levels according to the signs used by the API.

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 the researchers' participation in long-term projects. Opportunities for conducting action research in industrial settings are still uncommon in the field despite potential benefits for both research and practice. Besides, recording and reporting data gathered in long-term studies is quite complex. The work presented in this paper reports an action research that required the collaboration of a researcher in an industrial project for six months. Through this study, we investigated and improved 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 restricted context, we believe the lessons learned through the action research and the proposed solutions for designing conversational APIs address relevant contributions to the field. Besides, the study also led us to apply the concept of conversational APIs for the first time in practice. Another contribution of the action research is developing the conversational API required by the company. The action research cycles 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 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, which evolved to the guidelines presented in Section 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 developers." 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 the guidelines as important to support API designers in reflecting on the different levels of conversation that an API may offer according to the user experience. For him, these guidelines are 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 exemplified with another API: "The Python {anonymous} [2]API, for example, is fantastic. However, the use of the API is awful. The designer did fantastic work, but 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 access them may hinder or help a lot its use."

Despite the positive feedback, it is important to also discuss the possible limitations of the 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 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.

6 Colloquy

Based on the experience gathered with the action research, we designed the *Colloquy*, a method to establish conversation in APIs. Colloquy 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. The main characteristics of Colloquy were previously reported in [Bastos et al., 2020a].

Figure 4 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 identifying and characterizing 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. In the supplementary material, we illustrate 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 his API development process, offering techniques and tools for exploring and creating user-API conversations. Thus, Colloquy does not predict the participation of actual users in the development process. However, Colloquy does not prohibit such participation, and it is up to the designer to adopt or not users for evaluating the artifacts created.

Thus, we consider that a designer who has access to include actual users in the API 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 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 whether the modeling 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.

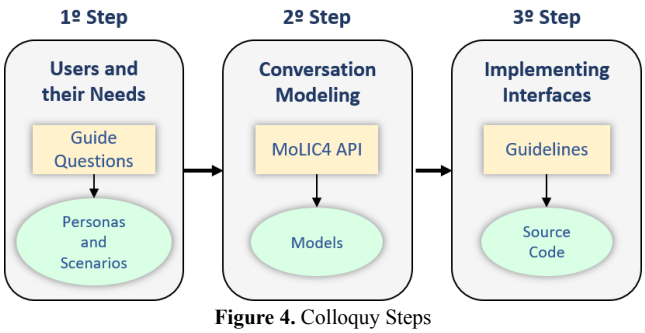

Figure 4. Colloquy Steps

6.1 Personas and Interaction Scenarios

The first step of Colloquy is aimed at discovering the potential API users and their corresponding requirements. This

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

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 purpose, Colloquy recommends adopting two technologies often adopted by human-computer interaction professionals: *personas and interaction scenarios*.

As already introduced, personas are fictional characters used to characterize and combine the different roles played by the software user [Cooper, 2004]. The definition of these characters will give the API 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. Interaction scenarios are composed of a textual and concrete narrative, rich in contextual details. This narrative reports the API usage status, describing the processes involved and (potentially) its data. Thus, we use interaction scenarios at this stage to describe all possibilities of interaction that users may have with the API.

6.1.1 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 for. 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 4) to help the designer make this reflection. These questions are inspired by the API metacommunication model proposed by Afonso [Afonso, 2015b].

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 conversation about the time zone and the date input and display format.

Table 4. 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

6.2 Conversation Modeling

Once the personas and the corresponding interaction scenarios are defined, the API designer needs to model the different conversations users 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 [de Paula and Barbosa, 2003] and OpenAPI [Initiative, 2020].

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 the programming language. The designer should use MoLIC4API to model all interaction scenarios 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 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 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 rect-

angular 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).

6.3 Interfaces Implementation

The primary goal of Colloquy is to help designers model 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 recommendation in subsection 6.3.1. Besides, the method also offers a set of guidelines to support the designer when naming and structuring API interfaces. These guidelines are grounded in Grice's principles of cooperation [Grice, 1975]. In Subsection 6.3.2, we list the guidelines with their corresponding principles.

6.3.1 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 operation call 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 be represented as an error message. In this way, the API designer should write these messages carefully. An error message must be comprehensive enough for the user to recover from the error and shift to a productive interaction. If the flow is progressive, the operation return may report a message indicating the operation's success. 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.

6.3.2 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 make clear which conversations are not allowed. That is, the API designer can choose to abstract some concepts as well as prioritize others. This guideline addresses Grice's principle of maximum relevance.

G02. The API documentation must be simple and concise: API documentation is a key opportunity for API designers to improve 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 despite 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 [Bastos et al., 2017; Lamothe and Shang, 2020]. Based on the principle of maximum quantity, we recommend that API designers should avoid establishing 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 to attend to 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 to avoid metaphors and idiomatic expressions.

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, ensuring efficient conversation flows. Based on the maximum quality principle, API designers should ensure consistency between the API documentation, interfaces, and 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.

7 Case Study Plan

The case study presented in this chapter aims to test the advantages provided by Colloquy when developing a real API. To do this, we established a context in which we could get an 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. The API was designed by a Doctoral researcher investigating refactoring and code smells.

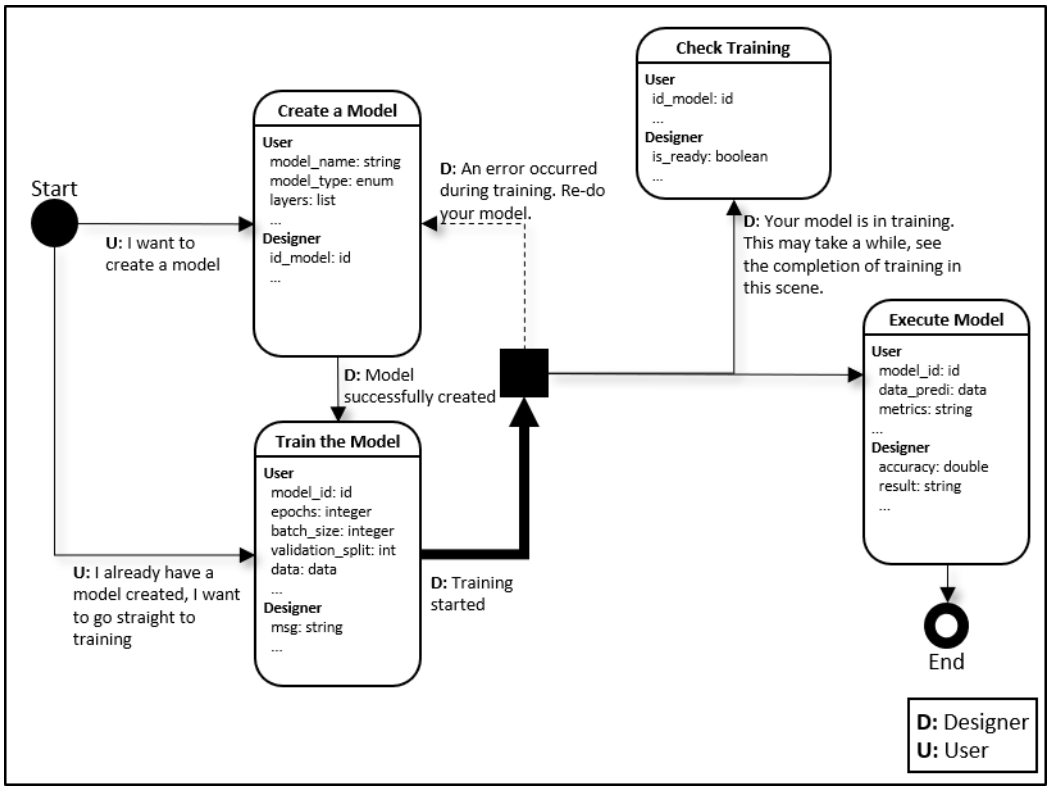

Figure 5. Example of MoLIC4API modeling

7.1 Goal and Research Questions

By relying on the guidelines provided by Wohlin et al. [Wohlin et al., 2012], 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 as follows.

RQ1. By using Colloquy, would an API designer 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 it in the results (Section 8).

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 the ease or difficulty of executing the Colloquy, its effects on API design, and the need for additional support.

7.2 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 of how to operate. Thus, an API dealing with source code refactoring can significantly benefit from the conversational API and the method that we propose for the design process. This study was conducted in the context of a project to build an API to refactor source code written in the Java programming language. The API was designed by an experienced Java programmer who has developed software in academic and industrial environments. The API designer, whom we will refer to as a participant, has extensive experience developing and designing reusable APIs and software. Besides, the participant has extensive experience developing refactoring and code smell 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 [Fowler, 2004]. Still, we find cases where users disagree about the operations performed by APIs [Oliveira et al., 2019]. Even simple refactoring as an Extract Method [Fowler, 2004] can find several ways to be performed among API users [Oliveira et al., 2019]. Thus, we believe that a conversational API may be the appropriate approach to decrease misuse recurrence for the refactoring context.

7.3 Data Sources

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

- **Interviews with the participant:** We conducted four 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 was created, including personas, scenarios, models, and API interfaces. These data were collected so that they could later be analyzed and discussed with the participants. We required the participant about each artifact to discover how Colloquy might have helped him in the creation.

7.4 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 interview data with the artifacts generated to understand if the participant was properly using the method. All the interviews were transcribed to allow 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 is the method, and also opening ways for future improvements. The results of this in-depth analysis of the participant's statements are presented in section 9.

7.5 Phases of the Study Execution

7.5.1 Phase 1 - API Design Following Another Method

In the first stage of the case study, we searched the technical literature for methods and guidelines to support the API design process. Among the papers found, we selected the approach proposed by Henning [Henning, 2007] because it was one of the few that offered a sequence of steps and guidelines for API design. Moreover, this work was the one with the 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. We present 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, and 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 Henning's method with the API designer. In this training, we guided him to use his skills and follow Henning's work to create 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 that the participant was not exclusively working on our study. Then, we conducted a second meeting with the API designer to collect 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.

7.5.2 Phase 2 - API Design Following Colloquy

At the last hour of the second meeting, we introduced the participant to the conversational API and Colloquy concepts. In this meeting, the participant was trained to perform the first phase of our method. More specifically, we introduced the participant to the personas and scenarios concepts, 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, we guided the participant 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 his experience regarding the Colloquy use. The same happened for the following 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, followed by another meeting to collect the participant's opinions and experiences. In Phase 2, the participant took four weeks to perform the modeling, and in Phase 3, it took 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 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 the user and API. In section 7.6 we discuss how the execution of Colloquy after a previous API design may have impacted the study validity, and how we face this threat.

7.6 Threats to Validity

We planned to first perform the API design by following an existing approach [Henning, 2007] and then redesign the API with our approach. Our intention was to identify which improvements our approach could bring to the design process. However, we are aware that this methodology has a learning bias as the designer has already acquired previous knowledge by using another 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 scratch. Moreover, we plan to conduct new studies following alternative designs. Besides, we intend to evaluate our method without a comparison with existing methods, ensuring that all API designs emerge from Colloquy.

Another threat to the validity of this study addresses its restricted context: a single API designed by a single professional. We know that more studies are needed to generalize our findings and conclude that our approach would be applicable in a large-scale industrial setting. However, case studies are complex studies, requiring considerable research effort. We are reporting this as 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 to our method since the participant may already have a vision about which API he wants 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 how the method helped him think and improve the API design, highlighting what would not have been possible without the method.

8 Case Study 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/she performs the three steps proposed by Colloquy.

8.1 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.

8.1.1 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 to ensure 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*

8.1.2 Persona 2: Philip - Experienced Freelance Programmer

Philip has a degree in computer science and is an experienced freelance programmer. During his five years 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 capable of supporting him in refactoring programs in his old code bases. The participant thought of two scenarios to assist this persona:

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

8.1.3 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 improving the structural quality of her code. Therefore, Katarina needs a tool to help her overcome this challenge. As Katarina is still studying software design, she wants a tool that would work as a 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*

8.1.4 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 needs 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 learning how to program, i.e., 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.

8.2 Diagrams

After creating the personas and defining the interaction scenarios, the participant proceeded to the Colloquy's next stage. Following the method's procedures, the participant modeled the interaction defined in the previous scenarios using the MoLIC4APIs modeling language. Figure 6 presents the final model generated.

During API modeling, the participant could think in detail about the interaction flows (conversations) that the user

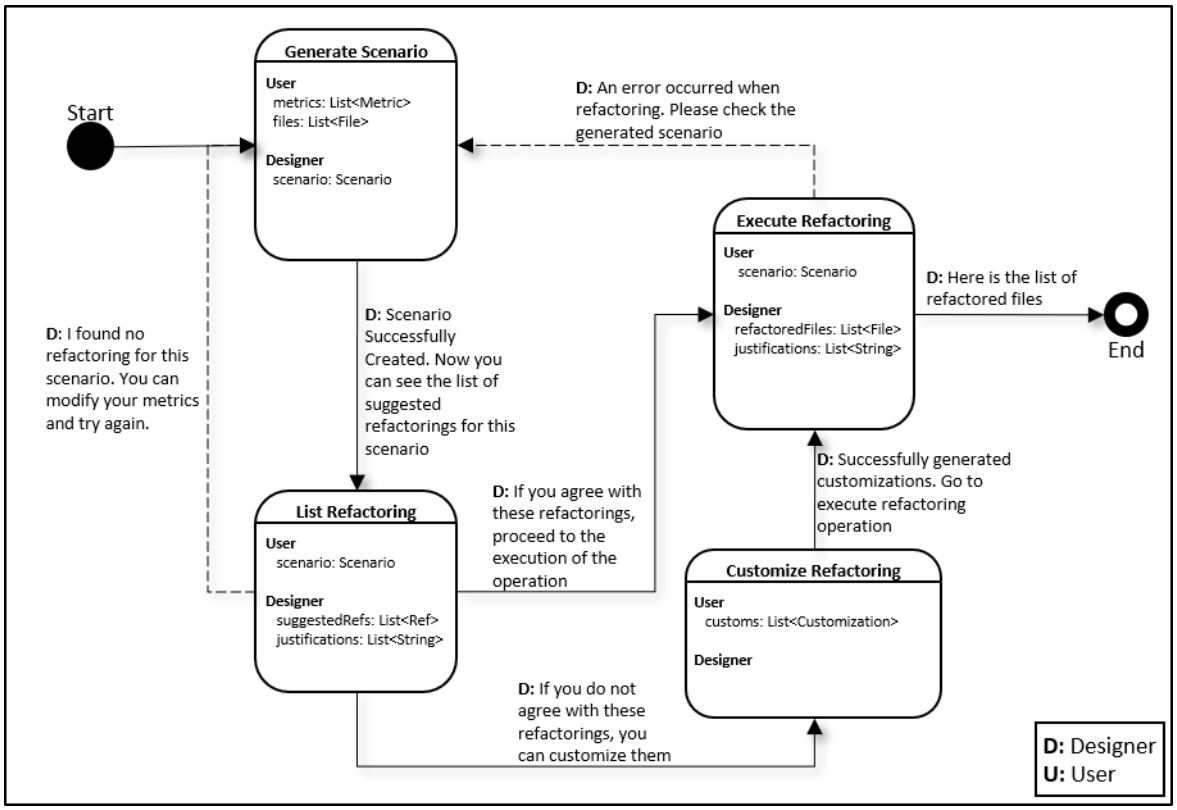

Figure 6. Modeling the Refactoring API Interaction

should perform with its API. In the participant's own words, modeling helped him think about the regressive flows, that is, possible problems the user would face and how to solve them. In Figure 6, we see the conversation scenes, conversation flows, and dialogues exchanged between designers and users, represented in the OpenAPI language (Section 6.2).

8.3 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 4 shows the source code of the interface and a short documentation above them. This documentation aims to explain what each operation does. Besides, the documentation also explains the parameters and return values. In the following subsection, we 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 resulted from 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. We present below the four public methods created in the final stage of Colloquy. In subsection 8.4, we discuss which elements represent improvements concerning the API generated using Henning's guidelines [Henning, 2007]. In subsection 8.5, 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

/** List the suggested refactorings and the

/** Add a set of refactoring customizations

/** Execute the refactoring in the scenario,
```

Listing 4: Generated API for Refactoring Java Programs Language

8.4 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 [Henning, 2007]. We highlight in this subsection the improvements that came with the method and their respective reasons.

The first improvement was the reflection made by the participant about the need to give his user a reason for each refactoring his API was suggesting. Inspired by the persona Katarina (subsection 8.1.3), 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 the 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 a 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 realized that the conversation flow has become more natural between users and designers. Moreover, splitting the original method into two methods brought a new feature the participant had not identified 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.

8.5 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 not be 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 capture 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 exposes 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 wants. Besides, it is important to note that all this conversation happens in the interface, even if a more detailed description of each operation is available in the 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 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.

9 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 participants during the interviews. Besides, we also discuss some opportunities to improve Colloquy that have emerged from participant opinions and experiences. Finally, we discuss some threats to the validity of our case study.

9.1 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 begin expanding 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 solve 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 found an advantage in modeling the API interaction as a conversation. The MoLIC4API language helped him think 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, the participant could not reflect on the API conversations and interaction flow.

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 requirements to users. Thinking about the personas made me expand on what the API could offer. It made me think about things I had not thought about before."*

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 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, when presenting my functionalities, I could 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 to increasing the quality of the API documentation.

The aforementioned benefits indicate that Colloquy is feasible, bringing effective contributions to designers developing better APIs. Such contributions perceived by the designer encompass the three steps of the method, resulting in 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 quality of use from the user's point of view.

9.2 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 8.4.

Prioritize which guidelines are most important: During the interviews, we noticed that the API designer faced some difficulties applying Colloquy. The first major difficulty for the participant was creating the API personas and scenarios. Although he reported that the guiding questions significantly helped him on thinking and writing about personas and scenarios, the participant pointed out several aspects to consider. For him, several aspects listed in the guiding questions do not make sense in his context. For example, for the participant, questions about "culture" or "demographic data" are irrelevant for characterizing the API personas. In contrast, questions such as academic background are strongly important, as he believes that academics are more prone to better understand code refactoring than practitioners. Thus, possible opportunities for improving Colloquy address 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, the participant considered the need to memorize exactly how to design each model element inefficient. In his words, he would have significantly benefited from a tool to assist him in building 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 only 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.

9.3 Colloquy and the Software Development Process

The experience of the case study reveals that the proposed method could address the challenges identified through the action research 4. We evidenced that Colloquy helps designers to be aware of the users' needs, model API conversations, and choose the appropriate signs for API Interfaces. However, the drawbacks identified in the case study point to the opportunity of simplifying the method's guidelines for improving communication with API designers. Besides, the case study revealed a complementary challenge: developing a tool for supporting the use of Colloquy.

The empirical studies reported in this paper, i.e., the action research for conceiving Colloquy and the case study for evaluating Colloquy, were searching for design solutions for the problems we identified. We chose to conduct two qualitative empirical studies with different natures and domains to explore in-depth possible solutions to support designers in creating conversational APIs. The experience of these studies showed us how urgent it is to offer techniques and tools to support API designers. In both studies, we realized that API designers do not usually follow traditional Software Engineering methods to improve the API quality. Thus, APIs are usually produced with low quality and do not meet the needs of conversation with their users, which may lead to API misuse or even disuse.

Both empirical studies were not performed in teams following specific software development methods or processes. However, we believe that Colloquy 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.

10 Related Work

The first work discussing the quality of use of APIs appeared in the 70s [Weinberg, 1971]. However, only at the end of the 90s the investigations in this topic gained notoriety. The study conducted by McLellan et al. [McLellan et al., 1998] investigated the use of HCI techniques, such as interviews, usage scenarios, and recordings to support the evaluation of APIs. The authors use their findings to discuss how code examples may influence the programmers' use of APIs. For instance, they found that example-based learning may lead programmers to rely on errored code, promoting misunderstanding of how an API works.

In the last ten years, different works have investigated the usability of APIs. Some investigations focus on approaches to improve the API design process, such as Watson [Watson, 2014], Mindermann [Mindermann, 2016], and [Mosqueira-Rey et al., 2018]. 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. proposed guidelines and heuristics to be applied along the design process of APIs to achieve usability. Although these works are concerned with API usage quality, they lack addressing communicability aspects, including pragmatic issues and promoting the understanding between users and designers.

The lack of conversational APIs leads API users to experience difficulties in effectively applying APIs in their projects. One of the main goals of our research is to support redesigning already existing APIs into conversational ones. Alternatively, technical literature presents tools for assisting developers in properly using APIs. For instance, Yessenov et al. [Yessenov et al., 2017] propose *DemoMatch*, a tool to support programmers in discovering how to use an API based on interactions between the API and software systems already using it. Ichinco et al. [Ichinco et al., 2017] propose *Example Guru*, a tool for recommending APIs based on the context of the programmer's code. Addressing code verification, Nguyen et al. [Nguyen et al., 2017] 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 works investigated the pragmatic issue of API misunderstanding [Nielebock et al., 2020]. Although they did not propose methods or techniques to improve API communication, these works may represent resources to support the understanding of communication limitations among designers and users. Nielebock et al. investigated the misunderstandings in using APIs, leading to their misuse and even the incidence of bugs. To mitigate this risk, the authors introduce a tool to identify API misuse, offering rules to fix its use. The work reported by Lamothe and Shang [Lamothe and Shang, 2020] aims to understand 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.

11 Conclusion and Future Work

In this article, we present the lessons learned from 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 to prioritize the user-

API interaction in the API design for empowering its 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 [Pontes et al., 2019] and APIs for refactoring [Oliveira et al., 2019].

Based on the lessons learned with the action research, we designed Colloquy, 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. In 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 observation 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 an evolved and toll-supported method. Thus, we aim to disseminate and evaluate its use on a large scale.

Artifacts Availability

Artifacts are avaliable here.

Acknowledgements

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

References

Afonso, L. M. (2015a). *Communicative dimensions of application programming interfaces (APIs)*. PhD thesis, Programa de Pós-Graduaçao em In-formática of the Departamento de Informática ….

Afonso, L. M. (2015b). *Communicative dimensions of application programming interfaces (APIs)*. PhD Thesis, Programa de Pós-Graduaçao em Informática of the Departamento de Informática ….

Bastos, J., Mello, R., and Garcia, A. (2023). On the support for designing a conversational software api: An action research study: An action research study. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, pages 57–66.

Bastos, J. A., Afonso, L. M., and de Souza, C. S. (2017). 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)*, pages 213–221. ISSN: 1943-6106.

Bastos, J. A. D., de Mello, R. M., and Garcia, A. (2020a). Colloquy: A method for conversational api design. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, pages 514–519.

Bastos, J. A. D., de Mello, R. M., and Garcia, A. F. (2020b). A conceptual framework for conversational apis. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, pages 509–513.

Carroll, J. M. (2000). *Making use: scenario-based design of human-computer interactions*. MIT press.

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

Costa Neto, M. A. (2013). Uma linguagem de modelagem da interação para auxiliar a comunicação designer-usuário.

de Paula, M. G. and Barbosa, S. D. J. (2003). Designing and Evaluating Interaction as Conversation: A Modeling Language Based on Semiotic Engineering. In Jorge, J. A., Jardim Nunes, N., and Falcão e Cunha, J., editors, *Interactive Systems. Design, Specification, and Verification*, Lecture Notes in Computer Science, pages 16–33, Berlin, Heidelberg. Springer.

De Souza, C. S. (2005). *The semiotic engineering of human-computer interaction*. MIT press.

Fowler, M. (2004). *Refatoração: Aperfeiçoamento e Projeto*. Bookman. Google-Books-ID: xV2_wAEACAAJ.

Grice, H. P. (1975). Logic and Conversation. In *Speech acts*, pages 41–58. Brill.

Henning, M. (2007). API Design Matters. *Queue*, 5(4):24–36.

Ichinco, M., Hnin, W. Y., and Kelleher, C. L. (2017). 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, pages 1105–1117, Denver, Colorado, USA. Association for Computing Machinery.

Initiative, O. A. (2020). OpenAPI Specification.

Lamothe, M. and Shang, W. (2020). When APIs are Intentionally Bypassed: An Exploratory Study of API Workarounds. page 13.

McLellan, S., Roesler, A., Tempest, J., and Spinuzzi, C. (1998). Building more usable APIs. *IEEE Software*, 15(3):78–86. Conference Name: IEEE Software.

Mindermann, K. (2016). 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, pages 62–63, Austin, Texas. Association for Computing Machinery.

Mosqueira-Rey, E., Alonso-Ríos, D., Moret-Bonillo, V., Fernández-Varela, I., and Álvarez Estévez, D. (2018). A systematic approach to API usability: Taxonomy-derived criteria and a case study. *Information and Software Tech-*

nology, 97:46–63.

Murphy, L., Kery, M. B., Alliyu, O., Macvean, A., and Myers, B. A. (2018). 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)*, pages 249–258. ISSN: 1943-6106.

Myers, B. A. and Stylos, J. (2016). Improving API usability. *Communications of the ACM*, 59(6):62–69. Publisher: ACM New York, NY, USA.

Nguyen, D. C., Wermke, D., Acar, Y., Backes, M., Weir, C., and Fahl, S. (2017). 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, pages 1065–1077, Dallas, Texas, USA. Association for Computing Machinery.

Nielebock, S., Heumüller, R., Krüger, J., and Ortmeier, F. (2020). Cooperative API Misuse Detection Using Correction Rules. page 4.

Nielsen, J. (1994). *Usability Engineering*. Morgan Kaufmann. Google-Books-ID: 95As2OF67f0C.

Oliveira, J., Gheyi, R., Mongiovi, M., Soares, G., Ribeiro, M., and Garcia, A. (2019). Revisiting the refactoring mechanics. *Information and Software Technology*, 110:136–138.

OMG (2016). About the Unified Modeling Language Specification Version 2.5.1.

Pontes, F., Gheyi, R., Souto, S., Garcia, A., and Ribeiro, M. (2019). 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, pages 636–646, New York, NY, USA. Association for Computing Machinery. event-place: Tallinn, Estonia.

Rosson, M. B. and Carroll, J. M. (2002). *Usability Engineering: Scenario-Based Development of Human-Computer Interaction*. Morgan Kaufmann. Google-Books-ID: sRPg0IYhYFYC.

Sangiorgi, U. B. and Barbosa, S. D. (2009). 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, pages 303–308, Pittsburgh, PA, USA. Association for Computing Machinery.

Souders, S. (2008). High-performance web sites. *Communications of the ACM*, 51(12):36–41.

Souza, C. S. d., Cerqueira, R. F. d. G., Afonso, L. M., Brandão, R. R. d. M., and Ferreira, J. S. J. (2016). *Software Developers as Users : Semiotic Investigations in Human-Centered Software Development*. Springer International Publishing.

Stylos, J. and Clarke, S. (2007). Usability Implications of Requiring Parameters in Objects' Constructors. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 529–539, USA. IEEE Computer Society.

Stylos, J. and Myers, B. A. (2008). 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, pages 105–112, Atlanta, Georgia. Association for Computing Machinery.

Thiollent, M. (1996). *Metodologia da pesquisa–ação (7ª edição)*. São Paulo-SP.

Watson, R. (2014). 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, pages 1–2, Colorado Springs, CO, USA. Association for Computing Machinery.

Weinberg, G. M. (1971). *The Psychology of Computer Programming*, volume 29. Van Nostrand Reinhold New York.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer-Verlag, Berlin Heidelberg.

Yessenov, K., Kuraj, I., and Solar-Lezama, A. (2017). DemoMatch: API discovery from demonstrations. *ACM SIGPLAN Notices*, 52(6):64–78.

Zhang, J., Jiang, H., Ren, Z., Zhang, T., and Huang, Z. (2019). Enriching API Documentation with Code Samples and Usage Scenarios from Crowd Knowledge. *IEEE Transactions on Software Engineering*, pages 1–1. Conference Name: IEEE Transactions on Software Engineering.