Source code comprehension and appropriation by novice programmers: understanding novice programmers' perception about source code reuse

Luana Müller
Escola Politécnica
PUCRS
Porto Alegre, Brazil
luana.muller@acad.pucrs.br

Milene Selbach Silveira
Escola Politécnica
PUCRS
Porto Alegre, Brazil
milene.silveira@pucrs.br

Clarisse Sieckenius de Souza
Departamento de Informática
PUC-Rio
Rio de Janeiro, Brazil
clarisse@inf.puc-rio.br

***Abstract*—Software development practices rely extensively on reusing source code written by other programmers. One of the recurring questions about such practice is how much programmers, acting as users of somebody else's code, really understand the source code that they inject it in their programs. The question is even more important for novices, who are trying to learn what programming is and how it should be practiced on a larger scale. In this paper we present the results of an ongoing research using a semiotic approach to investigate how novice programmers reuse source code, and how, through messages inscribed in the source code of the programs they write or reuse, they communicate, implicitly or explicitly, what such source code "means" to them and others. We carried out three studies with novice programmers, and results suggest that source code reuse may impact what programmers take their source code to mean.**

***Index Terms*—Source code reuse, Semiotic Engineering, novice programmers, metacommunication**

I. Introduction

While learning how to program, novice programmers need to face the difficulties of learning how to think computationally [1]. According to Keheller and Pausch [2], apart from learning how to create structured solutions to their problems and understanding how programs are executed, novice programmers also need to deal with syntax, and the meaning of programming languages commands from programming languages, and they have problems to translate their intentions to the computer.

As an alternative to help them during this learning process, novice programmers commonly use source code examples to support their activities, and several times, they opt for reusing this code [3], integrating it into their own source code and performing the necessary adjustments to achieve their goals. Examples can be used for several purposes, such as to introduce a programming language, to develop an algorithm to solve a problem, or to demonstrate a programming pattern [3]. Furthermore, examples are often used to show the importance of some concepts. Students must comprehend the importance of the concept, otherwise, they will continue programming without applying the concept to their source code [4].

According to Gaspar and Langevin [5], when solving a problem, students start by identifying the keywords presented in the software specifications, and they use these words to try to identify problems previously solved by them. Moreover, the Internet now offers a wide range of content that can be easily accessed by programmers. Such content may have source code examples that match a programmers' intentions very well, if not entirely.

The use of examples is a continuous practice during programmers' professional lives. Neal [3] observes that programmers with several levels of experience build code by studying, reusing, or revising software (or parts of software) written by others. To benefit from an example, programmers must understand it and also understand the concepts embodied by it [6]. However, examples are often reused without a full understanding of what they do or mean [7].

Software reuse is the process of creating software from an existing one, instead of building it from scratch [8]. Hoadley [9] proposes that software reuse may occur in three different ways: (1) as code invocation, when functions and procedures are reused; (2) as code cloning, when source code lines are copied from an example and edited to achieve a new goal; and (3) templates reuse, when learned patterns are applied in other situations. There are other kinds of classifications for software reuse. One is proposed by Sojer [10], who distinguishes two kinds of source code reuse: (1) snippet reuse, and (2) component reuse. The former is the equivalent of code cloning in Hoadley's approach. However, the author adds two branches to cloning. Code scavenging amounts to replicating several continuous lines from source code, and design scavenging, in turn, occurs when a structure composed by a large block of source code is used as a framework. Sojer's second general kind, component reuse, amounts to making use of components that have been developed, tested, and documented specifically for this purpose, such as Application Programming Interfaces (APIs), for example. These definitions complement one another, as shown in Figure 1.

In this paper, we take a semiotic perspective on programming and examine what novice programmers "communicate" through their source code. Thus, programs are viewed as message-carrying interfaces capable of communicating intent

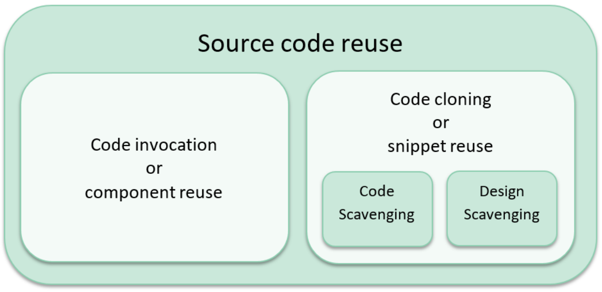

Fig. 1. Source code reuse approaches

and content. As a result, programmers who reuse code become "users of somebody else's program(s)". Following the perspective of computers as media [11]–[13], in the computer-mediated communication (CMC), a system's interface communicates its designer's intentions to users. Then, it is the user's task to interpret this source code and try to understand the meanings of the designer's message. This phenomenon is named metacommunication and has been extensively investigated and developed by the Semiotic Engineering theory [11].

In the case addressed by this paper, we observe that the source code is acting as an interface [14], mediating the communication between programmers and, just like a traditional interface, which needs to be appropriated for one to make better use of an interactive system. We believe that the interface represented by the source code of software also needs to be understood and appropriated by the programmers who, somehow, aim to use it. Thus, we are interested in the reuse of source code by invocation and cloning, and, in case of reuse by code cloning, we aim to understand how novice programmers perform this reuse, if and how they interpret the meaning of this source code and how they integrate it to their source code, and, we are also interested in understanding the metacommunicative impacts of this reuse.

The motivation for this research came from our own teaching practices, where we often observe students injecting other programmers' code into their programs. It is crucially important for teachers and learners to understand how injected code is (or can be) appropriated by novice programmers. This motivation leads us to some questions:

Q1. Why and how do novice programmers choose to reuse source code?
Q2. Considering the reuse through tools properly developed to this purpose, such as APIs, how do they learn how to interact with it?
Q3. Do they understand their own source code when it is built by reusing someone else's code?
Q4. Considering the possibility of communicating with someone else through their program's source code, with whom do novices believe they are communicating?
Q5. How do they understand the message delivered through their program's source code to other users?

This paper is an extended version from the paper published by the authors in the Proceedings of the 17th Brazilian Symposium on Human Factors in Computing Systems [15]. In this work, we present the results of three qualitative studies carried out to answer these questions. We look at how novice programmers reuse source code, how they understand them, and how they integrate them into their source code. Finally, we discuss the results given the importance of the program's metacommunication understanding by programmers and the elements that might help programmers, researchers, and teachers to evaluate and analyze the levels of understanding and appropriation a programmer has about a source code.

The following sections will present our Theoretical Backgrounds, the Research Design, and our Findings. Next, we present a Discussion and Conclusions regarding the results we found.

II. Theoretical Backgrounds

In this section, we present the theoretical background of Semiotic Engineering and Appropriation.

A. *Semiotic Engineering*

Semiotic Engineering [11] is a specialized semiotic theory mainly based on Peirce's [16] and Eco's [17] general semiotic theories. Its main study object is metacommunication between software producers (designers or developers) and software consumers (users). According to the theory, metacommunication occurs when users interact with software because messages expressed by the interface ultimately communicate how, when, where, and why users should communicate (hence "meta") with software. Users' purposes must be in line with the design views contemplated by designers and implemented by programmers, who thus become the senders of the metacommunication message. This theory defines the following abstract model (or template) of the semantic content of metacommunication:

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

This Metacommunication happens while the receivers (that is, software consumers) interact with the user interface. The interface represents producers at interaction time and enables their communication (mediated by the software) with their consumers. The main difference from Semiotic Engineering compared to other Human-Computer Interaction (HCI) theories is that it postulates that software designers and developers participate (mediated by the interface) in users' interaction processes.

In this research we extended the metacommunication process to software internal development layers, bringing it from the HCI field into the Human-Centered Computing (HCC) [18]–[20] field (where questions related to human interpretation and communication cover human processes, even if these subjects are not end users, but rather programmers, software architects, system analysts, etc.) [20]. In our study of metacommunication process among programmers through software source code, we have made two adaptations to the original Semiotic Engineering definitions. The first one refers to the receivers of metacommunication (from now on, programmers rather than typically non-technical end users). The second adaptation refers to the interface itself (from now on, a piece of code, with both its textual and executable facets, rather than the end user interface).

B. Appropriation

From the sociocultural perspective, appropriation is defined as the process of taking something that belongs to others and making it one's own [21]. From the technological perspective, appropriation is defined as how users evaluate and adopt, adapt and integrate technology into their daily practices [22]. Nevertheless, appropriation of technologies may not be interpreted as only a phenomenon that occurs when the software is being used in its expected domain, but also interpreted as a set of continuously activities performed by users to make this software works in a new environment, taking this artifact as a material and a significant object [23].

According to Dourish [24], appropriation is similar to customization, though, it refers to the adoption of technology standards and its transformation n a deeper level. Appropriation involves customization (which means the explicit reconfiguration of a technology to make it fits a specific need) but also may only involve making use of technology to a different purpose from which it was developed to attend.

In a similar way that technology is capable of shaping users' practices, it is also shaped by the users. Carroll et al. [22] defined a Model of Technology Appropriation, composed by three levels: the first level starts at the moment that the technology is presented to the users, and they face the decision of use it or not. After choosing to use this technology, users start the appropriation process in which they test, evaluate, and adapt this technology to their needs. Finally, the last level occurs when users integrate this technology into their practices, and it is considered stabilized.

Within this scope, software source code are technologies, and, for this reason, users need to appropriate from them to make better use. In this paper, we take the source code of software not only as words written in a programming language, through which we can solve a computational problem. We observe it from the Semiotic Engineering perspective, which considers software interfaces as a mean of communication between the interface designer and its users.

III. Related Works

Regarding source code reuse, the work from Rosson, Ballin, and Nash [25] approaches the challenges and opportunities for informal programming activities performed by professional programmers, who developed and maintained online content as part of their daily tasks. The authors observed that participants from their study used a few times the copying and pasting strategy. However, they used source code from other programmers as a pattern to something they were trying to learn. One participant cited the copying and pasting strategy as an opposite of learning, by reporting that *"I don't like to copy because I like to learn how to do it myself"*. The authors defend that, once source code is used as a learning tool, development tools should support questioning regarding how a source code works in order to make more accessible the learning and the reuse.

The work from Ichinco and Kelleher [26] focus on novice programmers and aims to understand more about the challenges related to reusing examples, and to list the existent barriers and strategies used by these programmers to use a source code example. During the study they performed, participants received six programs to be completed, each focused on a different programming concept, and participants should change the given program in order to create a specific animation. Participants were also provided with a code example for each task they should perform. One of the barriers observed by the authors was related to understanding the example, which made it difficult to reuse it. In some cases, participants did not even understand how the example was related to the task they were performing, and, due to that, they did not consider using it to help them. Other barriers were related to understanding their source code: sometimes they believed to knew how to complete the task; however, their ideas were incorrect or, sometimes, they did not understand how their source code works since it did not behave like expected. Besides, the authors noted that participants were slow to reach the "realization point", which is the moment when participants realized which part of the example could be used in their tasks, because they concentrated on running the example, rather than reading the example and trying to understand it.

Also related to our research, the work from Hoadley et al. [9] presents two studies regarding when, why, and how novice programmers reuse source code. During their studies, the authors have as goal to observe if the act of performing a summarization of source code would increase the probability of reuse and if there was a relationship among the quality of this summarization and the reuse. Besides that, they investigated if the programmers' beliefs about reuse could influence somehow their performance.

The authors observed that programmers must believe that reuse is possible and desirable to, in fact, reuse a source code while solving new problems. Around 20% of participants rejected the idea of source code reuse. Some of them understand it as a form of plagiarism, and some of them did not trust the source code written by others. Consistent with these beliefs, these same participants performed reuse in only 5% of the cases. On the other hand, 80% of participants remained neutral or in favor of reusing, claiming that source code reuse is an efficient practice, which reduces complexity and makes debugging tasks simpler. In this second group, it was observed that the understanding a programmer has about a source code would influence the frequency and the ways he will reuse it. Participants from the studies were required to perform a summarization of a source code according to their understanding of it. The source code summarized abstractly was more reused than those summarized algorithmically, and the source code summarized on an algorithmic level was less reused than those incorrectly summarized. The authors classified the summarizations as abstract (those in which participants correctly described the relationship among inputs and outputs, regardless of how it was done), algorithmic (those in which participants correctly described the actions of the function, without specifying the relationship among inputs and outputs) and incorrect [9]. It suggests that reuse-favorable beliefs, combined with the abilities to provide abstract summarizations, may increase source code reuse. Finally, the authors suggest that computer science courses must emphasize the understanding of source code as a matter to "improve" reuse. This work shows that exists a direct relation among source code understanding and reuse.

Software reuse can also be performed through APIs. Robillard [27] carried out a research with 80 professional programmers from Microsoft addressing questions about barriers that may difficult the use of an API. From this group, 49% were programmers from junior to intermediate levels. Regarding the ways they learn how to interact with an API, 78% of them pointed the API documentation, 55% pointed the use of examples, 34% said they made experiments using the API, 30% of them read papers, and, 29% of them used to ask for coworkers help.

With the arrival of the HCC area, issues related to human aspects have been raised concerning the several stages and artifacts used during software development process. The Semiotic Engineering theory has been contributing to the HCC area by researching and providing means to support the study of communicative aspects in these artifacts. In Afonso's [28] research, the author examines APIs as a process of communication among API designers and the programmers who are using it. In this communication, the designer expresses to the programmer an encoded message through which he explains how the programmer must use the API functionalities. His work proposes a conceptual framework based on semiotic and cognitive theories, which highlights the pragmatic aspects involved in this communication. The framework can be used as an epistemic tool to support the development of APIs. The work from Afonso helps to compose the book *Software Developers as users: Semiotic Investigations in Human-Centered Software Development* [29], which presents contributions to the advancement of the HCC area by using Semiotic Engineering theory, and, provides a set of conceptual and methodological tools to support research on how human meanings are manifested during software development and use.

IV. Research Design

In order to investigate how novice programmers reuse source code and if this reuse affect their understanding about the software they built, we conducted three qualitative studies, detailed as follows [1]. In table I, we present the relationship between each study and the research questions they helped answer. These studies are part of a more extensive ongoing research which aims to support novice programmers during the reuse of source code. All participants provide access to their produced source codes, and they signed the Informed Consent Form. More details regarding the research design and the findings are available in the Ph.D. thesis from Müller [31].

TABLE I
Relation among research questions and studies

Question	Study One (S1)	Study Two (S2)	Study Three (S3)
Q1	X	X	X
Q2			X
Q3	X	X	
Q4	X	X	
Q5	X		

A. Study One

1) Context and Goal: The S1 was carried out with the students from the introductory course about algorithms and programming offered to the undergraduate programs of Computer Science and Information Systems. The course's teacher proposed an exercise where students needed to build a program to manage a bookstore. As an example, the teacher made available to students, through the course's website, a solution to this exercise. A couple of weeks later, the students should build a program to register users' evaluations about educational games. We checked the delivered programs, and we identified that some parts of the programs were exactly like parts from the bookstore project. This fact led us to wonder how the appropriation process happens when programmers reuse code.

The study's goal was to understand general aspects of code reuse by code cloning. Furthermore, in the cases in which the example was reused, we aim to comprehend if the students understand their source code and observe if they appropriate from it.

2) Study procedure: This study was conducted in two steps, described as follows:

- Analysis of students' delivered source code in order to check if and how they reused the example: to analyze the source code of the 23 students involved in the study, we used Moss[2], a tool provided by Stanford University, that calculates metrics on texts' similarities. In our study, these texts comprised the source code from the students and the source code from the Bookstore example. Then, we invited to an interview the two novice programmers who produced source code that were the most like the example and the two novice programmers who produced source code that were the least similar to the example.

[1]Some steps from the studies will be omitted due to the fact that they are related to a research about self-expression through source code [30] and they are not relevant to the goals of this paper

[2]https://theory.stanford.edu/ aiken/moss/

- Interview with the four students selected according to results from the first step: during this interview we asked the participants to (a) explain some chunks from their produced source code (due to the size of the entire source code (between 585 and 4100 lines), we have selected some representative sections of each); (b) answer questions related to their initial steps to develop a new software, when and why they look for a source code example, how they search for a source code example, and their perceptions about their program as a mean of communication (related to that, we asked them about who they might be communicating with); (c) fill out the metacommunication template offered by the Semiotic Engineering.

3) Participants' profile: The profile of the participants we interviewed is presented in Table II.

TABLE II
S1 INTERVIEWEES' PROFILE

Participant	Graduation Program	Similarity index
S1P1	Computer Science	44%
S1P2	Mathematics	33%
S1P3	Computer Science	3%
S1P4	Information Systems	1%

B. Study Two

1) Context and Goal: The study was conducted at the end of an introductory course about programming offered to students of the Civil Engineering and Production Engineering undergraduate programs. The goal of this study was to deepen our knowledge about the subject of source code reuse by code cloning. The teacher shows, as an example, a source code that calculates and presents the initial 20 terms from the Fibonacci sequence. This example included a screen prototype responsible for presenting the result to the user. A few classes later, she asks the students to develop a program that calculates the sum of N initial terms from the Fibonacci sequence, on which N is a number provided the program's user. We observed that the students reused the example of the teacher, reusing even the screen prototype and keeping the variable's and component's name patterns from the example.

2) Study procedure: This study was conducted in two steps, described as follows:

- Analysis of students' delivered source code in order to check if and how they reused the example: to analyze the source code of the 30 students involved in the study, we used the JPlag tool[3], which, such as Moss, calculates metrics of text similarities, to analyze the similarity between the source code example and the source code provided by the students. After, we invited all the students to participate in an interview, six out of 30 students agreed to engage in.

[3]At the time we were conducting this study, the tool Moss, used in the previous study, was facing an instability. Due to that, in this study, we used JPlag tool (https://jplag.ipd.kit.edu/)

- Interview with the six students who accepted the invitation: during this interview we asked the participants to (a) explain their produced source code (the source code produced by them had less than 20 lines of code each.); (b) answer questions related to their initial steps to develop a new software, when and why they look for a source code example, how they search for a source code example, and their perceptions about their program as a mean of communication. Related to that, we asked them who they might be communicating with (the questions answered by them were the same from S1). During this study we did not ask the participants to fill out the metacommunication template offered by the Semiotic Engineering, since the code they produced was not intended to build a software solution but to solve a simple logical problem.

3) Participants' profile: About whose that participated from the interview, their profile is presented in Table III.

TABLE III
S2 INTERVIEWEES' PROFILE

Participant	Graduation Program	Similarity index
S2P1	Civil Engineering	84%
S2P2	Civil Engineering	84%
S2P3	Civil Engineering	84%
S2P4	Production Engineering	71%
S2P5	Civil Engineering	22%
S2P6	Production Engineering	18%

C. Study Three

1) Context and Goal: The study was carried out with high school students who are participants from a programming course offered by a Brazilian University. The main goal of the course was offering free classes to students from public schools, about logic, programming concepts, mobile development, marketing, design thinking, technology tendencies, and prototyping. The study's goal was to learn and to understand more about source code reuse by code invocation in a scenario of mobile development, where the use of APIs is frequent.

2) Study procedure: During this study, we conducted semi-structured interviews carried out with groups of four participants each time. Differently from the previous studies, in this one, we did not perform a similarity analysis of the codes developed by the participants. The interview addressed questions about their initial steps to develop a new software, their needs for external tools (libraries, frameworks or APIs), their experiences with this kind of tools, how they learn how to interact with the tools they used, and their need for source code examples.

3) Participants' profile: The profile of the participants is presented in Table IV.

V. FINDINGS

In this section, we will present the results obtained from the studies.

TABLE IV
S3 INTERVIEWEES' PROFILE

Participant	Group	Age	Previous programming experience
S3P1	1	16	Yes
S3P2	1	17	No
S3P3	1	18	No
S3P4	1	17	Yes
S3P5	2	16	No
S3P6	2	18	No
S3P7	2	16	Yes
S3P8	2	17	No
S3P9	3	18	No
S3P10	3	17	No
S3P11	3	16	No
S3P12	3	16	Yes
S3P13	4	22	Yes
S3P14	4	16	Yes
S3P15	4	17	No
S3P16	4	18	No
S3P17	5	16	No
S3P18	5	17	Yes
S3P19	5	17	Yes
S3P20	5	17	Yes

A. Why and how do novice programmers choose to reuse source code? (Q1)

Regarding how novice programmers search for source code examples, we found from S1 that they often search for examples that the domain is similar to that of the application they are building. If they did not find an example with this characteristic, then they would search for examples that implement the internal operations they need to develop. The interviewee S1P2 reported that he only seeks for an example from the same application domain, though, he justified that *"I use a source code example as a base that can be improved until the goal is achieved."*[4]. He also mentioned that this kind of example could be used as a frame to help him to start building his own application arguing that *"many times this frame allows only the replacement of objects by those that are pertinent to the required subject"*.

S2 shows us that novice programmers frequently use examples provided by their teachers and source code previously developed during classes. We summarize their searching approaches and present in Figure 2.

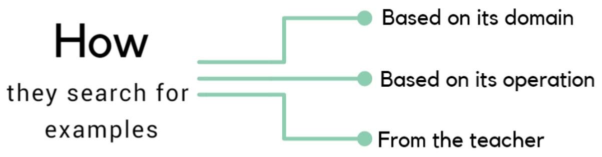

Fig. 2. How novice programmers search for examples

Related to why they need examples (Figure 3), S1P3 mentioned she uses examples to *"understand the problem's logic. If it is a question regarding the programming language, I will search for examples that represent the situation, apart from the subject. If it is a question regarding the problem's logic, I try to locate examples that can be applied in the situation, apart from the programming language or the subject."*. Participant S1P4 mentioned he uses examples to *"solve some logic problems"* and when he is stuck in a problem, and he considers that *"all alternatives of code variations were tried"*.

[4]The sentences presented in this paper were translated from Portuguese by the authors.

Some participants from S2 mentioned that they use examples to understand the problem and to optimize their applications. Regarding the need to understand the problem, S2P5 mentioned that he might need an example to understand more about a load cell, for instance, and *"to know some of the variables the problem will expose to me"*. Still, about problem understanding, S2P2 told us: *"I have the examples provided by the teacher during the classes, and when I am developing an application, I take a look at the teacher's example to check the logic used on it to achieve the results"*. About how the examples are used, S2P4 mentioned that she uses it to improve her source code (she refers to the use of examples to perform an optimization): *"I check [the example], and I work on what I have done. In fact, at this time, I already built the program, and then I fix it"*.

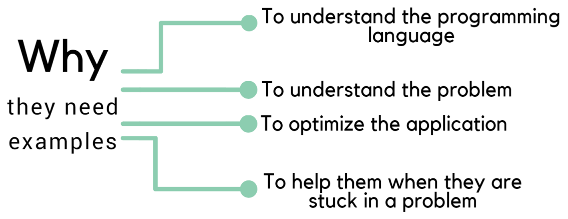
Why
they need examples
To understand the programming language
To understand the problem
To optimize the application
To help them when they are stuck in a problem

Fig. 3. Why novice programmers need examples

Regarding the ways they use the examples, they can be used as a framework that might be modified and improved in order to achieve a specific goal. As presented before, this strategy is named *design scavenging*. During S1, S1P1, S1P2, and S1P3 reported that they use example through the copying and pasting strategy. However, S1P1 mentioned that, according to the example being used, he might change his approach: *"Small source code, which requires few changes, I reuse them, changing what is necessary. To deal with more complex source code, which is usually longer, I use them as a reference. Although, even this way, I copy small parts of the example."*. The copy of small fragments of the example is defined as a *code scavenging* approach.

S2 has shown that novice programmers often reuse source code by cloning them to their own source code. Concerning that, S2P1 reported that "*in the first few times I copy and paste, however, after doing it several times, this gets etched in my brain, and then, I do not need to copy anymore.*" This same participant also mentioned that he used to perform copying and pasting by copying line by line, reading, and writing the lines. According to him, this is his approach to learn programming.

A different approach was observed in S1. Participants mentioned they use this source code as a reference to be consulted when it is needed. The same approach of source code reuse was mentioned during S2. About it, S2P3 reported: *"it is easier for me to use the source code as a reference; otherwise, I let something passes, like an operation or a variable that were not supposed to be there. So, I use it only as a reference"*. During this study, participant S2P1 mentioned that he reuses source code to save time. According to him *"during the class, we do not have much time, and sometimes the teacher asks us several things that we have to do. However, when you are trying to learn by yourself, using your free time, I believe you will try to do differently from the teacher"*.

Corroborating with the findings from studies One and Two, during S3, participants reported that they used to copy and paste small chunks of source code (code scavenging), and they also copied source code from video tutorials, used as a reference. According to them, when finding examples, they usually realize that they did not need the entire code, but only a few parts from it, that could be adapted to being reuse. About it, S3P9 mentioned that *"usually the example was not fully the code we needed, but part of this was. Then, we reused the code but needed to edit it."* and S3P8 mentioned *"you can edit the code and make it the way you need"*. They also mentioned that they do not have enough trust in the material they found on the Internet because *"usually lines are broken, something is wrong"* (S3P7). Such as mentioned by a participant from S2, one of the groups reported that after reusing a source code several times, they learn how to solve a kind of task, being able to code without the example. Figure 4 presents a summary of the ways novice programmers reuse source code.

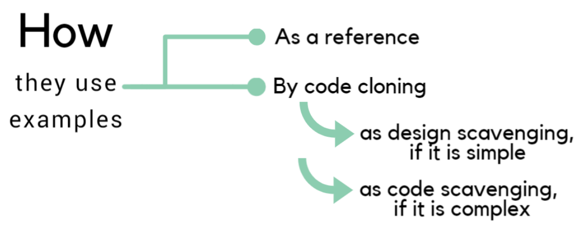
How
they use
examples
As a reference
By code cloning
as design scavenging, if it is simple
as code scavenging, if it is complex

Fig. 4. How novice programmer use examples

Additionally, during S2, we observed a programming approach, not related to reuse, but that corroborates with the previous observations, which show those novice programmers often not spend much time trying to understand what they are building. The development of source code by trial and error approach was mentioned by S2P4 *"sometimes, I did not have a basis, so I had to go by intuition. I was developing it by trial and error, coding and fixing"*. Still, S2P6 mentioned *"I develop in a way I think it will work."*.

B. Considering the reuse through tools properly developed to this purpose, such as APIs, how do they learn how to interact with it? (Q2)

The results presented in this section are exclusively from S3, due to the fact that the primary goal of the final project is the development of a mobile application, we found that such projects had great potential for code reuse through APIs. Participants from this study mentioned that their first steps while building a mobile application are planning and prototyping the app. During the planning, they chose priority features to be built, taking into consideration participants' knowledge, their available time, and the importance of the feature to the entire project. Further, they also search for APIs that may help them during development.

Regarding the use of APIs (source code reuse by code invocation), most participants mentioned they need to use these tools, especially the APIs provided by Google and Facebook, which were used to build the application login, and other APIs to developed different features of their projects. They also reused libraries provided by other programmers. A participant from Group 2 (S3P7) reported that the API was the kernel of application developed by his group. Besides that, all participants mentioned they had never used APIs or other external sources before the course.

About the ways they have learned to interact with the APIs, they used the API documentation, such as tutorials offered by this documentation, online forums, examples provided by their teachers, and video tutorials. These findings corroborate with the findings presented by [32], who observed that the Internet has being used as a tool to support learning about technologies that programmers were not familiar with. Additionally, our findings are aligned with the work of [27], who observed that programmers used to learn about how to use an API through its documentation and by examples, experiments, papers, and help from other people.

We also questioned participants about how they search for materials to help them to interact with the API. During this study, we observed that they discovered their need for an API while searching for how to fix some problem or develop a specific feature, focusing on the goal they want to reach instead of the algorithmic solution they need to achieve this goal. Only one group mentioned that they usually had an understanding about which algorithmic strategy they need, and they used it to guide their search. In this case, they reported that they seek an example only after already had developed their own source code in order to check if it is possible to improve their solution.

Finally, regarding their understanding of the reused source code, they mentioned that they invested time and effort trying to understand them. However, they were satisfied in understand only the essentials for being able to use it. Participant S3P9 mentioned *"we know what it does, but we do not know how it works"*. This is because usually, APIs works as a black box, it means, it is only possible for users to know about the required inputs and expected outputs, without knowing how it works internally.

C. Do they understand their own source code when it is built by reusing someone else's code? (Q3)

During S1 and S2, we asked the participants to explain their source code. Regarding S1, two participants high reused the Bookstore source code. Participant S1P1 was not sure about what his code does in several moments. He could neither talk about the operation of selected pieces of code nor from which part of the source code they come from. His explanation was shallow, and most of the time, he was only reading the code line by line, focusing on some details of syntax and semantics, but not details related to the role of that piece of code on his program.

On the other hand, S1P2, who also had a program that was very similar to the Bookstore project, gave a different explanation. His explanation was highly detailed, showing awareness of the role and location of the pieces of code inside the entire program and about the new lines inserted into it. This fact showed us a different level of understanding in relation to S1P1 because S1P2 was not only using the example code as a frame to create his own program, but he was also able to extend it to add extra features.

About S2 participants, from whose had source code similar to the source code example (S2P1, S2P2, S2P3, and S2P4), only S2P3 was not able to explain how his program works or explain how the Fibonacci sequence is calculated. With regard to the remaining participants, the one that caught our attention was S2P1, who started his explanation telling: *"I cannot talk too much about this one, because it was practically copied because she (the teacher) had already done it. It would be a waste of time because I had already understood how the code was working, and I only had to add the sum operation and nothing else."*. Even so, this same participant was able to explain several aspects of his source code.

D. Considering the possibility of communicating with someone else through their program's source code, with whom do novices believe they are communicating? (Q4)

Another question we addressed during the interviews was regarding the system as a mean of communication. We asked the participants with who they could be communicating with through their systems and which characteristics they believed the receiver of this communication could perceive.

About S1, only participant S1P2 mentioned the possibility of communication with another programmer. He reported that he was communicating with *"possible users or programmers located in different parts of the world"*. Both participants S1P1 and S1P3 reported they believed they were communicating to the users of their systems. During S2, some participants mentioned they were communicating with other programmers: *"I believe I am communicating only with students from the same area as mine, or someone who has questions and uses my source code as an example"* (S2P3). This participant complements by telling that *"I believe that programming must be clear and shared, I believe it can be used as an example to others, such as it served to me"*. It calls our attention that one of the participants mentioned that she was communicating with *nobody*. However, when explaining the reason, she mentioned the possibility of a communicative breakdown that could happen while communicating with another programmer: *"I believe that another person will not understand it because I used X and Y"* (S2P4). The reused examples were using variables named X, Y, and Z, and some participants who reused them kept the nomenclature pattern and even believing that this pattern would make it harder for another programmer to understand the content of such variable, the participants did not change it. Additionally, S2P1 and S2P5 mentioned a communication through the system's interface: *"I believe that I am communicating with the general audience, the clients"* (S2P1) and *"I worry about the screens, I do not know if is this, but the main goal is to make the person understand what is being done there"* (S2P5).

Regarding the characteristics that the receiver of the message could perceive, S1P1, S1P2, and S1P3 mentioned the ways the information is presented to the users. S1P2 mentioned that, while asking some information to the users, he often uses informal sentences, similar to communication with friends. S1P3 reported as characteristics of her writing style and the way she organized the system. Although he mentioned that he believed to be communicating with the user, S1P1 told that a meticulous person would use more methods and controls, or a person with a broader view would think of less likely problems, predicting this way unexpected situations. These characteristics are more likely to be perceived by another programmer who will read this source code than a final user who will only use it.

Like that, participants from S2 mentioned as perceived characteristics the way that graphical items are presented into the interface, and they reported some coding characteristics, such as code structure and variables' nomenclature pattern.

E. How do they understand the message delivered through their program's source code to other users? (Q5)

At the end of S1, we invited the participants to fulfill the metacommunication templates from the Semiotic Engineering theory. We did not mislead them about who the user might be (an end user, another programmer, or even the teacher), because we wanted to observe who the user was in their interpretation. We split the template into four parts, as follows, and the participants should complete the four sentences from the template based on their own developed programs:

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

About the first sentence, which goal is to define the user who would be interacting with the system, participants S1P2, S1P3 and S1P4 were able to clearly identify the users they were communicating with, describing the user as *"a person who* *is seeking for new tools to didactic application"* (S1P2), *"an ordinary person, a student or a teacher"*,(S1P3) or *"a teacher evaluating a new teaching tool or a research administrator analyzing the results of all evaluations"* (S1P4). We can observe in their sentences that they were aware that the appraisers could be people involved with education (such as a teacher or even a student). S1P4, who created different areas in his program, considered the existence of a researcher who would manipulate the information inserted by appraisers. On the other hand, S1P1 described the users from his application with a generic and incorrect sentence: *"somebody who works with register of games and players"*. The application aims to register educational games and teachers' opinions regarding the games. However, the registration of players was not required by the system's specification and was not developed in S1P1's system.

Regarding *"What I've learned you want or need to do, in which preferred ways, and why"*, participants S1P2 and S1P3 reported that their users *"need to select an application that fits to their students' needs and which has a satisfactory knowledge level to be clearly and objectively conveyed to them, using a nice interface which calls the students' attention"* (S1P2), and that their users *"want to store and handle information regarding games"* (S1P3). Once again, S1P1's answer was generic and referred to nonexistent features from the system: *"to register games and players, to correlate the data taking some parameters into consideration"*.

Participants S1P2 and S1P4 reported that they designed *"a system which allows you to identify from where are the other users who are using certain application, their ages, their qualifications and their opinions about the application"*, (S1P2) and a system with which *"the administrator can manage a small database regarding the participants, being able to organize and transform these data into useful information."* (S1P4). This last-mentioned participant created a system with two modules: one for administrator, and other for evaluators and he complements his sentence, by adding that *"to a regular user, the system was projected to offer a simple and effective way to expose his perceptions regarding the evaluated educational tools"*.

The sentence reported by S1P2 draws attention to the fact that he understood the kind of information his program is managing. If we compare his sentence with the sentence from the other participants (whom all provided satisfactory answers), we can see that he was the one who provided more details about what his program does, even more than those who created an entirely original program.

With respect to how the users can fulfill these systems purposes, participants S1P2 claims that the system needs to be *"offered in educational institutions that have computer labs or that are developing applications with this goal [development of educational games]"*, or they can achieve it only by following the menus (S1P3 and S1P4). About it, S1P4 says that *"you can follow the menus intuitively. Initially, you will have to register the respondents and the games. From this, the questionnaires can be answered (by the respondents, about the games). Finally, you will be able to consult the information generated from statistics and reports"*. Regarding the answers from S1P1, once again, it was vague, with no details about the system's features. He reported: *"to insert the ordered data and verify if there is any option related to what you want to know"*. The answers from participants S1P1 were generic and with little information about the system, and this characteristic was observed in some S1P4 sentences too. However, the answers from S1P2 and S1P3 were accurate, clear and objective, and showed their ownership of the messages they were delivering to their users.

From these participants, S1P1 and S1P2 were those who reused the example provided by the teacher as a framework to build their own source code. Although both have used the example in the same way, we observe that S1P1 was not fully aware of the message his application was delivering, and he did not have a full understanding of how his own source code works.

About S1P1 messages, by taking the point of view from the Semiotic Engineering theory, we observe that the designer's metacommunication message delivered by the system's interface to its users is composed of two messages: the one from the Bookstore project designer and the one from S1P1. However, although these messages complement each other, they are disconnected once S1P1 did not properly appropriate from the message used as a basis to his system. On the other hand, S1P2 showed that he was appropriate from the message delivered by his system, and he was aware of how his own system works.

It draws our attention to the fact that, when comparing the metacommunication messages from all S1's participants, the answers from S1P2 stands out, once his metacommunication message was as accurate as those from the participants who built their systems from scratch. Differently from the S1P1 case, the message delivered by S1P2's system is also composed of the same two messages, and in this case, the messages are connected to each other.

The following is a metacommunication message that, in our opinion, appropriately represents the systems developed. This message was elaborated based on the union of the answers given by the participants S1P2 and S1P3:

"You are an ordinary person, a student or a teacher (S1P3) who is seeking for new tools to didactic application (S1P2). You need to select an application that fits to their students' needs and which has a satisfactory knowledge level to be clearly and objectively conveyed to them, using a nice interface which calls the students' attention (S1P2). Then, I designed for you a system which allows you to identify from where are the other users who are using certain application, their ages, their qualifications and their opinions about the application (S1P2). Then you can follow the menus intuitively. Initially, you will have to register the respondents and the games. From this, the questionnaires can be answered (by the respondents, about the games). Finally, you will be able to consult the information generated from statistics and reports (S1P3)".

This section presented the results we found through the performed studies. We presented general aspects regarding source code examples and reuse, such as how they search examples to help them, why they need these examples and, when they decide by reusing it, how this reused is done. We also addressed questions regarding communication, and we found that novice programmers often consider the possibility of being communicating with other programmers through the source code in a scenario where their source code is being used as an example. Besides that, we presented findings about the impacts that source code reuse might have on the understanding and the metacommunication that programmers have about their own source code.

F. Discussion

Learning how to program and how to develop the Computational Thinking [1] may be a complex task. Beyond the regular challenges from this learning process, programmers often must work using source code written by others. In the case of experienced programmers, they frequently need working with someone else's source code during software development process [33]. Software reuse practices are widely disseminated as a means of raising productivity using APIs and function, project patterns, and chunks of source codes [8]. In the case of novice programmers, they need source code examples to show how to develop an algorithm to solve a problem, to demonstrate programming patterns, or to present some aspects from a programming language [3]. Such as the experienced programmers, the novices, also reuse source code examples [3], including this code into their source code, and, carrying out the necessary adjustments to achieve their goals.

We conducted our investigation regarding source code reuse by novice programmers by using the Semiotic Engineering theory and its recent contributions to the HCC area, and we visualize the source code as an interface through which the programmer who wrote the source code "talks" with the programmer who is reusing it. By using this perspective, we are allowed to see this source code as an implicit discourse that incorporates the programmers' intentions regarding how, whom, and where this source code may be used. We conducted three studies regarding source code reuse, including questions about programmers' understanding about their own discourse. The results of these studies could help us to deepen our understanding about the appropriation of source code during reuse. By analyzing the results, we could identify three distinct scenarios. The authorial scenario, in which are the participants who developed their source code from scratch. The non-authorial scenario, composed by those participants that reused the example, but were not aware about how it works, and which is the metacommunication message being delivered. Finally, a co-authorial scenario, composed by those that, despite the reuse of an example, were aware about how their source code works and the metacommunication message being delivered.

Taking as example cases of reuse observed during S1, despite both S1P1 and S1P2 had widely used the example, we identified different understandings about the code they produced, and the message they are communicating through this code.

As we observed, S1P2 had the same accuracy in his descriptions (such as S1P3 and S1P4 who built their programs without using the provided example). This participant was able to describe the commands we showed them as a unique concept, according to the command's goal, and specify essential details about their metacommunication.

The results we found introduced a reflection about the differences presented by S1P1 and S1P2 during the S1. Both participants fulfilled their goals by building a functional program that executed the required tasks. However, S1P2 showed a more precise understanding about the program, as precise as the understanding of those who created the source code without any external reference. Thus, we considered S1P2 a co-author of the program he built with the example's programmer (in this case, the teacher). He was not only reusing the code, but he also interpreted and understood its operation, and then reused it, aware of several meanings encoded inside this code.

During this research, we reflected about the "meaning of the meanings". It becomes clear that every piece of code, regardless of its creator, bears several meanings that will be decoded by the one who will use it. This user is the one that will define what the code means. Such signification, as well as the appropriation of this code, depends on the level of understanding the user has.

About levels of understanding, we observed, through the studies, that there are:

- A low level, where a programmer only paraphrases or explains what the code does by "translating" it to a natural language, line-by-line. This approach is named *algorithmic summarization* [9]
- An intermediate level, where a programmer has an abstraction level on the program's syntactic structure, being able to explain a set of commands as a unique concept based on this code's goal. This approach is named *abstract summarization* [9]. We observe that, in this level, there can occur two sublevels:
  - Without application domain references: it means that the programmer knows what the code does. However, this programmer is not capable of identifying pragmatic aspects, such as for what this code can be used.
  - With application domain references: unlike the previous one, the programmer in this level is capable of identifying some aspects about the source code, such as application domains and business rules to which it can be applied.
- An advanced level, where the programmer is not only able to do an abstract summarization of the code but also add elements, which refer to the intentions associated with programming. The level can present two sublevels:
  - Without referring users' intentions: it means that the programmer can identify the message passed through

a source code and the intentions encoded on it. However, the programmer is not able to identify the user who is expected to consume this message.
  - Referring users' intentions: Unlike the previous one, the programmer in this level is capable of understanding the intentions of the users who will consume this code, by knowing who they are, what they expect and/or how they intend to use the program.

About appropriation, it is not ontologically defensible if the programmer is not aware of the specific aspects of his development situation since these aspects are connected to pragmatics. Therefore, we understand that appropriation only happened from the level Intermediate II of understanding. Before this level, the programmer can manifest understanding, but not explicit appropriation.

Thus, we identified the levels of appropriation as only two possible ones:

- A lower level, which only happens when the programmer is able to transfer the code to the user's required domain, but not to make explicit his own intentions or the intentions that his user must have.
- A higher level that will happen when the programmer is also able to identify some elements related to the intentionality behind the code (his own intentions or users' intentions). We also understand that programmers on Advanced I and Advanced II levels have the same design acumen. The fact that a programmer is not able to refer to intentional elements related to the user will not make his appropriation "worse" than the other case. It is possible that the program built by this programmer has less usability or communicability; however, it cannot be considered an appropriation problem.

In Table V, we presented a set of elements to support the classification of understanding and appropriation levels, as described before. These classifications are the result of observations made during the studies, and they can be used in order to analyze reuse made by programming professionals or even programming students or lay users.

Another result of this work was to show the metacommunication template from Semiotic Engineering, originally proposed to build and/or evaluate interfaces, being used in a more HCC perspective. We used the template to support our investigation, which showed us its potential and possible usage in research about reuse by professionals, computer science students, or even lay users who use programming in order to achieve some task. Besides that, the template is what made us able to observe how a programmer or any kind of end user sees the intentions he encoded in a program and its source code. Moreover, the information provided by the participants' answers about the template was crucial during the research to establish the understanding and appropriation levels we defined.

Regarding the understanding and appropriation during the reuse of code, we identified from each understanding and appropriation level are the studies' participants (those who reused the bookstore code), according to the skills they presented during the studies.

From S1, participant S1P1 and participant S1P2 were those who reused the example provided by the teacher. S1P1's explanations regarding his system's working process and regarding the metacommunication template showed to us that he was able only to perform an algorithmic summarization. Thus, we can classify his *understanding level* as *low* and his *appropriation level* as *no appropriation*. On the other hand, participant S1P3 not only provided an abstract summarization, as he mentioned several aspects regarding the business rules of his system. Besides that, this participant was able to identify metacommunicative aspects of his system, referring to who its users would be and which could be their intentions regarding the system being used. Based on this, we can classify his *understanding level* as *advanced II* and his *appropriation level* as *higher*.

Related to S2's participants, those who reused the example were S2P1, S2P2, S2P3, and S2P4. During this study, due to the small size of the application they developed, we did not ask them to fulfill the metacommunication template. Without this information, we cannot establish if they were in advanced levels of understanding or higher levels of appropriation. However, as we already mentioned, only S2P3 was not able to explain his own source code. In this case, he did not provide an algorithmic summarization, but he tried to perform the abstract summarization, without success. Based on this, we frame this participant as *low level of understanding* and *no appropriation*.

Regarding the remaining participants, they could perform an abstract summarization, though at no moment they mentioned anything regarding business rules once the system specification had only one goal. Therefore, we have no information to classify them as more than an *intermediate I level of understanding*, and, *no appropriation*.

As we have seen, the size of the system can impact how and how much a programmer can appropriate from it. Also, it is essential to highlight that we believe that their skills can vary according to the program they are building. Factors such as knowledge about a programming language, business rules, or even the time available to build the program can be important factors in order to change their understanding and appropriation levels.

Taking into consideration the studies we carried out, we observe through S1 and S2 that the reuse of an example can affect the understanding programmers have about their own source code, being able to observe cases in which they were not able to explain how the source code written by themselves works. Nonetheless, these same studies showed other cases in which the programmers were able to appropriate from the reused source code, incorporating it to his own code and understanding the relations between it and his own goals. We consider the appropriation as the final goal we aim to achieve in a scenario where programmers are aware about how their source code work, even if they were written using the "words" other programmers.

Regarding S3, from this one emerges a scenario in which

TABLE V
APPROPRIATION CLASSIFICATION ACCORDING TO UNDERSTANDING LEVELS

Understanding level	**Algorithmic summarization**	**Abstract summarization**		**Abstract summarization and referring programmers' intentions**		**Appropriation level**
		Without domain references	**With domain references**	**Without users' intentions**	**With users' intentions**	
Low	X					No appropriation
Intermediate I		X				
Intermediate II			X			Lower
Advanced I				X		Higher
Advanced II					X	

novice programmers need source code examples to teach them how to use an API. However, due to the black box nature of APIs, which usually do not present their source code, we cannot determinate if participants appropriated or not from the source code's message, once they did not have access to this message.

Nevertheless, S3 presented to us an interesting information, corroborating with works from other [32] [27], that, one of the main alternatives used by a programmer to learn how to interact with an API is through its documentation. The documentation of an API is an abstract summarization which is written with the goal to support users understanding its general aspects, such as, required inputs, and expected operations and outputs. In Table V, we presented that, while reusing source code by cloning. If the programmer can perform an abstract summarization of the source code he created, he is in an Intermediate I or Intermediate II level of understanding.

During our studies, we observe that some of these programmers were aware that in some situations the user of their source code could be another programmer, who will use it as an example, and who will start a new cycle of interpretation of this source code (Figure 5). By providing an abstract summarization (in the form of documentation) of a source code, the programmer helps other programmers to understand better the code they will reuse and improve the capability of appropriation this programmer will have.

The studies showed that novice programmers often reuse source code in several ways and for several reasons. Regarding the ways, the reuse of source code occurs as design scavenging, when a large block of source code is used as a framework to a new source code and occurs as code scavenging when the programmer opts by copying small blocks of source code. An interesting fact we observed was that these programmers prefer to perform a copy line-by-line, using the source code as a reference, avoiding this way the insertion of unnecessary lines. Regarding the reasons, novice programmers seek for examples that can support them in understanding the problem or can help them to optimize their source code.

The time may have come to stop framing programming as no more than problem-solving. Our studies have shown that it is a technologically mediated social communication process going on in programming activities, as well, and that computer code carries the imprint of human intentions and meanings.

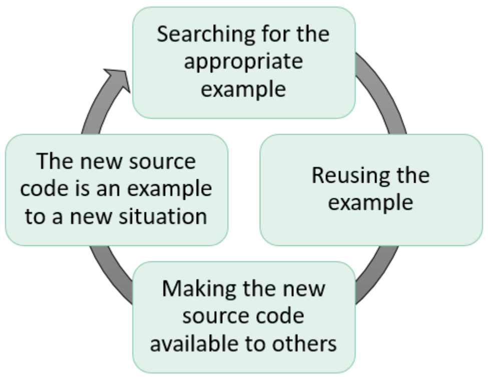
Searching for the appropriate example
Reusing the example
Making the new source code available to others
The new source code is an example to a new situation

Fig. 5. Examples reuse cycle

This view also accommodates with some advantages the easily observable fact that not only professionals are programming software these days. End users are also programmers and software developers. Viewing end user programming as a case of self-expression (in addition to problem-solving, or not) may be advantageous in teaching computational thinking for school children, for example, [34].

VI. CONCLUSIONS

Programmers use programming not only to solve problems but also to express something to consumers. This communicative process is a continuous cycle since the programmer is always changing roles between producer and consumer. Hence, we must carefully address questions about appropriation in this specific context since technology has become increasingly part of people's life and, consequently, there is a need for qualified professionals as well as appropriate software.

This research presented the results of studies that are part of ongoing research regarding how programmers reuse source code from other programmers, using them to build their own programs. Also, we present aspects regarding how programmers learn to interact with a source code when it is delivered through an API. It is necessary to comprehend how programmers understand and how they appropriate from these codes, and the impacts their ways to reuse code have over the quality of programs they are creating. In order to conduct this investigation, we appropriated from the Semiotic Engineering theory and its contributions to the HCC area [29]. In this way, we observed source code as an interface, which allows a conversation between the programmer who wrote the source code being reused and the programmer who is reusing it. Based on this perspective, we understand that this source code carries an implicit speech that incorporates the programmer's intentions regarding how, whom, and where this source code can be used.

Additionally, we presented conditions related to the impacts reuse of code has. In all the cases we analyzed, the software delivered by the participants who performed reuse were functional and, it was achieving its goals (even if some few mistakes). However, not all participants were aware of the message their software was communicating. To support investigations regarding source code reuse and its consequences, we presented a set of elements that can help us identify a programmer's syntax, semantics, and intentional understandings about a produced code and, with this, classify his appropriation about the program he built by code reuse. The classification might be useful not only for helping researches but also for teachers, companies R&D and programmers themselves to help them to understand and to evaluate the code's reuse made by programmers. However, this requires further investigation. Furthermore, it shows how the metacommunication message concept from a semiotic theory proposed to HCI can be used in a different context, bringing out human aspects of those who are responsible for building computational artifacts we daily use.

We believe that our work can call programming teachers' attention to the fact that we must take into consideration the time the students need to reflect on what they are doing. The process of reflection about these materials (source code) is a necessary step to solve a problem. Schön's [35] perspective about design is that there must be a reflection on action. When a designer starts his work, he must identify and interpret all elements involved in his development situation and know all the possibilities and limitations of the technology he needs to use. The designer's ideas must be represented in some way, allowing him to talk with this material by reflecting and expressing his new ideas by questioning *"and if I define in this way?"*, or *"it does not look good for me"*. The source code being reused is one of these elements, and as mentioned before, programmers must know and understand its limitations and appropriate from the code in order to make possible to reflect on its role in their solutions.

Nonetheless, we would like to mention the work from Hoadley et al. [9], which observes that when performing as abstract summarization, the probability of reuse increases. Moreover, they observed that sometimes programmers consider that an understanding in the algorithmic level is enough. However, as we mentioned previously, this understanding can be resumed as the capacity to translate source code lines, which were written in a programming language, to the natural language. We want to highlight that programmers may not be able to perform this kind of summarization due to the fact they do not know how to do it. Therefore, to support these students while reusing activity, we need to teach them how to perform meaningful summarizations and provide tools and methods that can support them during this activity.

As limitations of this research, we highlight its educational perspective, once the studies were conducted with novice programmers who were receiving a college education. Therefore, our results may not reflect the perceptions of self-taught programmers nor professional developers. Also, due to the fact we had a small number of participants during the studies, it not possible to perform a predictive interpretation based on our results.

Finally, as next steps of this research, we aim to work on the development of an epistemic artifact to support programmers, especially the novice ones, during the source code reuse activity. Our proposal is based on the use of the metacommunication template offered by the Semiotic Engineering theory [11], to support students to generate meaning to source code they want to reuse.

With this, we hope to contribute to the HCC area by warning these programmers while building their computational thinking about the importance of comprehending the meanings of what they are developing. We warn that this comprehension must be not related only to cognitive aspects, and it needs to be extended to source code metacommunicative aspects. We also hope to contribute to the development of the process of teaching and learning programming, presenting to programmers and teachers a perspective on which programming can be treated as more than a way to solve problems, but also a tool through which programmers can communicate with each other and express themselves.

Acknowledgment

We would like to thank all the participants for the time provided for this research. Clarisse S. de Souza thanks CNPq, the Brazilian National Council for Scientific and Technological Development, for partially supporting this research (Grant 304224/2017-0). This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

[1] J. M. Wing, "Computational thinking," *Commun. ACM*, vol. 49, no. 3, pp. 33–35, Mar. 2006. [Online]. Available: http://doi.acm.org/10.1145/1118178.1118215

[2] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 83–137, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1089733.1089734

[3] L. R. Neal, "A system for example-based programming," *SIGCHI Bull.*, vol. 20, no. SI, pp. 63–68, Mar. 1989. [Online]. Available: http://doi.acm.org/10.1145/67450.67464

[4] K. Malan and K. Halland, "Examples that can do harm in learning programming," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 83–87. [Online]. Available: http://doi.acm.org/10.1145/1028664.1028702

[5] A. Gaspar and S. Langevin, "Restoring "coding with intention" in introductory programming courses," in *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education*, ser. SIGITE '07. New York, NY, USA: ACM, 2007, pp. 91–98. [Online]. Available: http://doi.acm.org/10.1145/1324302.1324323

[6] E. Avidan and D. G. Feitelson, "Effects of variable names on comprehension: An empirical study," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 55–65.

[7] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 31:1–31:37, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2622669

[8] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, Jun. 1992. [Online]. Available: http://doi.acm.org/10.1145/130844.130856

[9] C. M. Hoadley, M. C. Linn, L. M. Mann, and M. J. Clancy, *When and why do novice programmers reuse code?* Ablex Publishing Company, 1996, pp. 109–130.

[10] M. Sojer, *Reusing Open Source Code: Value Creation and Value Appropriation Perspectives on Knowledge Reuse*, ser. Innovation und Entrepreneurship. Gabler Verlag, 2010. [Online]. Available: https://books.google.com.br/books?id=-z60hspDTlAC

[11] C. S. de Souza, *The Semiotic Engineering of Human-Computer Interaction (Acting with Technology)*. The MIT Press, 2005.

[12] A. Georgakopoulou, *Pragmatics in Practice*. John Benjamins Publishing, 2011, p. 326.

[13] J. Kammersgaard, "Four different perspectives on human-computer interaction," *Int. J. Man-Mach. Stud.*, vol. 28, no. 4, pp. 343–362, Apr. 1988. [Online]. Available: http://dx.doi.org/10.1016/S0020-7373(88)80017-8

[14] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers are users too: Human-centered methods for improving programming tools," *Computer*, vol. 49, no. 7, pp. 44–52, July 2016.

[15] L. Müller, M. S. Silveira, and C. S. de Souza, "Do i know what my code is "saying"?: A study on novice programmers' perceptions of what reused source code may mean," in *Proceedings of the 17th Brazilian Symposium on Human Factors in Computing Systems*, ser. IHC 2018. New York, NY, USA: ACM, 2018, pp. 17:1–17:10. [Online]. Available: http://doi.acm.org/10.1145/3274192.3274209

[16] C. S. Peirce, C. Hartshorne, and P. Weiss, *Collected Papers of Charles Sanders Peirce*, ser. Collected Papers of Charles Sanders Peirce. Belknap Press of Harvard University Press, 1932. [Online]. Available: https://books.google.com.br/books?id=u9fWAAAAMAAJ

[17] U. Eco, *A Theory of Semiotics*, ser. Advances in semiotics. Indiana University Press, 1976. [Online]. Available: https://books.google.com.br/books?id=BoXO4ItsuaMC

[18] L. Bannon, "Reimagining hci: Toward a more human-centered perspective," *interactions*, vol. 18, no. 4, pp. 50–57, Jul. 2011. [Online]. Available: http://doi.acm.org/10.1145/1978822.1978833

[19] S. Choi, "Understanding people with human activities and social interactions for human-centered computing," *Human-centric Computing and Information Sciences*, vol. 6, no. 1, p. 9, Jul 2016. [Online]. Available: https://doi.org/10.1186/s13673-016-0066-1

[20] A. Jaimes, D. Gatica-Perez, T. S. Huang, and N. Sebe, "Guest editors' introduction: Human-centered computing–toward a human revolution," *Computer*, vol. 40, pp. 30–34, 05 2007. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MC.2007.169

[21] J. V. Wertsch, *Mind as Action*. Oxford University Press, 1998. [Online]. Available: https://books.google.com.br/books?id=73Vv7Y3vf14C

[22] J. Carrol, S. Howard, J. Peck, and J. Murphy, "A field study of perceptions and use of mobile telephones by 16 to 22 years old," *Journal of Information Technology Theory and Application*, vol. 4, no. 2, pp. 49–61, 2002.

[23] G. Stevens, V. Pipek, and V. Wulf, *Appropriation Infrastructure: Supporting the Design of Usages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 50–69.

[24] P. Dourish, "The appropriation of interactive technologies: Some lessons from placeless documents," *Computer Supported Cooperative Work (CSCW)*, vol. 12, no. 4, pp. 465–490, Dec 2003. [Online]. Available: https://doi.org/10.1023/A:1026149119426

[25] M. B. Rosson, J. Ballin, and H. Nash, "Everyday programming: Challenges and opportunities for informal web development," in *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, Sept 2004, pp. 123–130.

[26] M. Ichinco and C. Kelleher, "Exploring novice programmer example use," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2015, pp. 63–71.

[27] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE Softw.*, vol. 26, no. 6, pp. 27–34, Nov. 2009. [Online]. Available: https://doi.org/10.1109/MS.2009.193

[28] L. M. Afonso, "Communicative dimensions of application programming interfaces (apis)," Ph.D. dissertation, Pontifícia Universidade Católica do Rio de Janeiro, 2015.

[29] C. S. de Souza, R. F. d. G. Cerqueira, L. M. Afonso, R. R. d. M. Brandão, and J. S. J. Ferreira, *Software Developers As Users: Semiotic Investigations in Human-Centered Software Development*, 1st ed. Springer Publishing Company, Incorporated, 2016.

[30] L. Müller, M. S. Silveira, and C. S. de Souza, "Mine, yours, ours: Examples reuse and the self-expression of programming students," in *Proceedings of the 14th Brazilian Symposium on Human Factors in Computing Systems*, ser. IHC '15. New York, NY, USA: ACM, 2015, pp. 30:1–30:10. [Online]. Available: http://doi.acm.org/10.1145/3148456.3148486

[31] L. Müller, "Uma abordagem semiótica para apoiar programadores iniciantes durante o processo de reúso e de apropriação de códigos-fonte," Ph.D. dissertation, Pontifícia Universidade Católica do Rio Grande do Sul, 2017.

[32] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09. New York, NY, USA: ACM, 2009, pp. 1589–1598. [Online]. Available: http://doi.acm.org/10.1145/1518701.1518944

[33] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 255–265. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337254

[34] I. T. Monteiro, L. C. de Castro Salgado, M. P. Mota, A. L. Sampaio, and C. S. de Souza, "Signifying software engineering to computational thinking learners with agentsheets and polifacets," *Journal of Visual Languages & Computing*, vol. 40, pp. 91 – 112, 2017, semiotics, Human-Computer Interaction and End-User Development. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1045926X16300234

[35] D. A. Schön, *The Reflective Practitioner: How Professionals Think in Action*. Taylor & Francis, 2017. [Online]. Available: https://books.google.com.br/books?id=OT9BDgAAQBAJ