Do I Know What My Code is "Saying"?

A study on novice programmers' perceptions of what reused source code may mean

Luana Müller
PUCRS
Porto Alegre, RS
luana.muller@acad.pucrs.br

Milene Selbach Silveira
PUCRS
Porto Alegre, RS
milene.silveira@pucrs.br

Clarisse Sieckenius de Souza
PUC-Rio
Rio de Janeiro, RJ
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 about the source code that they inject it in their own programs. The question is even more important for novices, who are trying to learn what programming is and how it should be practiced in larger scale. In this paper we present the results of an ongoing research using a semiotic approach to investigate how programmers send and receive, through messages inscribed in the source code of the programs they write or reuse, implicit and explicit communication about what such source code "means" to them and others. We carried out two studies with novice programmers and results suggest that source code reuse may impact the comprehension that programmers have about their own source code. In addition, how it impacts their understanding about the messages that are being communicated through their programs.

CCS CONCEPTS

• **Human-centered computing** → **Empirical studies in HCI**;
• **Software and its engineering** → *Reusability*;

KEYWORDS

Source code reuse, Semiotic Engineering, novice programmers, metacommunication

ACM Reference Format:
Luana Müller, Milene Selbach Silveira, and Clarisse Sieckenius de Souza. 2018. Do I Know What My Code is "Saying"?: A study on novice programmers' perceptions of what reused source code may mean. In *17th Brazilian Symposium on Human Factors in Computing Systems (IHC 2018), October 22–26, 2018, Belém, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3274192.3274209

1 INTRODUCTION

While learning how to program, novice programmers need to face the difficulties of learning how to think computationally [25]. According to Keheller and Pausch [14], 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 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 [19], integrating them into their own source code and performing the necessary adjustments to achieve their goals.

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 and content. As result, programmers who reuse code become "users of somebody else's program(s)". Following the perspective of computers as media [5, 10, 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 Semiotic Engineering theory [5].

In the last decade the importance of Human-Centered Computing (HCC) [2, 4, 12] has been growing steadily. This area aims to research design, development and deployment of several initiatives which involve the interaction between people and computers. Therefore, it covers a set of methodologies applied to any situation where people directly interact with computational artifacts [12].

In the case addressed by this paper, we observe that the source code is acting as an interface [18], mediating the communication between programmers, and, such as a traditional interface which we should appropriate from to make a better use, we believe that the interface represented by the source code of a software also needs to be understood and appropriated by the programmers that aims, somehow, use it.

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 lead us to the following questions:

(1) Why and how do novice programmers choose to reuse source code?
(2) Do they understand their own source code when it was built reusing someone else's code?
(3) Considering the possibility of being communicating with someone through their systems, with whom do they believe they are communicating?

IHC 2018, October 22–26, 2018, Belém, Brazil

ACM ISBN 978-1-4503-6601-4/18/10…$15.00
https://doi.org/10.1145/3274192.3274209

(4) How do they interpret the message delivered through their program's source code to other users?

In this paper we present the results of two qualitative studies carried out to answer these questions. We look at how novice programmers reuse source code, how they interpret them and how they integrate them into their own source code. Finally, we discuss the results in view of the importance of the program's metacommunication comprehension 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 to the results we found.

2 THEORETICAL BACKGROUND

In this section we present the theoretical background about source code reuse and the role of source code examples, Semiotic Engineering and appropriation.

2.1 Source code reuse and the role of source code examples

While learning how to program, 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 [19]. Furthermore, examples are often used to show the importance of some concepts. According to Malan and Halland [16], students must comprehend the importance of the concept, otherwise, they will continue programming without applying this concept to their source code.

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 [9]. In addition, nowadays the Internet offers to programmers a wide range of contents that can be easily accessed, which may have source code examples that fit to programmers' intentions.

The use of examples is a continuous practice during programmers' professional lives. Neal [19] observes that programmers with several different levels of experience code by studying, reusing or revising software (or parts of software) written by others. To benefit from an example, programmers must to understand this source code and the concepts embodied by it [1]. However, this example provided to help the programmer to understand some concepts, is often reused without fully understanding about what it does [15].

According to Hoadley [11], software reuse may occur in three different ways: (1) as code invocation, that occurs when functions and procedures are reused; (2) as code cloning, that occurs when source code lines are copied from an example and they are changed to achieve a new goal; and (3) templates reuse, which occurs when learned patterns are applied to other situations. In addition, there are other kinds of classifications to software reuse. One of these is proposed by Sojer [22] who classifies source code reuse in two ways: (1) snippet reuse, and (2) component reuse. The first approach is equals to the code cloning approach. However, the author proposes two branches to this, which can occur by code scavenging, that is, the replication of several and continuous lines from a source code, or design scavenging, in other words, when a structure composed by a large block of source code is used as a framework. The second approach is about the reuse of components that were developed, tested and documented to this purpose, such as APIs. These definitions complement one another, as we can observe in Figure 1.

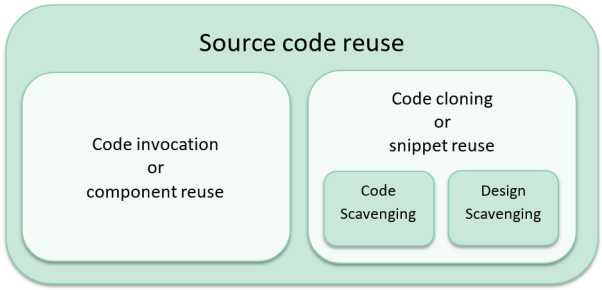

Figure 1: Source code reuse approaches

In this paper we are interested in the reuse of source code by cloning, and 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 own source code. We are also interested understanding the metacommunicative impacts of this reuse, based on the Semiotic Engineering theory, presented in the next section.

2.2 Semiotic Engineering

Semiotic Engineering [5] is a semiotic theory mainly based on Peirce's [20] and Eco's [8] theories. Its main study object is the metacommunication process between producers (designers or developers) and software consumers (users). According to the theory, metacommunication messages expressed by the interface must communicate how, when, where and why users should use this software. These several purposes must be linked to the design views made by designers and programmers who are the sender of this communicated message. This theory also offers an abstract model (or template) about the content of this metacommunication, which can be used as a support artifact when designing metacommunication or when evaluating it:

"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 during consumers' interaction with the software's interface. The interface represents producers during the interaction 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, changing it from the HCI field to the HCC field (where questions related to human interpretation

and communication cover human processes, even if these subjects are not final users, such as programmers, software architects, system analysts, etc.) [12]. We studied how the metacommunication process happens among programmers through the software source code. Thus, there are two adaptations: the first refers to those who receive the metacommunication (from now on, programmers instead of final - and lay - users); and the second one refers to the interface (from now on, a piece of code, with its textual facet and its executable facet, instead of the interface module to final users).

2.3 Appropriation

From the sociocultural perspective, appropriation is defined as the process of taking something that belongs to others and making it one's own [24]. From the technological perspective, appropriation is defined as how users evaluate and adopt, adapt and integrate a technology to their daily practices [3]. 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 [7], appropriation is similar to customization, though, it refers to the adoption of technology standards and its transformation in a deeper level. Appropriation involves customization (which means the explicit reconfiguration of a technology to make it fits to a specific need), but also may only involve making use of a technology to a different purpose from which it was developed to attend.

In a similar way that a technology is capable of shaping users' practices, it is also shaped by the users. Carroll et al. [3] defined a Model of Technology Appropriation, composed by three levels: the first level starts in 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 to their practices and it is considered stabilized.

Within this scope, software source code are technologies and, by this reason, users need to appropriate from them to make a better use. In this paper we take the source code of a 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.

3 RESEARCH DESIGN

In order to investigate how novice programmers reuse a source code from others and if this reuse affect their understanding about the software they built, we conducted two qualitative studies, detailed as follows [1]. These studies are part of a larger ongoing research which aims to support novice programmers during the reuse of source code.

[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 [17] and they are not relevant to the goals of this paper

3.1 Study One

3.1.1 Context and Goal. The study one 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 of 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 appropriation process happens when programmers reuse code.

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

3.1.2 Study procedure. This study was conducted in two steps, described as follows:

(1) 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 similar to the example, and the two novice programmers who produced source code that were the least similar to the example.
(2) 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[3]; (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.

3.1.3 Participants' profile. The profile of the participants we interviewed is presented in Table 1.

[2]https://theory.stanford.edu/ aiken/moss/
[3]Due to the size of the entire source code (between 585 and 4100 lines), we have selected some representative sections of each.

Table 1: Study One interviewees' profile

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

3.2 Study Two

3.2.1 Context and Goal. The study was conducted during 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.

The teacher shows as example a source code that calculate and present the initial 20 terms from the Fibonacci sequence. This example included a screen prototype responsible for presenting the result to the user. Few classes later, she asks the students to develop a program which 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.

3.2.2 Study procedure. This study was conducted in two steps, described as follows[4]:

(1) 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 JPlag[5] 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.
(2) Interview with the six students who accepted the invitation: during this interview we asked the participants to (a) explain their produced source code[6]; (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.

3.2.3 Participants' profile. In this study the students were not learning programming to make a career out of it. They were learning how to program in order to support their daily problems.

About whose that participated from the interview, their profile is presented in Table 2.

[4]All participants agreed to participate off both research steps and they signed the Informed Consent Form.
[5]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/), that, such as Moss, calculates metrics of text similarities.
[6]The source code produced by them had less than 20 lines of code each.

Table 2: Study Two 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%

4 FINDINGS

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

4.1 Why and how novice programmers reuse a source code

Related to how novice programmers search for source code examples, we found from Study One that they often search for examples that the domain is similar to that of the application they are building. If they 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."*[7]. He also mentioned that this kind of examples can 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"*.

Study Two 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.

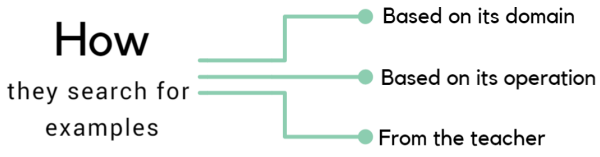

Figure 2: How novice programmers search for examples

Related to why they need examples (Figure 3), S1P3 mentioned she uses examples to *"understand the problem's logic. If is a question regarding the programming language, I will search for examples which represent the situation, apart from the subject. If is a question regarding the problem's logic, I try to locate examples which 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"*.

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

Some participants from Study Two 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 may 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 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"*.

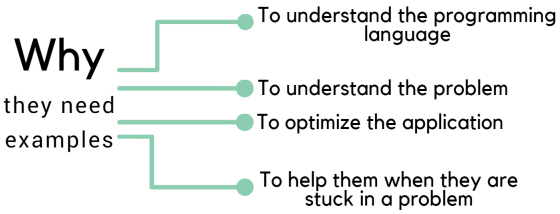

Figure 3: Why novice programmers need examples

Regarding to the ways they use the examples, they can be used as a framework, that might be modified and improved in order to achieve an specific goal. As presented before, this strategy is named *design scavenging*. During Study One, 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 may change his approach: *"Small source code, which require few changes, I reuse them, changing what is necessary. To more complex source code, that are 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.

Study Two has shown that novice programmers often reuse source code by cloning them to their own source code. With regard to 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 the 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 Study One. 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 Study Two. About it, S2P3 reported: *"it is easier to me to use the source code as a reference, otherwise I let something pass, 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"*. Figure 4 presents a summary of the ways novice programmers reuse source code.

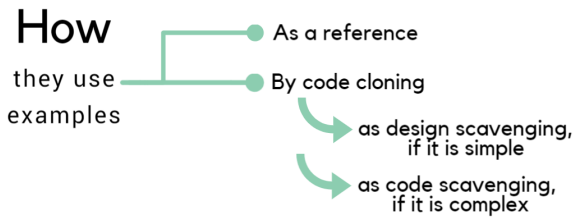

Figure 4: How novice programmer use examples

Additionally, during Study Two, we observed a programming approach, not related to reuse, but that corroborates with the previous observations which show that 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 about it, S2P6 mentioned *"I develop in a way I think it will works."*.

4.2 How they understand their own source code

During the studies One and Two, we asked the participants to explain their source code. Regarding Study One, we had two participants who performed a high reuse from 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 had also a program that was very similar to the Bookstore project, gave a totally 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 of extending it to add extra features.

About Study Two participants, from whose had source code similar to the source code example (S2P1, S2P2, S2P3 and S2P4), only S2P3 did not show traces of comprehension, not being 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.

4.3 With whom they believe they are communicating

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 Study One, 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 Study Two, some participant 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 other, 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 happens 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 difficult to 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 use informal sentences, similar to a communication with friends. S1P3 reported as characteristics 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.

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

4.4 How they interpret the message delivered through their program's source code

In the end of Study One, we invited the participants to fulfill the metacommunication templates from Semiotic Engineering theory. We split the template in 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 was seeking for new tools to didactical 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 gamers 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 refers to nonexistent features from the system: *"to register games and gamers, 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 an 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 (who 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 a fully original program.

With respect to how the users can fulfill these systems purposes, participants S1P2 claims that the system need 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 simply by following the menus (S1P3 and S1P4). Regarding to 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 few 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 about the message his application was delivering, and, he did not have full understanding about how his own source code works.

About S1P1 messages, by taking the point of view from Semiotic Engineering theory, we observe that the designer's metacommunication message delivered by the system's interface to its users is composed by two messages: the one from the Bookstore project designer and the one from S1P1. However, despite these messages complement each other, they are disconnected, once S1P1 did not properly appropriated from the message used as 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 about how his own systems works.

It draws our attention the fact that, when comparing the metacommunication messages from all Study One'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 the scratch. Differently from the S1P1 case, the message delivered by S1P2's system is also composed by the same two messages, and in this case, the messages are connected to each other.

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, we found that novice programmers often consider the possibility being communicating with other programmers through the source code in a scenario where their source code are being used as an example. Besides that, we presented findings about the impacts that source code reuse might have on the comprehension and the metacommunication that programmers have about their own source code.

4.5 Discussion

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 the 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 Study One, despite both S1P1 and S1P2 had widely used the example, they showed different interpretations 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 important details about their metacommunication.

The results we found introduced a reflection about the differences presented by S1P1 and S1P2 during the Study One. Both participants fulfilled their goals, by building a functional program that executed the required tasks. However, S1P2 showed a more precisely 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, he 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 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 appropriation of this code, depend on the level of understanding the user has.

About levels of understanding, we can observe 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* [11]
- 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* [11]. 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 to programming. The level can present two sublevels:
  - Without referring users' intentions: it means that the programmer can identify message passed through a 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 3, we presented a set of elements to support the classification of understanding and appropriation levels, as described before. These classifications 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, 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 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 Study One, 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 Study Two'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. 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 in 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. In addition, it is important 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 Study One and Study Two that the reuse of an example can affect the comprehension programmers have about their own source code, being able to observe cases in which they were not able to explain how source code written by themselves work. 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.

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 understanding the problem or can help them to optimize their source code. In addition, we observe that some of these programmers are aware that in some

Table 3: 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	Without users' intentions	
Low	X					No appropriation
Intermediate I		X				
Intermediate II			X			Lower
Advanced I				X		Higher
Advanced II					X	

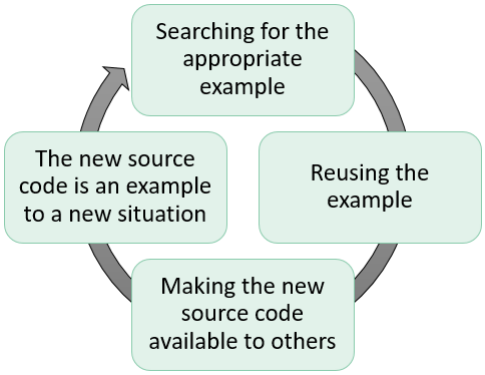

Figure 5: Examples reuse cycle

situations the user of their source code will be another programmer, who will use it as an example, and who will start a new cycle of interpretation and comprehension of this source code (Figure 5).

5 CONCLUSIONS

Programmers use programming not only to solve problems, but also to express something to consumers. This communicative process is an uninterrupted 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 of 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. 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 [6]. By 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 which 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 taking into consideration the time the students need to reflect about what they are doing. The process of reflection about these materials (source code) is a necessary step to solve a problem. Schön's [21] 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 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 about its role in their solutions.

Nonetheless, we would like to mention the work from Hoadley et al. [11] 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 of 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 the fact they do not know how to do it. Therefore, to support these students while the reusing activity we need to teach them how to perform meaningful summarizations and provide tools and methods than 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 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 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, specially 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 [5], to support student 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 need 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 that a way to solve problems, but also a tool through which programmers can communicate with each other and express themselves.

ACKNOWLEDGMENTS

We would like to thank all the participants, for the time provided to 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).

REFERENCES

[1] Eran Avidan and Dror G. Feitelson. 2017. Effects of Variable Names on Comprehension: An Empirical Study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 55–65. https://doi.org/10.1109/ICPC.2017.27

[2] Liam Bannon. 2011. Reimagining HCI: Toward a More Human-centered Perspective. *interactions* 18, 4 (July 2011), 50–57. https://doi.org/10.1145/1978822.1978833

[3] Jennie Carrol, Steve Howard, Jane Peck, and John Murphy. 2002. A field study of perceptions and use of mobile telephones by 16 to 22 years old. *Journal of Information Technology Theory and Application* 4, 2 (2002), 49–61.

[4] Sangil Choi. 2016. Understanding people with human activities and social interactions for human-centered computing. *Human-centric Computing and Information Sciences* 6, 1 (05 Jul 2016), 9. https://doi.org/10.1186/s13673-016-0066-1

[5] Clarisse Sieckenius de Souza. 2005. *The Semiotic Engineering of Human-Computer Interaction (Acting with Technology)*. The MIT Press.

[6] Clarisse Sieckenius de Souza, Renato F. de G. Cerqueira, Luiz Marques Afonso, Rafael R. de M. Brandão, and Juliana S. J. Ferreira. 2016. *Software Developers As Users: Semiotic Investigations in Human-Centered Software Development* (1st ed.). Springer Publishing Company, Incorporated.

[7] Paul Dourish. 2003. The Appropriation of Interactive Technologies: Some Lessons from Placeless Documents. *Computer Supported Cooperative Work (CSCW)* 12, 4 (01 Dec 2003), 465–490. https://doi.org/10.1023/A:1026149119426

[8] Umberto Eco. 1976. *A Theory of Semiotics*. Indiana University Press. https://books.google.com.br/books?id=BoXO4ItsuaMC

[9] Alessio Gaspar and Sarah Langevin. 2007. Restoring "Coding with Intention" in Introductory Programming Courses. In *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education (SIGITE '07)*. ACM, New York, NY, USA, 91–98. https://doi.org/10.1145/1324302.1324323

[10] Alexandra Georgakopoulou. 2011. *Pragmatics in Practice*. John Benjamins Publishing, 326.

[11] Christopher M. Hoadley, Marcia C. Linn, Lydia M. Mann, and Michael J. Clancy. 1996. *When and why do novice programmers reuse code?* Ablex Publishing Company, 109–130.

[12] Alejandro Jaimes, Daniel Gatica-Perez, Thomas S. Huang, and Nicu Sebe. 2007. Guest Editors' Introduction: Human-Centered Computing–Toward a Human Revolution. *Computer* 40 (05 2007), 30–34. https://doi.org/10.1109/MC.2007.169

[13] John Kammersgaard. 1988. Four Different Perspectives on Human-computer Interaction. *Int. J. Man-Mach. Stud.* 28, 4 (April 1988), 343–362. https://doi.org/10.1016/S0020-7373(88)80017-8

[14] Caitlin Kelleher and Randy Pausch. 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Comput. Surv.* 37, 2 (June 2005), 83–137. https://doi.org/10.1145/1089733.1089734

[15] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *ACM Trans. Softw. Eng. Methodol.* 23, 4, Article 31 (Sept. 2014), 37 pages. https://doi.org/10.1145/2622669

[16] Katherine Malan and Ken Halland. 2004. 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 (OOPSLA '04)*. ACM, New York, NY, USA, 83–87. https://doi.org/10.1145/1028664.1028702

[17] Luana Müller, Milene Selbach Silveira, and Clarisse Sieckenius de Souza. 2015. 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 (IHC '15)*. ACM, New York, NY, USA, Article 30, 10 pages. https://doi.org/10.1145/3148456.3148486

[18] Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (July 2016), 44–52. https://doi.org/10.1109/MC.2016.200

[19] Lisa R. Neal. 1989. A System for Example-based Programming. *SIGCHI Bull.* 20, SI (March 1989), 63–68. https://doi.org/10.1145/67450.67464

[20] Charles S. Peirce, Charles Hartshorne, and Paul Weiss. 1932. *Collected Papers of Charles Sanders Peirce*. Belknap Press of Harvard University Press. https://books.google.com.br/books?id=u9fWAAAAMAAJ

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

[22] Manuel Sojer. 2010. *Reusing Open Source Code: Value Creation and Value Appropriation Perspectives on Knowledge Reuse*. Gabler Verlag. https://books.google.com.br/books?id=-z60hspDTlAC

[23] Gunnar Stevens, Volkmar Pipek, and Volker Wulf. 2009. *Appropriation Infrastructure: Supporting the Design of Usages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 50–69. https://doi.org/10.1007/978-3-642-00427-8_4

[24] James V. Wertsch. 1998. *Mind as Action*. Oxford University Press. https://books.google.com.br/books?id=73Vv7Y3vf14C

[25] Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (March 2006), 33–35. https://doi.org/10.1145/1118178.1118215