**SHARING DOMAIN
KNOWLEDGE AMONG
PROGRAMMERS USING
THE SOURCE CODE AS
THE MEDIUM**

ALVARO VIDELA

Meaning and Context in Computer Programs

When you look at a function program's source code, how do you know what it means—that is, what object or process is this function representing? Is the meaning found in the return values of the function, or is it located inside the function body? What about the function name?

Answering these questions is important to understanding how to share domain knowledge among programmers using the source code as the medium. Whether debugging or adding new features to a program, programmers must read the code to understand what the program is doing. From this reading, the programmers must also know how the problem domain is represented in the code, so they can be certain that their changes to the source code won't make the program work in unexpected ways.

Programming tends to happen in teams, and a programmer might be added to a team during an ongoing

project. The new programmer is expected to understand what the program is doing by reading its code. In the case of a programmer working alone, when coming back to the source code after some time away, that programmer must understand what they wrote in the past. In any case, the program is the medium of communication among programmers to share their solutions.

“Programs must be written for people to read, and only incidentally for machines to execute,” as the adage goes,¹ so where in the program is this shared meaning expected to be found?

FUNCTION AND METHOD NAMES

Suppose we have a class called `Animal`, which has a method called `getName`. The class is used in an iteration as in this `forEach` loop:

```
class Animal
    public getName()
    {
        return name;
    }
}

animals = loadFromDatabase();
forEach(animal in animals) {
    print(animal.getName());
}
```

This method could return either the *pet's* name (for example, `toto`) or the *species* name (for example, `cat`).

Running this program would produce such output as `horse`, `cat`, and `dog`, which would help you deduce that the method is returning the species name. (You could argue that a better name for the method is `getSpeciesName()`, which is a valid criticism.)

The method `getName` presents an interpretative ambiguity that is resolved by looking at the method's return values. Still, you cannot be sure you are on the right footing, since you don't know how many cases are required to make the right deduction. Is there another device in code that could help with the meaning? To answer that question, pretend you keep browsing this project's code until you are surprised to find a different file that has the exact same code inside as the `Animal` file. Is this a duplication mistake, or is something else happening here? After executing the newly found code a couple of times, you see that it returns values such as `unicorn` or `mermaid`—fictional animals. Inspecting the folder structure of both files, you find that the last one lives in the folder `lib/animals/fictional`, while the first class lives in `lib/animal/non-fictional`. Thus, a piece of context external to the source code—a *paratext*—is helping with interpretation.³

A paratext is a piece of information such as a book title, chapter title, or preface that indicates how to interpret a text, so even if you are tempted to think that *Don Quixote* could be a historical account, the fact that it is described as a “novel” in the book's front matter tells the reader that it cannot be taken as factual.

In the case of source code, while the return values of `unicorn` and `mermaid` could help you deduce the meaning of the function, it is the folder structure that can provide

a stronger piece of information about what the method means—for example, revealing that the project is working with two kinds of animals, fictional and nonfictional. [Programming languages such as Java and C# support this feature at the language level, called packages or namespaces, respectively.]

RETURN VALUES

Can a function's return value be trusted as an indicator of what a function does, and thus its possible meaning? Consider the following function:

```
function square(x) {  
    // implementation goes here.  
}  
square(5) // returns 25
```

[A double forward slash indicates a code comment—that is, source code that is going to be ignored by the compiler and thus not executed.]

The function called `square`, when provided with an integer `x`, raises it to the power of 2. In this example, provided with 5, the function returns 25. So far, so good. What would happen if an inspection of the function's source code found the following?

```
function square(x) {  
    return 25;  
}
```

The function returns a hardcoded 25, so if you passed 5 or -5, the function would work as expected. Function return values can be seen as indexical signs—that is, they indicate that the function was executed, but they can't tell you anything about whether the function is working as expected. It's like seeing footprints in the snow. You can assume a hare just crossed the forest, but you cannot be 100 percent sure about it based on the footprints alone, since they could have been planted by someone wanting to prank you (see Umberto Eco's writing on indexical signs²).

Likewise, a return value is not enough to deduce that a function works correctly—even though the function's return values can be compared with known correct values that fall inside of what you might understand as the function's meaning. In the case of `square`, you could write a test program that checks that if provided the value 3, the function returns 9, provided -5, it returns 25, and so on. This becomes tautological: Since you assume the function `square` implements mathematical exponentiation to the power of 2, you then assume that 9 and 25 are correct return values.

As in the previous example of `Animal.getName()`, these return values indicate that `square(x)` is not a graphical function that draws on screen a square whose sides' lengths equal the provided `x` input value. Whether the function `square(x)` returns an integer or draws a square on screen, however, can be deduced from the type of the function's return value, assuming that information can be specified in the program's source code. Certain programming languages such as Haskell or Java allow

the programmer to provide type definitions for functions. Therefore, in the previous case, you could have written something like `square(x:Integer) -> Integer`, which means that the function `square` takes an integer parameter `x` and returns another integer—guiding your abduction or hypothesis that it returns `x` squared.

From return values you can then look at the actual body of the function, since that is where you can learn about the type of algorithm being implemented.

FUNCTION BODIES

The next task is to understand whether all return values are created equal. Say you have the function `random(start:Integer, end:Integer) -> Integer`—that is, the function returns a random integer that falls between the interval specified by the `start` and `end` input parameters. The problem is, you still don't know what kind of random numbers you are receiving from the function. From a security standpoint, when working with PRNGs (pseudo-random number generators), you might want to know the algorithm used to generate them, as some are more or less secure for cryptographic applications. This information is found in the function body where you see what type of algorithm is implemented inside the function. So, if you have two `random` functions, with different implementations, that once executed both happen to return the number 7, then you might want to know if these numbers come from a series such as the one generated by the Linear Congruential Method proposed by Donald Knuth,⁴ or one coming from the series proposed by Guy Steele.⁵

A similar example can be seen in a function called `sort(List[Integer]) -> List[integer]`, which takes a list of values of type `Integer` and returns a list of sorted `Integers` (whether in ascending or descending order is not important now). To know what kind of sort algorithm was used, you would have to look at the function's source code, where you might find an implementation of the quicksort or insertion sort algorithms, just to name a couple of examples.

Meaning in a program lives not only in various parts of the program's source code—the function names, the function parameters, the function body—but also in the package name that contains the function, as well as the various tokens of the function type received with each function execution, the return values. Each random number is a token revealing what kind of series the random algorithm is generating.

CONCLUSION

What can programmers do with this information?

Understand that the code doesn't "speak on its own," but there are various locations both inside and outside the code that guide the interpretation. This short article cannot provide a guide on how to write code that's easier to read or understand, but it can recommend that paying attention to each of the sections or aspects of the code mentioned here can guide the decisions to be made about using the source code to transmit information about the problem domain. This provides future developers approaching the code with many handrails to guide them as they interpret the code. They won't find just some

random words representing a model, but also the context in which those words make sense.

A future article could explore the relationship between the words used in names inside programs—function names, variable names, type names, and so on—and explain how they are used to build a sort of lexicon or DSL (domain-specific language) that represents some process of the real world—much the way a supermarket inventory works. This could help in understanding what kinds of competencies programmers need in order to understand what a program does. This article limited its exploration to seeing where that information could live in a program, but not how it is produced or used from a semantic point of view.

References

1. Abelson, H., Sussman, G. J., with Sussman, J. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press.
2. Eco, U. 1979. *A Theory of Semiotics*. Bloomington, IN: Indiana University Press.
3. Genette, G., 2001. *Paratexts: Thresholds of Interpretation*. Cambridge, England: Cambridge University Press.
4. Knuth, D. 2011. *The Art of Computer Programming, Volume 2*. Boston, MA: Addison-Wesley Professional.
5. Steele, G., Lea, D., Flood, C. H. 2014. Fast splittable pseudorandom number generators. In *Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages and Applications*, 453-472; <https://dl.acm.org/doi/abs/10.1145/2660193.2660195>.

Alvaro Videla is a developer advocate at Microsoft and organizes *DuraznoConf*. He is the coauthor of *RabbitMQ in Action* and has written for ACM. He is on Twitter as *@old_sound*.

Copyright © 2021 held by owner/author. Publication rights licensed to ACM.