Lector in Codigo or The Role of the Reader

Alvaro Videla
Durazno, Uruguay
videlalvaro@gmail.com

ABSTRACT

In this article I want to explore the relation between the process of writing computer programs with that of writing literary works of fiction. I want to see what parallels can we trace from the ideas presented by Umberto Eco in *Lector in Fabula* and *Six Walks in the Fictional Woods*, with the way we write programs today.

The goal of this article is to ask–and try to answer–the following questions: what can we learn as programmers from literary theory? What ideas can we incorporate from that discipline into our day to day programming activities, so we write code that's easier to understand for other humans (or our future selves)?

CCS CONCEPTS

• **Software and its engineering → Abstraction, modeling and modularity**; *Software design tradeoffs*; *Programming teams*; • **General and reference** → *General conference proceedings*;

KEYWORDS

metaphors, abstraction, literature, model reader, model author, encyclopedia, literate programming

ACM Reference Format:
Alvaro Videla. 2018. Lector in Codigo or The Role of the Reader. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (*Author version *<Programming'18> Companion).* ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/3191697.3214326

1 PROGRAMMING AS A HUMANS-FIRST ACTIVITY

The need for computer programs that are easier to understand for humans is not a new one. Since the early '50s as programmers started to distance themselves from machine code and began to program in higher level languages, we began to see a need to write code that appeals not only to machines but to humans as well. When David Wheeler [**?**] introduced the idea of the subroutine to the programming world, he said that:

> sub-routines are very useful–although not absolutely necessary–and that the prime objectives to be born in mind when constructing them are simplicity of use, correctness of codes and accuracy of description.

So we see that *subroutines* are not necessary for a program to work, but they are a device that can help with program understanding among other things.

Almost a decade later an anonymous article was published in the Datamation magazine called *What A Programmer Does* [**?**], which is surprising for how relatable those ideas are today. Its author is also concerned with the way humans communicate knowledge to each other, in the forms of programs. The money quote is this one:

> A programmer does not primarily write code; rather, he primarily writes to another programmer about his problem solution.

Later they reaffirm their point by saying:

> Both the value and quality of a programmer's work improve directly with the importance he places on communicating his program to a human, rather than merely to the machine.

We could say that this sentiment is finally echoed in its most popular form in Structure and Interpretation of Computer Programs (SICP) [**?**] when their authors say:

> Programs must be written for people to read, and only incidentally for machines to execute.

That is a strong statement, elevating humans as the main recipients of computer programs. Computers becoming mere tools that are able to run those programs at a speed that's useful for our practical purposes.

These ideas, presented by programmers for programmers, contrast with the ways programs are being seen by the industry and by the public in general. That is to say, we program because we expect results from the programs we produce. Whether it's accounting, or finding the route to hour hotel in a foreign city, most consumers of programs use them because of the results they produce. In fact, we could argue that a user will be quite displeased if they find themselves lost after following the routes offered by their GPS, even if customer support arguments that the code for their GPS software looks really pretty. We could say that this practicality has placed more emphasis in the results produced by programs than in how well they read for other people. When the goal is to meet a client's deadline, then the results produced by a program, it's *visible output*, is what matters (unless in cases where we are selling the software itself, where the end consumers will be other programmers).

This last point shows us that if we want to advocate for writing programs that other programmers can understand, then we need to understand that we need to accommodate these ideas in the software industry, an industry where the results produced by the programs is in constant tension with the quality in which those programs are produced, usually due to budget requirements. If we want to advocate for programs that are easier to understand, we need not forget that while programmers might want that as well, there are other forces at play that may compel programmers to declare their programs to be ready for delivery as soon as they produce the expected outputs, rather than investing time in tasks like refactoring. I said *expected outputs* because talking about *correct outputs* would be the topic of a complete different discussion. Having said that, the rest of this article assumes that we share the preoccupation of the authors of *SICP*, of the anonymous author

Author version
<Programming'18> Companion, April 9–12, 2018, Nice, France
2018. ACM ISBN 978-1-4503-5513-1/18/04...$15.00
https://doi.org/10.1145/3191697.3214326

of *What A Programmer Does*, and that of David Wheeler, among others. Who else was interested in these ideas? Donald Knuth.

2 LITERATE PROGRAMMING

In Literate Programming Donald Knuth [**?**] wrote about the attitude we should have when we write programs: "instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do". He goes on in his paper explaining the WEB system for writing software documentation, which we could say has been partially adopted by the industry in tools like Java Docs and the like.

So while we have systems that can help us write documentation, we still have the problem of how to effectively transfer our tacit knowledge about the system to other programmers. Literate Programming is a good step towards placing emphasis on documentation and communication, but still doesn't explain how we could accomplish that goal. It's interesting at least that he points at literature as a place where we could learn techniques for writing programs that are easier to understand for other humans.

Here we need to address a difference between writing (as in literature), and programming that's been raised by papers like *Programming is Writing is Programming* by Herman and Aldewereld [**?**]. There they say:

> a difference between writing and programming, [is that] in programming, the programmer gets feedback very early on whether the program text is executable, during compiling. Furthermore, they get feedback on whether the program is working as intended.

One could argue that while this is true, knowing that a program is working as intended doesn't indicate that we know what the program actually means, that is, what process from the real world is trying to represent, what problem is trying to solve, neither is telling us how said problem has been solved.

We could compare this with playing music on a guitar by reading the notes as they appear on a staff. In the guitar an `E5` could be played on the 12th fret on the 6th string, on the 7th fret on the 5th string and on the 2nd fret on the 4th string. Since all these finger positions produce the same sound, we could say the music was executed as written in the staff, but this is not sufficient to consider it correct playing. As Carlevaro explains in his *Serie Didáctica para Guitarra* [**?**]:

> Correct guitar playing is unconceivable without correct fingering.

So it's not only is important the result produced by a function, or the musical note emitted by an instrument, but also it's important how these results were produced.

As Knuth writes in The Art of Computer Programming [**?**], is a `random` function that returns the number `2` as result, a valid implementation? Equally we could ask: is a `square` function that returns `25` a correct implementation? As long as the input values for the `square` function happen to be `5` or `-5`, then the implementation appears to be correct.

While this example might sound like a stretch, TDD [**?**] advocates to start writing stub functions that behave exactly like this, providing the minimum implementation that could yield a correct result for a particular test. This kinds of testing presents the problem of needing to submit a program to every kind of possible input to show that it works as expected. In the words of Dijkstra [**?**]:

> Program testing can be used to show the presence of bugs, but never to show their absence!.

Tools like QuickCheck [**?**] try to alleviate this problem by generating random input for tests, while at the same time presenting techniques like *Property Based Testing*. So while we are creating better tools to prove the correctness of programs, we are still left to our own means when it comes to understanding what a program is doing.

Does this means we should consider every program as a *work of literature* like Knuth says in his paper? In Cybertext: Perspectives on Ergodic Literature, Espen J. Aarseth presents a different point of view [**?**]:

> [...] a search for literary value in texts that are neither intended nor structured as literature will only obscure the unique aspects of these texts and transform a formal investigation into an apologetic crusade.

I concur with Aarseth. Trying to find literary value in programs seems to be a task destined to fail, because programs aren't written with a literary goal. That doesn't mean there isn't any value in thinking of programs as literature. I think there is, but in a different way. We should look at literature and learn from the techniques authors use when writing fictional works. This means we should go to the Humanities side of the building and learn from Literature Theory, Linguistics and Semiotics and incorporate some of their techniques into our day to day activities as programmers.

3 KNOWLEDGE SHARING

Before we delve into the various ideas from Literary Theory that could help us improve the way we write our programs to we share knowledge between programmers, we need to ask ourselves *what* kind of knowledge are we trying to share. What is this *meaning* that programs should convey and that me as a programmer reading a certain piece of code should be able to understand.

In *Programming as Theory Building* Peter Naur [**?**] draws our attention towards the idea of a *Theory* based on Gilbert Ryle's *The Concept of Mind* [**?**]. Naur explains the knowledge a software engineer builds about a system in the following terms:

> [...] a person who has or possesses a theory in this sense knows how to do certain things and in addition can support the actual doing with explanations, justifications, and answers to queries, about the activity of concern.

and he continues:

> [...]what has to be built by the programmer is a theory of how certain affairs of the world will be handled by, or supported by, a computer program.

So from the moment we do requirements gathering, to how we implement those requirements in code, we are building a theory that tacitly includes the different tradeoffs we took along the way. For example: why are we using `Long` instead of `Integer` for the numbers we use in certain class; why the time resolutions is in nanoseconds and not milliseconds; why we added a caching layer to avoid network roundtrips; why we decided to implement our own library for problem X instead of using what was available on the market; why our key value data structure is backed by a tree instead of an array; why we decided to not add a layer of abstraction between the database and its query language; and so on. Of course the list is not exhaustive. Some answers to those questions might be clearly obtained by reading the code, like for example: let's say there's just one SQL query in the whole project. So we could argue there was no need for a library that abstracted the database away.

In this article we are interested about those instances where knowledge cannot be immediately obtained from the source code, its tests and documentation, or any of the other artifacts produced by the code. There is where we want to go and look for help in Literary Theory, Linguistics, and related disciplines, so we see how we could improve the way we write software.

When we try to read code written by other programmers or by a past version of ourselves, we are presented with the task of reverse engineering that code. We start by trying to understand what the original programmers tried to solve with that program. It doesn't matter if this is legacy code or if we just landed on a team and we are handed with the next issue on the bug tracker. We will be faced with some code and we will have to read it and understand it so we can proceed with the confidence that we are modifying it in the right way, or adding the new feature in the right place, so we solve the problem at hand.

The problem with this approach is that it places the burden of understanding the code in the reader. The reader has to be able to decipher the code, but the difference with cryptography is that we are not adversaries, but members of the same team, so understanding code shouldn't be a challenge, but a matter of collaboration.

Imagine if every time we tried to read a book, we had to play code breakers? Unless we were reading the Finnegans Wake, I'm sure that wouldn't be a enjoyable reading experience. In the case of programming, unless we have are in the business of reverse engineering code, then we shouldn't have the need to play Finnegans Wake every time we are faced for the first time with a program.

So what can we do as programmers to help others understand our code?

4 THE ENCYCLOPEDIA

In *Lector in Fabula* [?] and *Six Walks in the Fictional Woods* [?] Eco introduces the idea of the *Encyclopedia*, which comprises all the knowledge in the world. It's safe to assume that no human posses that knowledge. We have our own limited *encyclopedias* (in lowercase). Whenever we try to interpret a text we bring our own encyclopedia into the game and we rebuild its contents, we actualize it according to our own competence. Consider this very short story:

> I sat in front of my computer and started coding.

When we read a text like in the previous example, we make a lot of choices when it comes to actualize that text. If we imagine the programmer typing their program on a laptop or a on a keyboard connected to a computer, it doesn't matter, unless the device used is consequential to the rest of the story. I'm used to write programs on a laptop, so most probably I will image that scene with a person typing their program on that kind of computer. What's the programmer's gender? Are their blond or have dark hair? For some readers that doesn't matter, so they don't even think about that, while other people might render them according to their own preferences or biases. The lesson to take home from this, is that *we fill in the blanks on a story using what's available on our own encyclopedia.* This opens a very interesting question of how we as authors manage to transfer our ideas from our minds to our communication recipients or *destinataries.* If we fail at that, the other person might end up building a different interpretation of what the code does.

A key idea from *Lector in Fabula* is Eco's criticism of communication theory, he says:

> [...] the competence of the recipient is not necessarily that of the sender.

and later he adds:

> Therefore, in order to "decode" a verbal message, in addition to linguistic competence, a differently circumstantial competence, an ability to trigger presuppositions, to repress idiosyncrasies, etc., and so on.

So how do we go about understanding the *target audience* for our programs? Eco has an answer for that as well, the *Model Reader*.

5 THE MODEL READER

The *Model Reader* is not the empirical reader (not you, or me). It's a reader that lives in the mind of the author, which the author builds as they write their story. This model reader will help the author decide how much detail is required in their work, so the empirical readers are able to understand it. Let's go back to our one sentence story:

> I sat in front of my computer and started coding.

In that text I've made some choices about how much info I wanted to convey based on my Model Reader; in this case I assumed that I don't need to explain that I'm using a keyboard to type the program, because in most cases, that's how it's done. In that one sentence story, my Model Reader is a person that's familiar with how computers work, and with how we type programs in a computer today.

To show how important the model reader is, let's imagine we see a sign at the London Underground system that says: *dogs must be carried on escalator.* In *Literary Theory: an Introduction* [?], Terry Eagleton, asks the following question about that simple sentence:

- Does it mean that you must carry a dog in the escalator?
- Are you going to be banned from the escalator unless you find a stray dog to carry?
- "Carried" is to be taken metaphorically and help dogs get through life?
- How do I know this is not a decoration?
- I need to understand that the sign has been placed there by some authority Conventions: I understand that "escalator" means this escalator and not some escalator in Paraguay.
- "Must be" means must be now.

So we see that such a simple sentence like "dogs must be carried on escalator", implies a Model Reader that understands that if they are using the London Underground system, and they bring a dog, the must carry that dog if they have to use an escalator. All that information, while not provided in the text, was implied when the designers of the sign chose their Model Reader.

So one aspect of the idea of the Model Reader revolves around how much information or context do we provide in the text, so the message we are trying to convey is understandable. Another aspect is that, as the story progresses the author also builds the Model Reader.

Now let's imagine my previous text went as follows:"I sat in front of my computer and started coding. The clouds outside my window had cleared, revealing the Mars landscape." Once the word *Mars* appears in the text, we can easily tell that what seemed to be a personal recollection (about me typing into a computer), turned out to be a science fiction story about space exploration. With that small clue (Mars) the author built a Model Reader that must accept that the story is fictional, since so far we don't have humans living in Mars.

In *Lector in Fabula*, Eco goes deeper into the idea of the Model Reader (and the Model Author), to present the concept of textual cooperation. There he says:

> A text is a lazy (or economic) mechanism that lives on the surplus value of meaning introduced by the recipient [...]

Reading is essentially a work of cooperation between the author and the reader.

To summarize, the author uses a Model Reader in order to decide how to tell the story, both in how much detail they need to provide, and on how to give clues to tell the reader this is a noir novel, or this is epic fantasy, or everything that was said so far *was part of a dream so be careful if you keep reading, since what comes next might still be part of a dream.* The Model Reader is someone that's willing to cooperate with the author. The Model Reader actualizes the text.

The question is how does the concept of the Model Reader can help us think about how we write programs? Who is our Model Reader? Is it the computer? Is it another human, or a future me?

6 A PROGRAMMER'S MODEL READER

As I wrote in the introduction to this article, a striking difference between writing programs and writing literature is that most of the time programs are written for computers to execute, not for humans to find the development of plots and storylines like one would expect from a work of fiction. As we already argued in this article, when we are writing software we are trying to satisfy some very practical needs, trying to address some human problem. So how can we integrate these two seemingly opposing points of view?

We can try to answer that question by pondering who is the Model Reader when we write programs? Who do we keep in mind while we type our code? The first answer that comes to mind is the compiler. We write code trying to follow all the syntactical rules of a specific programming language, while providing all the necessary clues for it to find the definitions of the functions and types used in the program. We can say that when we program "we play computer" in our heads. Since today almost nobody writes code for a specific computer architecture, we can say that we build a model computer in our minds, and then we try to second guess how it will run our code based on our assumptions about that ideal computer. So our Model Reader fluctuates between trying to satisfy a compiler and an imaginary computer.

7 A COOPERATION GAME

The idea of playing computer in our heads is interesting because it relates to Eco's idea of producing text as a game of strategy. To win in a game of strategy we build a model opponent and use that model to try to anticipate their moves, so we can put in place a strategy that help us win the game. Eco says that in a way at Waterloo, Wellington built a more accurate model of Napoleon.

Let's forget about fighting and let's think in terms of cooperation, let's think on how we build strategies that can help the recipients of the text actualize it. In the words of Eco:

> A text wants someone to help it work.

How can we help a computer/compiler to actualize a text, so it's able to bring the code to life, in the same way we do when we read fiction? Borrowing from literature we can say we do world building when we declare types, or when we define interfaces in header files. Later we tell the compiler where to find the headers that define the interfaces used in our programs. Whenever we use names, like variables, we usually have to declare them first. In strong-typed languages, we even assign types to those variables, which help us reason about those programs, but also give clues to the compiler about what to do whenever they find those datatypes further down in the program. So on one hand a variable name, or its type, appeals to the reader to access their encyclopedia to try to understand what that variable represents, on the other hand, when we define those variables, or specify their types, we are building said reader as the text is built.

Still, we keep seeing this duplicity of writing for the computer, and writing for a human reader. Again, Espen Aarseth raises an interesting point in his book about cybertext:

> Programs are normally written with two kinds of receivers in mind: the machines and other programmers. This gives rise to a double standard of aesthetics, often in conflict: efficiency and clarity.

This poses an interesting question, which can be related to Eco's idea of the different levels of readers.

8 DIFFERENT LEVELS OF READERS

In the essay *Intertextual Irony and Levels of Reading* [**?**], Umberto Eco writes that texts that have an aesthetic aim tend to construct several levels of Model Readers. In the first level, the reader just wishes to finish the story, to know what happens, to know how it ends. The second level reader is a semiotic or aesthetic reader, who wants to know how what happens has been narrated. To become a second level reader, one has to read a story many times.

I want to posit that we humans are the second level readers of our programs. We are the ones that want to know how what happens in the code has been narrated.

The question is then how do we manage to interpret a text, and add meaning to it, understanding its intended goal? And how we as authors/programmers can help others understand our code? Let's talk about metaphors.

9 METAPHORS

In *Metaphors We Compute By* [**?**] I wrote that code is a metaphor for a solution we found. A program is a translation into code of that problem solution. The explanatory power of that code, how well the solution is transmitted, is the measure of its elegance.

I will not repeat the whole argument presented in that article, but the gist of it is that if we have a collection of elements that need to be unique, a `Set` data structure has more explanatory power than an `Array`, even tho an `Array` would fit the job just fine, because the `Set` implies element uniqueness. A second point is the power metaphors have for obscuring and augmenting our knowledge. In *Epidemic algorithms for replicated database maintenance* Demers et al. [**?**] explain how *gossip* was a good metaphor for understanding the replication technique they were proposing, but it wasn't until they come to the realization that it mapped quite well to the idea of *epidemics* and how they spread in a population. This realization helped them wield all the mathematical theory behind the prevention of epidemics' spread.

As Peter Gärdenfors says in *Geometry of Meaning: Semantics Based on Conceptual Spaces* [**?**]:

> Metaphorical mappings preserve the the cognitive topology of the source domain in a way consistent with the inherent structure of the target domain. Metaphors transfer information from one conceptual domain to another. What is transferred is a pattern rather than domain specific information. A metaphor can thus be used to identify a structure in a domain that would not have been discovered otherwise.

So metaphors help with understanding, which means we could leverage their power when choosing the abstractions we use in our code. The understanding of who the Model Readers of our code are, will serve as guide when needing to choose the right metaphor. For example if we implement the exponentiation algorithm presented in section 1.2.4 of Structure and Interpretation of Computer Programs [**?**], should we do it by allowing our function to just accept `integers`, or should we go for an abstraction like `Monoids` so the code would work for other data types that provide an associative binary operation and identity, like `Strings` and concatenation, or a `Matrix` and exponentiation? (See *Elements of Programming* by [**?**] for a discussion on how to evolve the exponentiation algorithm for from Integers to Monoids).

What other tool could we bring from literature that could help us write programs that are easier to understand? Enter paratexts.

10 PARATEXTS

In literature there's the idea of the *Paratext*, which Eco quoting Gérard Genette defines as:

> [...] the "paratext" consists of the whole series of messages that accompany and help explain a given text–messages such as advertisements, jacket copy, title, subtitles, introduction, reviews, and so on.

Genette adds the following to his definition of paratext [**?**]:

> [a paratext is] a privileged place of a pragmatics and a strategy, of an influence on the public, an influence that–whether well or poorly understood and achieved–is at the service of a better reception for the text and a more pertinent reading of it.

An obvious instance of paratexts in literature could be found in old books, like Don Quixote, where chapter titles are followed by text like this:

> Chapter I. Which treats of the character and pursuits of the famous gentleman Don Quixote of La Mancha.

Do we have these kinds of paratext in code? Yes, as said earlier, we not only specify imports, we arrange the code in modules, packages and libraries. We specify flags for the compiler that can change the way the software is built (for example changing a library's location). In languages like Haskell we could add pragmas to the source code that could change the meaning of the program or even its efficiency. This means that as we write code, we are not only targeting our model computer, we are also targeting a specific compiler. We are trying to provide it with information so it's able to understand what it should do when it finds code stating that the variable `isActive` is of the type `AtomicBoolean`.

We also need to consider how paratexts helps us humans understand programs. When we read `utils` as part of a package name we build a different set of expectations about the contents of that package from when we read `network`, or `persistence`. A class name like `FileSystem` is telling us a lot of things about what to expect in its API. Finding a public method called `stringCompare` inside it will feel very strange. Code comments are probably one of the most important paratexts that can help other programmers understand our code. Keeping them in sync with the code is a whole different problem. Not even Cervantes escaped this fate: in Don Quixote, the original description for Chapter X doesn't match the contents of the chapter!

To understand how important paratexts are, consider what Genette says about them:

> To indicate what is at stake, we can ask one simple question as an example: limited to the text alone and without a guiding set of directions, how would we read Joyce's Ulysses if it were not entitled Ulysses?

Let's try to visualize this with a simple code example. Consider the following class representing a user:

```
class User {
    String username;
    String password;
    String role;

   User(String username, String password, String role) {
        this.username = username;
        this.password = password;
        this.role     = role;
    }

    public String getUsername() {return username;}
    public String getPassword() {return password;}
    public String getRole()     {return role;}
}
```

The code of this class doesn't offer enough information to tell us what is the class purpose. Now let's consider the following test case:

```
User user = new User('alice', 'secret', 'admin');
assertEquals(user.getUsername(), 'alice');
assertEquals(user.getPassword(), 'secret');
assertEquals(user.getRole(), 'admin');
```

The previous test can give us feedback about the code working as expected, but we are still in the dark about what is this class purpose, that is, what concept of the real world this class is trying to represent.

Now let's see the following class:

```
package database;

class User {
    String username;
    String password;
    String role;

   User(String username, String password, String role) {
        this.username = username;
        this.password = password;
        this.role     = role;
    }

    public String getUsername() {return username;}
    public String getPassword() {return password;}
    public String getRole()     {return role;}
}
```

The only difference with the previous code is the line saying `package database`.

The mere fact that we include a paratext telling us which package this class belongs to, tells us immediately that this class serves a different purpose from another `User` class that could appear inside the `model` package. The latter could represent a `User` in the system, say our social network website, which has different roles, like *paying member*, while the database `User` class could offer roles like `admin`, `app_read` or `app_write`, for example. In a language without packages, the same information could be conveyed via the project's folder structure.

On a funnier note, what would Magritte think of the following code?

```
// This is not a person
class Person {
    String name;
    String age;

    User(String name, String age) {
        this.name = name;
        this.age  = age;
    }

    public String getName() {return name;}
    public String getAge()  {return age;}
}
```

So while it's dubious that someone will think the previous class encompasses the whole definition of a person, how many of us have spent hours debugging a `FileSystem` class that doesn't live up to its API's promises, because ultimately we didn't realize that this class *is not* the File System? So that comment about the class not being a person, even if funny, can lead us to interesting semiotical questions about the realities represented by our code.

11 CONCLUSION

The last point brings us back to metaphors, maps, and ultimately how we see the world. Espen Aarseth writes:

> [...] paradigms such as object orientation [inspire] practical philosophies and provides hermeneutic models for organizing and understanding the world, both directly (through programed systems) and indirectly (through the worldviews of computer engineers).

Earlier I said that a program is a metaphor for a solution we found. A map between reality and the world of data structures and algorithms. Code is a kind of visualization for a particular problem. Visualizations are made with certain goals and certain audiences in mind. This is what Noah Iliinsky has to say about maps in the book *Beautiful Visualization* [**?**], when he describes the style of the London Tube's Map, a map that detached itself from a mere geographical representation of the underground lines:

> That freed the map of any attachment to accurate representation of geography and led to an abstracted visual style that more simply reflected the realities of subway travel: once you're in the system, what matters most is your logical relationship to the rest of the subway system.

Then Iliinsky writes about what makes for an effective visualization:

> The first area to consider is what knowledge you're trying to convey, what question you're trying to answer, or what story you're trying to tell.
> [...] the next consideration is how the visualization is going to be used. The readers and their needs, jargon, and biases must all be considered.
> The readers' specific knowledge needs may not be well understood initially, but this is still a critical factor to bear in mind during the design process.
> If you cannot, eventually, express your goal concisely in terms of your readers and their needs, you don't have a target to aim for and have no way to gauge your success.
> "Our goal is to provide a view of the London subway system that allows riders to easily determine routes between stations,"
> Understanding the goals of the visualization will allow you to effectively select which facets of the data to include and which are not useful or, worse, are distracting.

His points resonate with the ideas of choosing a Model Reader, where we ask ourselves what elements from our encyclopedia we need to share, and which ones could be readily actualized by the recipients of our texts. We also must write code with certain goals in mind, not only thinking about what our final users will be able to do with the product, but also thinking about what we will allow other programs to perform with the APIs we expose. How good is the *theory* we build about the problem we are solving, will determine the usability our maps–our code–will end up having. Because in the end, with abstractions–whether they are types, classes or interfaces–we are designing maps.

As William Kent says in *Data and Reality: a Timeless Perspective on Perceiving and Managing Information in Our Imprecise World* [**?**]:

> After a while it dawned on me that these are all just maps, being poor artificial approximations of some real underlying terrain.

and he adds:

> What is the territory really like? How can I describe it to you? Any description I give you is just another map.

The *theory* we built about our software, with its implicit decisions and tradeoffs, will be effectively shared by choosing the right *metaphors* that will resonate with our reader's *encyclopedias*. These metaphors will work like maps that guide understanding. We will also count with the help of *paratexts*. These paratexts will work like sign posts on a country road, so when the map is not good enough, our users will still be able to orient themselves. Because ultimately we need to understand what Borges so clearly illustrates in his short story *On Exactitude in Science* [**?**], and that is: **the map is not the territory**.