Lector in Codigo or The Role of the Reader

by Alvaro Videla

In this article I want to explore the relation between the process of writing computer programs with that of writing literary works of fiction. In particular I want to see what parallels can we trace from the literary theories 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 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)?

In this article first I will introduce some literary concepts that are key to these discussions, like those of Model Reader and the Encyclopedia, while trying to relate them to programming. Without further ado, let's enter the woods and lets try to navigate the garden of the forking paths.

The Model Reader

The first concept we have to understand is that of the Model Reader of a text. 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 use an example to illustrate this point.

If I write a text saying "I sat in front of my computer and started coding […]", I've made some choices about how much info I need 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 was a person that's familiar with how computers work, and with how you type programs in a computer today.

So one aspect of the idea of the Model Reader revolves around how much information or context do I provide as an author in my text, so the message I'm trying to convey is understandable.

At the same time, 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 appeared 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 the book Lector in Fabula, Eco goes deeper into the idea of the Model Reader (and the Model Author), to present the idea of textual cooperation. There he says[1]:

> Un testo è un meccanismo pigro (o economico) che vive sul plusvalore di senso introdottovi dal destinatario […]
>
> 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 what we have so far: we have an author that 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?

A Programmer's Model Reader

A striking difference between writing programs and writing works of fictions 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.

When we are writing software we are trying to satisfy some very practical needs, trying to solve human problems. Yes, sometimes we write code just for fun, but the point is that we expect from a program some very specific results. To give a very trivial example. If we write a power function, and provide it with 5^2 as input, we expect to receive 25 as result. That means we –usually– know what it means for a program to be correct: did the ABS system in the car kick in at the right time causing the car to stop, or was there an accident? This means that if what matters are the results, then there's no need to interpret the code, since we know what it should do.

With the former point of view, we should treat code as a black box that produces results. As long as the computer is able to run it, we are good. The problem with having only this view is that we miss the part where humans come into play. This quote from Structure and Interpretation of Computer Programs comes to mind:

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

The problem is that even if a person can enjoy the activity of reading source code, at the end of the day the validator of that program will be a computer executing the code. 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, we can say that we build a model computer in our minds, and then 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.

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 managed to build 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 destinatary of the text actualize it. In the words of Eco:

> Un testo vuole que qualcuno lo aiuti a funzionare.
>
> 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, and then we tell the compiler where to find the headers that define the interfaces used in our programs. Whenever we use names, like variables, we have to declare them first. In strong-typed languages, we even assign types to those variables, all clues that unless the language provides type-inference, are there to help the compiler, otherwise what's the point of writing something like `User user = new User();` other than to help the compiler?

In literature we have the idea of the "paratext", which can be anything from chapter titles, to intro text telling us that the book is based in some manuscript that got lost, or even chapter subtitles that tell us about what the following text should be about. In Don Quixote for example 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 this kind 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 and in languages like Haskell we can use pragmas. This means that as we write code, we are not only targeting our model computer, we are also targeting a compiler, and we are trying to provide it with information so it's able to understand what it should do when it finds code saying that the variable `isActive` is of the type `AtomicBoolean`.

While this explain who's our Model Reader when we write programs, this doesn't say much about how or why we should make programs that are for people to read, that is, programs that should be understood by humans.

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? To answer this question we need to introduce the idea of the encyclopedia.

The Encyclopedia

In Lector in Fabula and in 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. Whenever we try to interpret a text we bring our own encyclopedia into the game and we rebuild the story, we actualize it according to our own competence.

Whenever we read a text, like in the example I gave above, about a person typing a computer program, we can make many choices when we 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 in this 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:

> […] la competenza del destinatario non è necessariamente quella dell' emittente.
>
> […] the competence of the destinatary is not necessarily that of the sender.

and later he adds:

> Dunque per "decodificar" un messagio verbale occorre, oltre alla competenza linguistica, una competenza variamente circonstanziale, una capacità di far scattare presupposizioni, di reprimere idiosincrasie, eccetera eccetera.
>
> 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.

In Metaphors We Compute By, I wrote that we should use the right data structures to help others understand our code. When we choose to store our collection in a Set instead of an Array, we are telling the other person that items in this collection must be unique. Choosing the right metaphors can help us reason about our code and algorithms. In the distributed systems literature there's a paper called Epidemic Algorithms for Database Maintenance which introduced randomized algorithms for replicating distributed updates. Its authors explain the idea using the gossip metaphor, but when it was time to prove the results of their work they found out that the epidemic metaphors was better suited for reasoning mathematically about their ideas.

We also need to consider how paratexts helps us 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!

Another point to keep in mind is what Peter Naur calls *Programming as Theory Building* in the paper of the same name. He argues that to be able to write a program we must first build a theory about the problem we are trying to solve, say, a hardware store inventory. When a new person joins the team, how do we transfer that tacit knowledge, that theory, so they can understand the code? What happens when we become maintainers of a project whose original developers are long gone? How do we make sure all that implicit knowledge that the original programmers had about the problem is shared later through the code? What kind of texts do we need to create to successfully transfer tacit knowledge into explicit knowledge? In *Making history: writing the docs that need to be written* I argued that we should write as much documentation as possible, to help people understand our code. We need to pass beyond tacit knowledge. There's a reason why societies collect their lore in written form so it becomes history. Oral tradition gets lost.

Failed Expectations

Another thing to keep in mind when trying to understand code is how a small change in the context of execution can change completely the meaning of the code, or in any case how we should interpret it. What's the meaning of a code that instantiates a class like `DatabaseConnection` and then calls methods from its API when during a test run that class has been mocked or stubbed?

What about code that targets various operating systems, or different flavors of databases or file systems, and ends up implementing certain operations as noops, since there's no equivalent for them in the target system? How are we supposed to interpret and understand code that calls such operations? Maybe here paratexts comes to rescue, in the form of comments that can alert us of this particular implementation. If we read a comment that indirectly says "this operation is not supported in ___ OS, but we must implement it to satisfy the API", what are we supposed to understand from the rest of the code? This seems like trying to satisfy the paradox of a text that starts with "everything in this text is a lie".

Questions

Instead of a conclusion I want to leave some open questions for us.

What if instead of having a compiler/computer in our minds while we write code, we try to build a Model Reader that's closer to a fellow programmer? At the end of the day our chosen IDE will help us satisfy the compiler, and validation tests can help us verify that our assumptions about our model computer were correct. We are then left with the task of making sure another programmer will understand our code.

How does the idea of the Encyclopedia affects us when we write/read code? Eco says that we fill in the blanks using content from our encyclopedia, we added the keyboard to that character that sat in front of their computer and started coding. When someone presents us with a model for which we need to implement a program, how much do we fill in into that model taking from our own encyclopedia? Taking inspiration from Data & Reality by William Kent, how much does our mode and implementation of a hardware store inventory matches the model hardware store manager has of its inventory? How much did we fill in with our own encyclopedia when said manager tried to transfer us their model? This ultimately leads us to consider that the map is not the territory.

Notes:

1 — Translations by Google, adapted by myself.

Bibliography:

Lector in Fabula - Umberto Eco - https://www.amazon.com/Lector-Fabula-Italian-Umberto-Eco/dp/8845248062/ Six Walks in the Fictional Woods - Umberto Eco - https://www.amazon.com/Six-Walks-Fictional-Woods-Umberto/dp/0674810511/ On Literature - Umberto Eco - https://www.amazon.com/Literature-Umberto-Eco/dp/0156032392 Epidemic Algorithms for Replicated Database Maintenance - Alan Demers et al - https://dl.acm.org/citation.cfm?id=41841 Don Quixote - Miguel de Cervantes - http://www.rae.es/obras-academicas/ediciones-conmemorativas/el-quijote-edicion-de-2015 Data & Reality - William Kent - https://www.amazon.com/Data-Reality-Perspective-Perceiving-Information/dp/1935504215 Metaphors We Compute By - Alvaro Videla - https://queue.acm.org/detail.cfm?id=3127495 Making history: writing the docs that need to be written - Alvaro Videla - https://hackernoon.com/making-history-writing-the-docs-that-need-to-be-written-1396e27fae65