Documentation: a reproducibility issue

Elia Ravella

Politecnico di Milano

Contents

1 Documentation

What is documentation? "Documentation is any communicable material that is used to describe, explain or instruct regarding some attributes of an object, system or procedure, such as its parts, assembly, installation, maintenance and use"[5]. Documentation is a set of technical documents, employed in the development of software tools; it is not to be confused with bureaucratic documentation.
Why is documentation needed, and what role does documentation play in the software development world?

1.1 Software documentation

In the software world, especially in the software *development* world, documentation is a very broad term: it includes

- manuals: organized sources of information about the *use* of a specific piece of technology, as a tool or a framework whole;
- APIs, or Application Programming Interfaces: a collection of human readable descriptions of the behaviours of the single functions offered by a tool; these are the standard go-to documentation when a developer must understand and use a new library or language;
- code comments: when a programmer needs to delve deep into the details of a technology s/he is using, s/he probably reads its source code: documentation is also there, in the form of *comments on the code* which are "statements by programmers explaining their code to other programmers who may work on the same programs and to remind themselves of what they did or what remains to be done" [5].

These documents are fundamental to the description of a technology, and so they concur to the *general usability* of such technology: it is pretty straightforward that a well documented tool is far easier to work with with regard to a poorly described one.
Software documentation is the designated entry point for a software: while a mechanical tool, such as a screwdriver or a wrench, conveys its correct use easily enough through the geometry of the tool itself, the same is not valid for software tools as development frameworks, programming languages, development environments or even *approaches to a problem* when talking about software; the inherent complexity of developing a software renders very difficult to just *guess* how a software tool would work, or how to use it in a proper way. Documentation is a corner stone in all the phases of a software development process[1]:

- when defining requirements and analyzing the reality in which the software will operate in, documentation on the technologies used could highlight which one is the more suitable for the kind of problem the developers are facing, or which ones could not be used for, for example, compatibility issues;
- when designing a software application, the chosen technology stack guides some design choices, either because of the design of the tools themselves or due to known fallacies in such tools; these "imposed design choices and guidelines" are (and *must*) be specified in the documentation of the technologies used;
- during the development of a software solution APIs and manuals are the main resurce a programmer utilizes to write code: since the tools to develop a software are *software themselves*, a developer relies on the already present documentation to efficently write code that is correct, performing and maintainable;
- the testing phase itself is a development process, in which code is used to test other code; as the development itself, it needs the documentation of framework and tools in order to be performed in an effective way;

[1]I will try to analyze the most generic cycle of a software development process.

- the deployment and maintenance phases of a software are as complex as the design and development phases: they involve a lot of different technologies, ranging from hardware-related ones to networking to cloud solutions; needless to say, documentation on how to use these tools is crucial to their correct functioning and use.

It is now clear that documentation is a crucial tool in the software development process; it's arguably the most important one, even more than the "operative" ones as the IDEs or code editors themselves.

1.2 Writing software documentation

Who is in charge of writing software documentation? The answer is simple: the same people in charge of designing, developing and testing the software they are realizing. Some documentation is written *inside the code* in form of comments, so it is quite obvious that it must be produced by the very same person writing the code. Not only for a locality principle but also because the knowledge which must be embedded in comments (and in documentation in general) should come from the same people who designed the software, to effectively convey the reasoning behind design choices, architectural structures and functioning schemes. We can consider software documentation a part of the final delivered product, that comes *alongside* the software and is needed to instruct other people on how to use such tool.

2 Reproducibility

Reproducibility is a property unique to digital artifacts, and it can be described as the ability of being copied, distributed and moved without harming (or even modifying) the original copy. No other tangible artifact on earth has this property: it is just impossible to take something from someone without harming the owner of the object; instead, copying *a piece of software* or more in general a *a digital document or artifact* does not harm anyone in principle, because the code is just copied and the original owner still has an unaltered version of it. Reproducibility is a property that can be exploited in countless ways, both with good and bad intentions: a software can be distributed without physical supports, so reducing the environmental impact; a document can be passed by copy paste instead of printing, thus speeding up both the adoption of that book and all the processes that involve that book; bank account transactions can be copied, so violating the privacy of people; infinite examples are possible.

2.1 Reproducibility and value

Reproducibility also poses a problem with the meaning of *value* when it comes to certain artifacts: in a way, a reproducible artifact is a never ending resource. There is no scarcity of it so to justify a value of the single copy, as it happens with every limited resource on earth. If the value of an object only came from the fact that there are a *fixed number of copies* of it then reproducible items would be completely empty of value, because they can be reproduced endlessly. On the other hand, we can see a reproducible artifact as a infinitely valuable resource, given its abundance: even if we were to reproduce it a number of times equal to the number of people on earth, we would not harm[2] the original copy nor the original owner of the artifact, because he still will have his own copy intact. Another case could be the value held by items that are not scarce, but are *difficult to access*: it is the case of special recipes in the food industry or proprietary algorithms in the CS field, for example. They can be easily reproduced and replicated but the *procedure needed to get to know them* is very long, heavy and expensive in some cases; this gives them an incredible value. Again, reproducibility poses a problem in this case: once reached, if some of these items are reproducible, then it is very easy to redistribute it and nullify the access barriers to it; this mechanism is often used *the other way around*: companies and institutions put barriers in front of resources to justify a certain high value of them.
Again, I use the real world equivalent example: if I take my sister's bike to use it, I am directly harming her. It is easy to see that she cannot move as fast as with a bike, and she eventually will buy another bike in order to move efficently in the city. The value of the first bike was *forcibly removed from her*, and so she has lost it while I gained it. This would be different if I could make an

[2]directly, at least

exact replica of my sister's bike, out of thin air: then we *both* would be able to cycle through the city and move faster and more efficently, and the original value hold by my sister would be the same, while mine would be increased by that bike. What is impossible with a physical object is possible with a digital artifact: instead of a bike, think about a digital document as a book, a program or an album of photos. I can make an exact copy of it and give it to my sister. Now, we both can access the document, each one of us can access an unaltered copy of it, each one of us can read and make ours the knowledge inside it.
The last distinction I have to point out is the difference between *instrinsic* and *instrumental* value of something: an item (or action or property) is intrinsecally valuable when it is an end itself, opposed as when it is a means to an end, in which case has instrumental value[4]. This distinction will be crucial in understanding the problem exposed in this essay.

3 Software documentation reproducibility

So far I have introduced software documentation in order to explain what is the main subject of this paper, and I have talked about a property of all digital artifacts: reproducibility. Being a digital artifact itself, software documentation has this property; how this property is managed in the industry and education will be the main problem discussed in the rest of the paper.
I will analyze how documentation reproducibility is handled in two cases: the former (that I will call *classical* approach in this paper) is the traditional method of exchanging knowledge between people, while the latter (called *web* approach) is the one that has been emerging for some years now in the software development world, and I think it proposes another interesting point of view on the subject.
Before delving into the two approaches, I need to make a clarification: I am talking about *documentation* and not the *entirety of the software*: even if I borrow a lot of ideas from the Stallman's essay "Why software should be free"[6] the field in which I apply them is slightly different; moreover, I do not share the whole Stallman's vision on the software industry.

3.1 Classical approach

What I call classical approach to documentation in this article aims to describe the method with which knowledge about software (specifically documentation) is passed through in universities, companies and institutions. In these contexts documentation is locked behind amount of money:

- universities distribute knowledge through documentation by selling it to students;
- companies and institutions keep their documentaion secret to everyone, except their employees, so people under contract.

The knowledge is usually very well organized, the manuals are rich of details and the guides are well written and exhaustive. Moreover, usually big companies adopt strict rules for the internal code documentation, enforcing policies on developers to push them to write organized and complete documents, from code comments to manuals passing through design documents, enriched with diagrams and other representation tools to help non technical people to understand the software.
Specifically in universities, manuals and books are sold for even hundreds of euros in order to explain an approach to programming (books explaining how functional programming works instead of object oriented are an example), a specific subdomain of the development world (as books on parallel systems, or distributed ones), or even specific languages in specific frameworks, in certain cases.
The classic approach ensures that all this knowledge is, again, well organized and taught in a very clear way; however, in order to pay for the effort that went in writing such a knowledge base, this type of documentation must be kept absolutely secret to whoever is not willing to pay for it. This approach tries to limit reproducibility: the value of the document (and most of the cases, also its *price*) comes from the limited number of copies available, or the significant accessibility barriers to it, as explained in 2.1.

3.2 Web approach

With "web" approach I address the method of sharing knowledge typical of the open source web projects, as the most popular web developing frameworks[3]. I avoided the term "open source" doc-

[3]Such as VueJS, ReactJS or AngularJS

umentation on purpose, to not mislead to reader into thinking the problems I want to refer to are related only to the authoring of documentation (2.1).
When adopting this method, documentation is always, from the beginning of the development process to the end, fully available to the public. The idea behind such a full disclosure of information (that encompasses APIs, manuals, comments, guides, walkthrough, examples etc) is that the more documentation is available to the end user, the easier the first steps in learning and using a technology will be, thus leading to some significative gains for the developers and designers of such piece of technology:

- faster adoption: a "simple" (or perceived so) technology will be preferred against the ones that lack or have less available documentation;
- deeper comprehension of the technology;
- richer feedback: if the documentation covers enough topics and branches, and the user is willing to study it deeply, the feedbacks on possible problems in it or improvements/extensions to be added would be very well thought out;
- community involvement: especially for open source projects, the documentation plays a huge role in attracting people to work on it;

This kind of approach can be used with all styles and paradigms of software development, from full proprietary software (with even zero disclosure on vulnerabilities) to open source projects, where it is actually more diffused. The reason is that if you are ready to open source your project, chances are you want to distribute freely also the documentation of it.
This approach to the documentation does not provide guarantees on the quality of the documents, and often the tools documented in this way have big issues with the guides and APIs[4]; this is not true when an open project is backed by a big company or institution.
Web documentation exploits fully the reproducibility property of digital documents, allowing everyone to access the knowledge needed to use a certain tool.

3.3 Comparison of approaches and analysis

The classic and the web approaches to software documentation are completely different in theory, but this does not mean that they are used in very different projects: the classic approach is still utilized in some of the "academic" computer science fields where all the code is available for free, as for example formal model checking or compiler theory; the web approach instead is more often seen used in the big companies world, where tech giants exploit the community-creating power of it to develop their tools at best.
I think that the main difference between the two approaches is the interpretation of *the value of documentation*; they are also very different under a *pure utilitarian* and under a *deontological* points of view. From whichever perspective though, the classic approach only results to be damaging to the documentation itself, to the software, to the developers community whole.

Documentation value The first aspect that differentiates the two approaches is the way the documentation value is perceived: while the web approach sees the documentation as an essential *accessory* to the software, having no value on its own, the classic approach interprets the documentation as a *standalone entity* that holds value. The difference is quite stark: the web approach gives documentation instrumental value, while the classical approach gives it intrinsic value. While the former sees the documentation as a *means* to achieve an end (the end being developing well) the latter instead perceives it as an *end* by itself, so a completely unrelated entity with regards to the development process.
As an example, we can consider the role of a wrench in the process of building a car: we can easily see how the wrench *alone*, without the car to be built, holds only the value that comes from the materials it has been built with. This is because it was designed specifically as a tool to help people in a specific process, and without such process it is just a mixture of metals and plastic. But when the process is software development, and the tool we are considering is a *reproducible* one, then

[4]a good example could be all the poorly documented libraries available for typescript: being in most of the cases just a simple extension of their equivalent in javascript, developers simply *refer to the original documentation*, forgetting the completely different approaches between the two languages.

we cannot even consider the materials that went into the construction of it! Documentation is the perfect example for a pure *mean to an end*.

Utilitarian interpretation The utilitarian principle "[...] approves or disapproves of every action whatsoever according to the tendency of [such action] [...] to augment or diminish the happiness of the party [involved][...]"[1], so from an utilitarian point of view every action should be taken (or not taken) measuring only the amount of people that benefit from it, so the *overall quantity of happiness* produced by such action. Bentham argues that being humanity driven only by pain and happiness, all actions should be taken in order to tend towards the happiness; but he does not limit to individual actions, but extends to "every measure of government"[1].
It is quite simple to see, how this apply to the documentation issue: the "happiness" to be looked for in this context is the developing of a tool that helps people in their activity, and the choices we can make are the two approaches to the documentation; the one that *quantitatively* generates more happiness (simply by granting access to knowledge to more people) is the Web approach.
Even in a non-quantitative approach as the Mill's qualitative approach[3], where the philosopher argues that pleasures cannot be measured in a quantitative manner but must be analyzed also from a qualitative point of view[5], the situation does not change: the web approach generates *more* happiness by allowing more people to access that knowledge base and generates *better* happiness because it generates education and knowledge itself; on the other hand, the classical approach generates happiness only in form of (generally) economic values for the institutions or individuals who own (or have rights on) the documentation.

Deontological interpretation The categorical imperative in its first formulation "Act only according to the maxim whereby you can, at the same time, will that it should become a universal rule"[2] gives us another interesting standpoint from which we can compare the two approaches. A person who wants to embrace the classical approach to documentation (for the formulation of the imperative) will want everyone to follow his/her example, not to distribute the documents alongside the software they are developing. This means that no programmer shares the knowledge about his/her work (at least, not freely) and if it happens that s/he distributes freely his/her work, this would be very difficult to utilize and integrate and *put to a good use*. It is easy, I suppose, to picture what this will lead to: no programmer will be capable of accessing other developers work, and so they would have to "reinvent the wheel" every time, significatively slowing down whichever project they are working on or process they are part of. The whole software development world would have an additional entry barrier, represented by the cost of accessing the knowledge needed to use all the tools necessary for the work. Open source software would make no sense: what would be the point of distributing freely a software that nobody can use, because they have to purchase several manuals to do so?

4 Conclusions

Documentation is the corner stone of software development, but such as every digital artifact it is difficult to regulate its reproducibility and intangibility. There are two main approaches to this issue, but only one of them actually exploits the reproducibility property in order to generate additional value for the people working with that tool, while the other tries to block such property in order to align a digital artifact to its physical equivalents, damaging it and not making use of the full potential it offers, just to generate economic value.
The software development world as a whole, and in particular the people in charge of teaching and sharing knowledge about programming and designing software, should put more emphasis on the need for accessible, free and good documentation, because it is the only tool available to make a technology usable, useful, and last but not least enjoyable.

[5]so preferring intellectual pleasures as friendship, art, reading and conversation, for example

References

[1] Jeremy Bentham. *On the Principle of Utility*. 1780.

[2] Immanuel Kant. *Groundwork of the Metaphysic of Morals*. 1785.

[3] John Stuart Mill. *Utilitarianism*. 1863.

[4] Iwao Irose; Jonas Olson. *The Oxford Handbook of Value Theory*. 2015.

[5] The Linux Information Project. *Documentation Definition*. URL: `http://www.linfo.org/documentation.html`.

[6] Richard Stallman. *Why Software Should Be Free*. 1991.