[L9P1]

The M in RTFM: When Code Is Not Enough

Writing developer documentation

Developer-to-developer documentation can be in one of two forms:

1. Documentation for *developer-as-user*: Software components are written by developers and used by other developers. Therefore, there is a need to document as to how the components are to be used. API documents form the bulk of this category.
2. Documentation for *developer-as-maintainer*: There is a need to document how a system or a component is designed, implemented and tested so that other developers can maintain and evolve the code.

Writing documentation of the first kind is easier because a component with a well-defined API exposes functionality in small-sized, independent and easy-to-use chunks. Examples of such documentation can be found in the Java API (http://download.oracle.com/javase/8/docs/api/). Writing documentation of the second type is harder because of the need to explain complex internal details of a non-trivial system. Given below are some points to note when writing the second type of documentation.

Go top-down, not bottom-up

When writing project documents, a top-down breadth-first explanation is easier to understand than a bottom-up one. To explain a system called *SystemFoo* with two sub-systems, front-end and back-end, start by describing the system at the highest level of abstraction, and progressively drill down to lower level details. An outline for such a description is given below.

[First, explain what the system is, in a black-box fashion (no internal details, only the external view).]

SystemFoo is a ....

[Next, explain the high-level architecture of *SystemFoo*, referring to its major components only.]

SystemFoo consists of two major components: *front-end* and *back-end*.
The job of *front-end* is to ... while the job of *back-end* is to ...
And this is how *front-end* and *back-end* work together ...

[Now we can drill down to *front-end*'s details.]

front-end consists of three major components: *A*, *B*, *C*
A's job is to ... *B*'s job is to... *C*'s job is to...
And this is how the three components work together ...

[At this point, further drill down the internal workings of each component. A reader who is not interested in knowing nitty-gritty details can skip ahead to the section on *back-end*.]

...

[At this point drill down details of the *back-end*.]

...

The main advantage of this approach is that the document is structured like an upside down tree (root at the top) and the reader can travel down a path he is interested in until he reaches the component he has to work in, without having to read the entire document or understand the whole system.

'Accurate and complete' is not enough

Documentation that is "accurate and complete" is not going to be very effective. In addition to accuracy and completeness, a document should be easy and pleasant to read. That is to say, *it is not enough to be comprehensive, it should also be comprehensible*. The following are some tips on writing effective documentation.

- Use plenty of diagrams: It is not enough to explain something in words; complement it with visual illustrations (e.g. a UML diagram).
- Use plenty of examples: When explaining algorithms, show a running example to illustrate each step of the algorithm, in parallel to worded explanations.
- Use simple and direct explanations: Convoluted explanations and fancy words will annoy readers. Avoid long sentences.
- Get rid of statements that do not add value. For example, 'We made sure our system works perfectly' (who didn't?), 'Component X has its own responsibilities' (of course it has!).

Do not duplicate text chunks

When describing several similar algorithms/designs/APIs, etc., do not simply duplicate large chunks of text. Instead, describe the similarity in one place and emphasize only the differences in other places. It is very annoying to see pages and pages of similar text without any indication as to how they differ from each other.

Make it as short as possible

Aim for 'just enough' documentation. The readers are developers who will eventually read the code. The documentation complements the code and provides just enough guidance to get started. Anything that is already clear in the code need not be described in words. Instead, focus on providing higher level information that is not readily visible in the code or comments.

Explain things, not list diagrams

It is not a good idea to have separate sections for each type of artifact, such as 'use cases', 'sequence diagrams', 'activity diagrams', etc. Such a structure, coupled with the blatant inclusion of diagrams without justifying their need, indicates a failure to understand the purpose of documentation. Include diagrams when they are needed to explain the system. If it is a must to provide a comprehensive collection of diagrams, include them in the appendix as a reference.