[Handout for L4P3]
The M in RTFM: When Code Is Not Enough

Writing developer documentation

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

1. Documentation for developer-as-user: Most software components we write will be used by other developers. Therefore, we have to document how to use those components. API documents form the bulk of this category.
2. Documentation for developer-as-maintainer: We have to document how a system or a component is designed, implemented and tested so that other developers can maintain and evolve the code.

Writing the first kind is easier because components expose functionality in small-sized, independent, easy-to-use chunks. For examples of this type, refer to Java API documentation (http://download.oracle.com/javase/6/docs/api/). Writing the second type is harder because we may have to explain a non-trivial system with a complex design. Given below are some points to note when writing the second type of documents.

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. Let us say you are explaining 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. Given below is an outline for such a description.

[First, explain what the system is, in a black box fashion (no internal details, only the outside 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*
> *front-end*'s job is to ... *back-end*'s job 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 you can further drill down to the internal workings of each component. A reader who is not interested in knowing these nitty-gritty details can skip ahead to the section on *back-end*.]

> ...

[Here, we can drill down to *back*-end's details.]

> ...

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

If you take the attitude "it is good enough to be accurate and complete", your documentation is not going to be very effective. In addition, make it as easy and pleasant to read as possible. Some things that could help here are:

- 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: Do not stop at explaining your algorithms in words. Show a running example to illustrate every step of the algorithm, parallel to your 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 you have to describe several similar algorithms/designs/APIs, etc., do not simply duplicate large chunks of text. Instead, describe the similarity in one place and emphasise 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. Your readers are developers who will eventually read the code. The documentation complements the code and provides just enough guidance to get started. Things that are 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 artifacts, such as 'use cases', 'sequence diagrams', 'activity diagrams', etc. Such a structure indicates that you do not understand the purpose of documenting and are simply dumping diagrams without justifying why you need them. Use those diagrams where they are needed to explain the system. If you must, you can give a comprehensive collection of those diagrams in the appendix, as a reference.

---End of Document---