Documentation Guidelines

Note: this document is a *draft*, not endorsed by the TCT.

Written by Mario Juric, 2012-04-05

Why document?

Writing *good* documentation is hard. It takes experience, effort, and some talent. Writing one that is *good enough* is not. Numerous developer-documented open source projects prove it's possible, and essential.

Our code is unevenly documented. While there are exceptions, most of the documentation is either function-level (autogenerated by doxygen), or scattered on the wiki with little organization or context. Most worryingly, some (critical) packages have little or no documentation at all.

Due to this, we're beginning to observe the following:

- Developers who are not authors of a particular package spend lots of time reverse engineering the code to understand what it does.
- Alternatively, they get started by talking to the original developers. This is repeated for every new developer, taking time from all.
- Sometimes, they conclude that reverse engineering is not worth the time and duplicate the functionality. This defeats the major purpose of the modular design (code reuse).
- Other times, they may not even know the needed functionality already exists, again leading to code duplication and waste of resources.
- Developers come and go; the code they have written but not documented reduces in usefulness when they leave. This is again a waste of both resources and time.
- On a higher level, without any documentation there's no way to connect the code to high-level DM plans. This makes it difficult for the project to keep track of progress. It leads to periodic "heroic runs" to reverse engineer and document the code for every review and every report.
- Also, the lack of starter documentation makes it impossible for the project to hire technical writers to take it to the next level. With a geographically distributed team, a technical writer cannot produce something from nothing.
- Finally, if the current practices continue we're in danger of taking a huge productivity hit when construction starts: veteran developers will spend a large fraction of the time training the newcomers (larger than they would otherwise).

Mind melds do not scale

The primary argument against documenting is that it distracts from writing code. In a large, distributed, team, where the age of an average developer is ~3 years (of work on the project), **not documenting also distracts from writing code**. Most insidiously, most do not feel this intuitively: while we all tend to remember the hateful hours spent trying to churn out a paragraph of documentation, we tend to forget the hours spent explaining things in person or via e-mail.

0*exp(number of technical writers) is still 0

The other argument against documenting is that developers are incapable of producing good documentation, and that it's a technical writers' job to do it. Firstly, developers **are capable of producing good enough documentation**. However, even if we had a technical writer (which we currently don't), they require a starting point on which to expand on. **Technical writers are not a tool to get from no documentation to some documentation, they're a tool to get from 'good enough' documentation to 'great documentation'.** Having some developer-written notes is a prerequisite for hiring a technical writer.

Going forward

As with everything, we need to strike the right balance. Asking for excellent documentation that is up-to-date at all times is obviously unreasonable. But having no documentation at all, or just doxygen-level documentation, is equally undesirable. We need to strive for the middle ground: documentation that is "good enough", sufficient for new developers (as well as the authors themselves!) to understand the code, and sufficient for management to track progress and plan future development.

This document is an attempt to set guidelines for how to write documentation what is 'good enough', but does not overtax our limited resources.

Levels of documentation

I propose having three levels of documentation:

System Architecture Overview and Developer Guide

A high level overview of the whole DM system. It needs to connect the actual coded packages to high-level designs from LDM-XXX documents. For example, LDM-151 chapter 2.2.3 describes the DRP pipeline. This document needs to show how the current prototype code (`pipe_tasks` et al.) maps to the designed pipeline (note: it's not obvious to me that these should be separate documents).

A second purpose of this document is to get new developers started. The text that is in devenv/doc seems like a great start. This document should link to package-level documentation for more details. For ease of editing, the master copy of this document should live on the wiki. Help will be provided to keep this document up-to-date (e.g., by Frossie or Mario).

Package-level documentation

This is the entry point for a new developer looking at the package for the first time, and it must quickly answer what the package does, why, and how. It can begin with something as simple as a `README` file, growing to full-fledged reST, wiki, or other documentation as more details are added. Alternatively, the `README` can hold pointers to other places where non-autogenerated documentation can be found; if it's on the wiki, the link should be of the form `LSST/Documentation/<package_name>`, to maintain consistency. Just having a pointer to doxygen output does not qualify as having provided documentation. Note: it is very important that package-level docs are kept up-to-date, and it is the developers' and reviewers' sole responsibility to make it so.

API (reference) documentation

This is documentation auto-generated from the sources (e.g., Doxygen). Every public API function **must** be documented. Private functions (implementation details) should be documented in the code, but **must not** be exported to public (doxygen generated) documentation. Doing otherwise clutters the documentation and exposes implementation details to the world.

Package-level Documentation

Package-level documentation needs to provide sufficient detail to allow a reasonably skilled developer to discern what it does, how it does it, and why. That is its purpose, no more, no less.

At minimum, it should provide answers to the following:

What the package does

Overview of *what* the package does (the intent). This should be no longer than 10-15 sentences, sufficient for the reader to quickly understand what is the purpose of the package.

Who and how should use it

Usage description. This may be written as a short general introduction, plus examples on how to perform the most common user stories defined for the package. If any examples are bundled with the package, it should have pointers to those. Note that **all examples have to be documented**; having a directory of undocumented `.py` files with cryptic names *does not* qualify as having provided example code.

Architecture overview (how it does it)

This should provide a high-level overview of the architecture of the package, sufficient for the reader to learn about the details from reference documentation. It must explain any non-obvious design decisions or unconventional implementation details (can be in form of a F.A.Q.). It should contain a historical introduction, if it helps the reader understand how the package came to be the way it is (e.g., this package started as . . . but was then split off when it was deemed too complex). Finally, it should note any identified deficiencies and TODO items (i.e., expected future development).

Notes

Any other relevant information a developer or user should know about. For example, if there was data included in the package, where did it come from.

Changelog

A brief log of most important changes.

References

Links to autogenerated documentation (doxygen/sphinx), or to related packages.

Maintainer and Authors

Who is the current maintainer of the package. It is also customary to acknowledge the major contributor(s), or initial author(s).

If this documentation resides in a `README` or other text files within the package, it is recommended reStructuredText (reST) is used as markup. This makes the documentation source easy to read, yet machine readable and usable by tools like Sphinx or similar. LaTeX, in particular, should be avoided; LaTeX sources are notoriously difficult to read.

Examples

Some examples can be found at:

- http://dev.lsstcorp.org/cgit/LSST/DMS/pex_config.git/plain/README
- http://dev.lsstcorp.org/cgit/contrib/demos/lsst_dm_stack_demo.git/plain/README

- http://dev.lsstcorp.org/cgit/LSST/DMS/devenv/devtools.git/plain/README

Unfortunately, the latter two of these are *unusual*, as they describe packages that deal either with data or developer support tools. More relevant examples should be added as soon as they appear.

When to document

Documentation should be written once a design is stable, and an initial implementation exists. Documents written for design reviews are usually good starting points.

In particular, no tickets should be merged to master without the reviewer having made sure that:

1. all public API packages, modules, functions, classes, etc. are documented (via doxygen blocks or otherwise);
2. the package-level documentation (the "README" or equivalent), including examples, have been updated to reflect new features or design implemented by the ticket;
3. the high-level documentation has been updated if necessary, or a ticket to update it has been opened.

Note that 1) will apply to almost every merge, 2) will be needed for features that add or change major features of the package, and 3) will be required only when changes to the package have a significant impact to the overal design (examples: replacement of old pipeline code with `pipe_base/pipe_tasks`, or `pex_policy` with `pex_config`.

What does *not* qualify as high-level documentation

Having only doxygen-level docs

In the words of Django's Jacob Kaplan-Moss:

> *"Auto-generated documentation is almost worthless. At best it’s a slightly improved version of simply browsing through the source, but most of the time it’s easier just to read the source than to navigate the bullshit that these autodoc tools produce. About the only thing auto-generated documentation is good for is filling printed pages when contracts dictate delivery of a certain number of pages of documentation. I feel a particularly deep form of rage every time I click on a “documentation” link and see auto-generated documentation.*
>
> *There’s no substitute for documentation written, organized, and edited by hand.*
>
> *I’ll even go further and say that auto-generated documentation is worse than useless: it lets maintainers fool themselves into thinking they have documentation, thus putting off actually writing good reference by hand. If you don’t have documentation just admit to it. Maybe a volunteer will offer to write some! But don’t lie and give me that auto-documentation crap."*

E-mailed it to lsst-data

An e-mail to lsst-data is just one notch above (below?) talking to someone in person. It should be considered transient.

Mentioned it on a phonecon

See above.