Science and Technology Facilities Council

Technical Report
DL-TR-2020-004

Software Outlook Documentation Tools: overview and best practices

B Mummery

December 2020

Enquiries concerning this report should be addressed to:

Chadwick Library
STFC Daresbury Laboratory
Sci-Tech Daresbury
Keckwick Lane
Warrington
WA4 4AD

Tel: +44(0)1925 603397
Fax: +44(0)1925 603779
email: librarydl@stfc.ac.uk

DOI: 10.5286/dltr.2020004

ISSN 1362-0207

Software Outlook

DOCUMENTATION TOOLS

Overview and Best Practices

Dr Benjamin Mummery
STFC Hartree Centre
September 2020

Contents

1 Why Document?

One of the core tenets of the group in which I personally work is “undocumented code is unusable”. This is not to say that undocumented code is never functional, indeed we are certain that many undocumented codebases are of excellent quality and are perfectly fit for purpose. However, it is almost guaranteed that anyone attempting to use, develop, or maintain a piece of code will have limitations on their time – deadlines, scheduling, or simply the opportunity cost of choosing one task over the other. Undocumented software represents a large overhead in terms of time and effort. If your maintainer faces an uphill battle each time they make a change, updates will be more demanding, less efficient, and therefore less frequent. Less frequent updates make your software less competitive. And as soon as it looks like it might be quicker or simpler to use a different codebase for the task, be it due to the learning curve, a lack of features, or any other reason, that is precisely what the end-user will do. *Undocumented code is unusable because nobody will ever use it.*

Documentation therefore needs to address two sometimes wildly disparate audiences: end users, and current and future developers. Thankfully, it is seldom necessary for a single piece of documentation to serve both masters, however this does mean that requirement for documentation can feel like a significant burden for the developer. Fortunately, there are a number of tools and techniques that exist to aide us in efficiently creating good documentation.

In this guide, we will discuss the attributes that form good user and developer documentation, and examine some of the tools available for streamlining this process.

2 Developer Documentation

Sooner or later, all code will require modification, be it patches for newly discovered bugs, updates to make use of new versions of dependencies, or the addition of entire new features. In order to accurately gauge the effect of any change to a codebase, we as developers need a reasonable understanding of how the code operates, both on the micro level of “how does this specific function work”, and in the macro level of “what scenario does this method address, how is it approached, and why is that approach the one chosen.” The former is most efficiently covered by code comments, while the latter can require further supporting documentation, although these trends are by no means absolute.

2.1 Code Comments

Code comments are exactly what they sound like – pieces of text written into the source code which serve no functional purpose for the computer, but instead provide helpful information for a person reading the code. This information can be anything from notes to yourself or other developers or maintainers that something needs to be changed, to overviews of what each module or method is intended to do, to explanations of why something needs to be done a specific way. All of these enrich the code, providing essential context and clarity where it might otherwise be lacking.

2.1.1 Why Comment Code

A strong argument can be made for sufficiently well-written code not requiring documentation. If your variables and functions have sensible (i.e. human-readable) names, and your code is structured cleanly, then anybody reading it should be able to follow it with no problem. However, while this argument is compelling, it fails to address 4 factors:

"Sensible" is context dependent.

While it might make perfect sense to you what *amp_perp* means, somebody without your complete knowledge of what the code does, how it does it, and why it needs to do it, may find it rather more difficult to understand.

"Sensible" has a limited half-life.

It has, at time of writing, been a little over a week since I selected *amp_perp* as an example variable name for point 1, and I honestly cannot recall what it was short for. I am *reasonably* certain that "perp" was a contraction of perpendicular, but was "amp" short for amplitude, or amplified? Now sure, in the context of an actual piece of code there is a lot of other context I could look at to figure out what *amp_perp* actually means, but the need to do so presents a real barrier to easily reading code. And again, I'm the one who came up with *amp_perp*!

The readability of code depends on the reader's ability to read code.

Ideally your code should be understandable by the widest audience possible. If you depend on well written code to explain itself, this limits the reader base to those who are already fluent in the language in which you wrote it. Reasonably well commented code can accommodate a reader whose understanding of the language is still developing. Very well commented code can actually be read and understood by someone with no understanding of the programming language used – not on a line-by-line detail level, but the broad functionality, purpose, and order of operations should be understandable.

Code lacks nuance.

It can, at best, describe *what* it does, but lacks any insight into the thought process or constraints that dictated *why*. This may not be an issue for someone simply looking to understand what function your code performs, but anyone tasked with maintaining, updating, or making any change to it runs the risk of not being aware that the "better" way they just thought up to approach the problem is something that you'd already tried and proved impossible. At best this leads to needlessly duplicating work, at worst it can break entire codebases.

Commenting is, in other words, a small action that can make using and maintaining your code exponentially easier, regardless of whether it is you or somebody else doing the maintenance.

2.1.2 When to Comment Code

By dint of being embedded within the code itself, code comments naturally skew towards the micro view, and it can be easy to neglect the higher-level overview. To combat this, it is necessary to have an overall hierarchical scheme for where comments should be added, ensuring that the higher-level is not neglected. This can also aide in minimising and focussing comments, reducing the additional work required to maintain their accuracy as the code develops around them.

The specifics of this can vary from case to case, but as a rule of thumb, comments are required at the module level to present a high-level overview of the module's functionality. Procedures/methods should also be commented, describing their specific functionality, data types, and brief instructions as to how to use them. Comments describing a particular line or block are required *only* where the complexity of the code means that it would be arduous for the reader to interpret, or where they contain pertinent information that is absent from the code.

2.1.3 How to comment code

Exactly what information should and should not be included in a comment, and often whether a comment should be included at all, is often down to the judgement of the developer, and it can be easy to find yourself on the prongs of a dilemma over whether something should be included. Below we have set out a series of guidelines and factors that it can be useful to keep in mind when doubt arises. Most of these can be summed up by one simple, overarching message: *write for the reader.* Taking a step back to look at your code with fresh eyes (or as close as you can get while still being the person who created it in the first place) is a useful technique in general, but is essential for effective commenting.

Priorities Intent Over Function

Comments that describe what the code they are commenting does are useful. They address points 1, 2, and 3 above, and make it much easier to scan through code quickly. However, given enough time a reader will be able to compensate for their absence. Comments that describe the *intent* of the code, the purpose it fulfils rather than the mechanism by which it operates, provide information that cannot be inferred by other means. They are also significantly more robust, requiring far less work to keep up to date as code is modified – the specific methodology of a function changes frequently during development, but its overall intent seldom changes.

Comment at Higher Abstraction

There is a word for a description of a piece of code that explains its purpose and function in perfect detail. That word is "code". The purpose of comments is to provide an understandable description and explanation of the code, not to restate it in exhaustive detail.

Consider Reading Order

Much of the time, the order in which content is laid out in code is less than intuitive for a human reader. This can often be addressed with comments, providing a high-level overview at the top of the document to provide a context with which to understand the details below. Keeping to a consistent scheme (see section 2.1.2) can often be enough to address this need.

Keep Consistent Spatial relationships

Watch out for ambiguity about what in the code each comment relates to. Having a consistent approach to the spatial relationship between comments and the relevant code chunk helps to prevent this, as does considering the clarity of this relationship. Comments should, for example, be indented to the same level as the code they describe, on the immediately preceding line, and follow at least one blank line. In this way, the relationship between comment and code is made immediately visible.

Clean up Comments

Comments that are incorrect, or refer to code that has since been removed, are worse than having no comments at all. Comments need to be viewed as a part of the code to which it is attached, and modifications to one must be reflected in the other. This can seem to present a large additional workload when it comes to maintenance, which brings us onto our next point:

Comment Minimally

Hopefully by now we have persuaded you that commenting your code is important, and that you should be doing it. However, a balance must be drawn between complete coverage in comments, and the additional work needed to create and maintain this alongside the code itself.

Include Authorship and Licencing Info by Default

Depending on your organisation, this can be anything from a nice-to-have to a legal requirement. The quickest and easiest approach is to simply have a pre-formatted comment block that can be inserted at the top of any piece of code you write and modified appropriately. As with many things, it is better to have the information included and not need it, rather than the other way around. This benefits the end-user, removing any ambiguity about what they are permitted to do with the code; it benefits the maintainer, providing a point of contact if they need to contact you for further information; it also benefits you by providing basic legal protections from plagiarism and theft of intellectual property.

Choose a Style Guide

As we will be discussing later (see Section 5), consistent commenting is essential if you're using an automatic documentation tool such as Doxygen or GhostDoc. They are also a massive boon for ensuring that code written by more than one person is consistently and easily readable. Finally, even when you are the only person working on a project, choosing or writing a style guide can help to coalesce and codify your approach to commenting, ensuring that you yourself are consistent and complete in your documentation.

2.1.4 Example

Consider the following function:

```
def hello_world():
    print("Hello, world")
    return 0
```

What this does and how is probably evident even to readers who are not familiar with Python, and therefore it *could* be assumed that adding comments would be unnecessary. However, compare it to the following commented version:

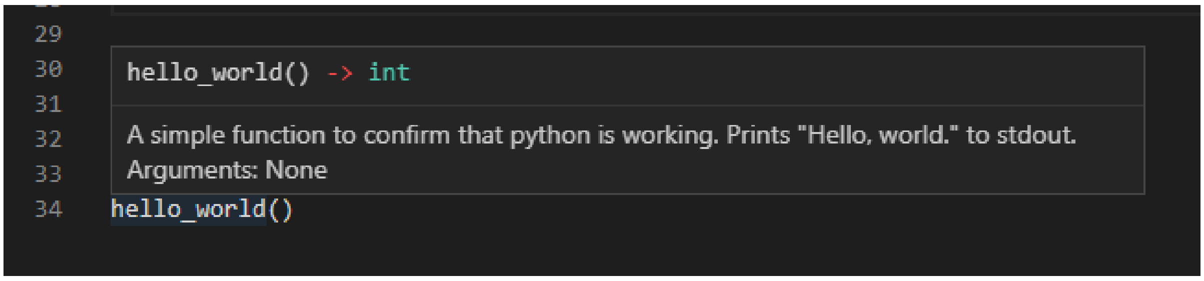
```
__author__      = "Benjamin Mummery"
__copyright__   = "Copyright 2020, STFC"
__credits__     = ["Benjamin Mummery"]
__license__     = "MIT"
__version__     = "1.0.1"
__maintainer__  = "Benjamin Mummery"
__email__       = "benjamin.mummery@stfc.ac.uk"
__status__      = "Demo"

"""
examply.py: demonstrations for commenting styles.
"""

def hello_world() -> int:
    """
    A simple function to confirm that python is working. Prints "Hello, world.”
    to stdout.
    Arguments:
        None
    """
    print("Hello, world.") # Comma included because I'm a stickler for grammar.
    return 0
```

This version includes the header for the file itself, containing useful information regarding the author, copyright, licence etc. in a fashion that can be easily read by interpreters. Python docstrings are used to detail the *intent* behind both the module and the function it contains, explaining not just what the code does, but what purpose it serves within the software. We have also added type hints to explicitly dictate the expected types, so other pieces of the code that might use this function do so with no ambiguity as to what behaviour to expect of them. Finally, an in-line comment is used to explain the reasoning behind a specific decision, in this case the inclusion of an often-neglected comma for grammatical reasons.

As a practical example of the utility of these comments, certain editors (in this case Visual Studio Code) can use docstrings to provide tooltips while editing:



```
hello_world() -> int

A simple function to confirm that python is working. Prints "Hello, world." to stdout.
Arguments: None
hello_world()
```

This removes the need to move through the code to consult the docstring or examine the function directly, allowing the developer or maintainer to arite much more efficiently.

This is obviously an extreme example; it is very rare for the comments to make up the majority of your code. However, we hope that this demonstrates the utility of properly commented code. Note also that each of these comments is *motivated*, i.e. that it either adds context information to the code, or collates existing information in a human- (or machine-) readable fashion.

2.2 Supporting Documentation

Code comments form the bulk of developer documentation, however it is frequently useful, or even necessary, to include materials such as graphical or pictorial information, or simply more general, higher level discussion, that fall outside the scope of these comments. In these cases, additional documents are included to meet these needs.

What form these documents take will depend on the context and specifics of the project. They can be technical reports, the results of profiling, explanations of the reasoning behind certain algorithmic decisions, project requirement overviews, or photographs of hand-drawn diagrams of the software's structure – any file that might assist a developer or maintainer may be included.

Given the much more piecemeal nature of this documentation, specific guidelines are not helpful here. The best rule of thumb is that any project-specific document to which you refer while developing the code should probably be included to help future developers.

3 End-User Documentation

Developer documentation such as code commenting is an invaluable tool in ensuring that your code is understandable and maintainable. However, most users will never actually see the source code or supporting docs for your software.

The end-user is unlikely to be concerned with how the code works, but they are going to be looking for the quickest route to achieving their goal. As such, writing suitable documentation requires that we reframe our approach to the software.

In general, end-user documentation can be divided into four categories: Tutorials, How-to Guides, Reference Guides, and Explanations. These form a hierarchical sequence, with tutorials as an entry point for new users, and reference guides as a useful tool for those already familiar with the software.

3.1 Tutorials

Tutorials are the most hand holding and least technical framing of documentation. A tutorial is a lesson, taking the reader through a series of steps to quickly and simply illustrate how to achieve one or more of the software's functions. This serves a second, subtler purpose of demonstrating what the user can achieve with your tool. A tutorial can be expected to be at least skim-read by the vast majority of users, since it's the easiest way to quickly get a handle on your software.

3.2 How-to Guides

How-to Guides build on the foundation laid by tutorials. They are aimed at users who possess a basic knowledge of the features, tools, and functionality of the software, but who need guidance on how to achieve a broader set of goals. How-to Guides can be compared to recipes – a reader is assumed to be competent in basic tasks such as how to whisk an egg, or sift flour, and to require only the correct sequence of steps to laid out for them in order to bake a cake, or breaded chicken, or whatever meal it is they desire.

How-to guides are not intended to be read by all users, but rather to be consulted when a user realises that they don't know how to perform a specific task. As such, it is paramount that the end-goal of the guide be made explicitly clear at its beginning. One should not have to read the entire recipe in order to determine whether it is for cake or chicken.

3.3 Reference Guides

Reference guides are the most technical and least self-explanatory framing of documentation short of reading the commented code directly. They are also, in general, the easiest to write since they most closely mirror how we as developers think about our software.

Reference guides offer a basic description of how to use the software on a technical level. To continue our cooking analogy from the previous section, these would be the equivalent of the user-manual for the blender. Most users don't need to understand its operation beyond the level of "which button makes it go", but an advanced user with a specific task in mind needs to know how to change the attachments and adjust the settings. Reference Guides should cover each public function, method, API, etc.; everything that the user could access without modifying the code itself should have a reference illustrating its specific usage. When the user needs to know how to instantiate a specific class or call a particular method, this information should be encapsulated within a reference guide.

3.4 Explanations

Explanations are in-depth discussions of topics that you consider to be helpful or necessary to developing a higher-level understanding of the code. Explanations seldom appear as separate sections in their own right, but tend to be scattered throughout the other sections. Unlike tutorials, how-to guides, and reference guides, explanations are not defined by a specific goal the user might want to achieve, but rather they function as paratextual addenda that offer illumination on specific topics.

To once again return to the well of cooking analogies, explanations would be the equivalent of digressions on the chemistry of bicarbonate of soda or the interplay of spices, things that would enhance the reader's understanding of not only what is being achieved, but how.

Much of the content of Explanations may well dovetail with that of Supporting Documentation (See Section 2.2), however Explanations will be presented with a non-technical reader in mind.

4 Manual Documentation Generation Tools

With the exception of code comments which are, unsurprisingly, written in the same editor you are using to write code, writing documentation requires that you make a choice with regards to the format and editor you use. This may be dictated by the specifics of the project, but often the decision falls to the development team.

In principal, documentation can be written in any format that a user can read, however in practice the requirements of readability and maintainability mean that there are a number of features that are all but essential. In general, the following features are more-or-less essential in an editor:

- Formatting options – clear and easy to follow formatting makes for documents that are easier to read, plain-text is often unsuitable.
- Images – visual aids are invaluable, for example in documenting a UI or describing a network of relationships or dependencies.
- Alt-text – Accessibility readers rely on alt-text to interpret images, so if you're using an image this feature is a must.
- Code snippets – whether including an example function call or referring to a particular block of code, you will want to be able to include code snippets (and for these to be visually distinct from the bulk of your text). Bonus marks if you can also include syntax highlighting.
- Portability – development is frequently a collaboration, and people often use multiple machines, so your documentation must be modifiable from any computer that can modify your code.

Last, but by absolutely no means least, all of the features mentioned above must be easy to use – the more hassle it is to maintain your documentation the less accurate it will end up being.

These requirements still leave far too many potential formats to cover, however there are [2] commonly used options that we wish to discuss: Markdown and LaTeX.

4.1 Markdown

4.1.1 Overview

Markdown is a lightweight markup language with plain-text-formatting syntax. In other words, you write the formatting of your document at the same time as you write the contents, so rather than highlighting a word you wish to emphasise and pressing Ctrl+I to add italic formatting, in markdown you would simply write _text to be emphasised_. This makes it easy to write clearly formatted, well-structured documents very quickly, without the need for a specific editor. This convenience has led to markdown becoming probably the most widespread tool for writing software documentation.

Markdown's primary advantage is its simplicity. Speed and convenience are essential for preventing documenting your code becoming an onerous task, and markdown can be written with almost any plain-text editor. It is light and portable, human readable even in its uninterpreted form, and provides built-in functionality for code blocks. Many renderers also support syntax highlighting for code blocks.

Where markdown falls down is where this simplicity begins to get in the way. The more complex a task becomes, the less likely markdown can meet your needs. Most renderers support a degree of inline HTML to provide additional functionality, but this varies greatly and reduces the attractive portability of the language. Then again, if your documentation is getting that complex, you may wish to rethink your approach rather than the tool!

The one major caveat to Markdown is the lack of standardisation. While the core remains the same, different renderers support different sets of additional features. This has led to the development of "flavours" of markdown, for example the GitHub flavour includes support for mentions, SHA-1 hashes, Pull Requests, Task Lists, etc., all of which help integrate it with GitHub's broader functionality, but reduce the portability of documents that use them.

4.1.2 Editors

Any plain-text editor will do the job of allowing you to create a markdown document. However, it can be very helpful to have a preview or renderer built in so that you can check the final appearance. A full overview of available markdown editors is beyond the scope of this guide, however we include a brief overview of the most commonly used editors in

Overview of Markdown Editors.

I personally use VSCode for a lot of development tasks and therefore find it to be the most convenient tool for writing documentation, especially since I can do so in parallel with developing the software. The extension-based nature of VSCode also allows for additional functionality to be added when needed, for example exporting Markdown files as PDFs.

4.2 LaTeX

4.2.1 Overview

LaTeX is an amazing, powerful tool for typesetting documents, managing references and bibliographies, and displaying mathematical equations and figures. It is the gold standard for scientific writing, allows a degree of control on the part of the writer that makes most text editors look like finger painting, and is dense and complex to the point of being completely opaque to the average person. The result of this complexity is that the overheads involved in editing LaTeX documents are larger than for most other formats. As one of the major barriers to good documentation is the inconvenience of keeping it up to date, LaTeX is, in almost all cases, not a good choice.

A possible exception to this, as we will cover in more detail below (see Section 5), is the use of automatic documentation generation tools. These can often export content directly to LaTeX, removing some of the hassle entailed. However, as we will discuss in Sections 3 and 5 generators can only cover the most technical level of end-user documentation, and while the less technical documentation requires less frequent edits by dint of being more abstracted from the code, having it be in LaTeX for consistency still entails additional overheads. Additionally, generators can usually also export to more convenient formats such as HTML.

If you are already familiar and comfortable with LaTeX and would rather use it for documentation rather than learn how to use literally any other tool, or have a specific need for LaTeX to be the format you use, then we would strongly recommend finding a nice simple template and using as little of the LaTeX-specific tools as is possible. As with Markdown, an editor with a live (or close to live) preview function such as Overleaf is an invaluable tool.

4.2.2 Editors

Or rather editor. By and large there have been two options for LaTeX editing that, in our opinion, have stood head-and-shoulders above the competition: Overleaf and ShareLaTeX. However, as of 2018 any discrepancy between these options has been resolved as they have merged to create Overleaf v2.

Overleaf is free (unless you need additional storage space), provides automatic syntax highlighting, simple formatting buttons (that can be ignored if you don't need them), and a live side-by-side preview of the rendered document. It therefore provides all the features one would require of an editor, and while it is browser based and therefore requires an internet connection, it features persistent online storage rather than depending on browser storage. Overleaf projects can be shared with a simple link, sharees can be granted access either to view the current preview of the document or to edit the source files themselves. Changes, and who has made them, are tracked and can be reviewed at your leisure, and comments can be left to aide communication between collaborators. It also offers free, community sourced templates for a huge range of common document types. If you're writing in LaTeX, you will probably find it easier with Overleaf.

5 Automatic Documentation Generation Tools

Documentation is, as we have discussed above, both essential and also a potentially time-draining overhead on top of the existing demands of development. Unsurprisingly, tools have arisen with the intent of addressing this imbalance by automating, if only in part, the process of creating documentation.

5.1 Why and How to Use Them

As discussed above, bare code can only ever describe its functionality, not its intent, and therefore any information not encapsulated by the behaviour of the code must be added by the developer. Equally, the user experience cannot be extrapolated directly from the code source, so tutorials and how-to guides must be written by humans.

While automatic documentation generation tools cannot, and are not intended to, address these points, they do provide 2 significant advantages:

1. They are far more effective than humans at accurately, objectively, and consistently analysing large quantities of code. Tasks which would be arduous and error-prone for a developer such as mapping out large source distributions, constructing dependency graphs, inheritance diagrams, and collaboration diagrams, can all be done quickly and easily with near perfect accuracy. This has obvious advantages both for documentation and for development.
2. Documentation is extracted directly from the sources, meaning that a change in the documentation will always be reflected in any documentation that is subsequently generated. This makes it simple to keep the documentation up to date with the source.

Generators tend to be capable of outputting to a variety of formats, including LaTeX and HTML. This raises possibilities for further time savings, for example if your documentation is hosted online, you can use commit hooks with your git repo to have the documentation automatically update when you commit to certain branches.

For generators to work correctly, the code they are operating on must be commented in a way that they can interpret. By a shockingly convenient coincidence, this aligns closely with general good practice in commenting code.

As an example, consider the following code.[1]

```
/*! A test class */
class Afterdoc_Test
{
    public:
        /** An enum type.
        * The documentation block cannot be put after the enum!
        */
        enum EnumType
        {
            int EVal1, /**< enum value 1 */
            int EVal2 /**< enum value 2 */
        };
        void member(); //!< a member function.
    protected:
        int value; /*!< an integer value */
};
```

While the *content* of these comments is all but useless, being utterly degenerate with even a cursory reading of the code itself, they serve as useful demonstration of how these comments can be interpreted.

The HTML formatted output generated by Doxygen for this code snippet is included below. As we're sure you can see, it is still not *hugely* reader friendly, and would turn off the uninitiated just as fast as would the code snippet it interprets. What it achieves is to take the information contained within the comments (and some properties of the code itself) and collate them in a way that can be quickly scanned through and consulted.

For convenience, we have highlighted the content replicating the comments in red.

[1] Example taken from https://www.doxygen.nl/manual/docblocks.html

Afterdoc_Test Class Reference

#include <afterdoc.h>

Public Types

enum EnumType { EVal1, EVal2 }

An enum type. More...

Public Member Functions

void member ()

a member function.

Protected Attributes

int value

Detailed Description

A test class

Member Enumeration Documentation

EnumType

enum Afterdoc_Test::EnumType

An enum type.

The documentation block cannot be put after the enum!

Enumerator	
Eval1	enum value 1
Eval2	enum value 2

Member Data Documentation

value

int Afterdoc_Test::value

an integer value

The documentation for this class was generated from the following file:

- afterdoc.h

5.2 Overview of Common Tools

Somewhat inevitably in a field as fast-paced as software development, there is no clear "best" tool. In the table below we present an overview of some of the more commonly used pieces of software, and some of the positives and negatives that accompany them.[2]

EDITOR	LANGUAGES	PROS	CONS
DOXYGEN	C, C++, C#, objective C, Python, VHDL, Fortran, IDL, Java, PHP, D	• Outputs to HTML, RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages • Can generate various diagrams and graphs illustrating interrelations within the source • Can generate browsable version of code cross-referenced with documentation • Can do the above for undocumented sources. • Widely used and well supported.	• Formatting is very specific, so converting into or out of Doxygen-compatable formatting can be time consuming. • Docs tend to be visually cluttered. • Little capacity for non-technical documentation, i.e. notes, examples, rationale, etc.
GHOSTDOC	C#, VB, Javascript	• Strong emphasis on visual editing – edit documentation directly with changes propagated back into source code. • UI tools for XML formatting.	• Requires Visual Studio. • Free version is limited with more expensive options available for more features ($60-100 per user)
JAVADOC	Java	• Already part of the Java 2 SDK.	• Lacks diagrammatic functionality. • Need to manually enter HTML tags.
PDOC	Python 2, 3	• Quick and simple setup.	• Python-specific.
PYDOCTOR	Python 2	• Particularly good inheritance tracing. • Clean, readable output. • Can pass resulting object model to Sphinx for nicer output stype.	• Python 2 – specific.
SPHINX	C, C++, Ada, Fortran, PHP,	• Clean, modern formatting of output documentation.	• Lacks the useful ability to extract API

[2] This list is by no means exhaustive, so if you just want to find all the generators that might be viable for your project we recommend wikipedia's surprisingly comprehensive overview: https://en.wikipedia.org/wiki/Comparison_of_documentation_generators

	Python, Ruby, JavaScript	• Outputs to HTML, LaTeX (for printable PDF versions), ePub, Texinfo, manual pages, plain text • Supports reStructuredText in docstrings.	documentation from C++ headers. • Setup requires multiple configuration steps even with the provided quickstart script.
SWAGGER	>40: android, aspnet5, aspnetcore, async-scala, bash, cwiki, csharp, cpprest, dart, …	• Incredibly broad language support. • Open-source. • Ensures OpenAPI Specification compliance. • Automatically generate server and client code and SDKs. • Interactive UI allowing documentation editing and live API calls allows for interactive tutorials/examples.	• Tight focus on API development. • No hypermedia support. • Limited customization.

5.3 Recommendation

For projects using and of the C family of languages, Doxygen is probably your best bet. Its wide usage and support, and the ability to generate visual depictions of interrelationships within the source makes it a powerful aide in a broad range of circumstances.

For Python-centric projects, Sphinx is a clear front-runner. Its shares many of Doxygen's capacities while providing a cleaner output with a greater capacity for annotative documentation beyond the strictly technical. The only caveat is that the setup is a little on the complex side, and for simple projects this may be overkill. In these cases, a less involved tool such as pdoc may be a better fit.

For Java projects, Javadoc may seem attractive due to being bundled with the Java SDK 2, however in our opinion Doxygen has the edge due to its graphical capabilities, more minimal requirements for commenting, and the fact that experience using it is applicable to languages other than Java.

6 Summary

- Software without documentation is unlikely to be used.
- Code comments are a must, since without their inclusion crucial context and design logic is lost.
    - Comments should be at least one level more abstracted that the code they describe, and should describe the intent of the code.
    - Comment at the module level to present a high-level overview of the module's functionality.
    - Procedures/methods should also be commented, describing their specific functionality, data types, and brief instructions as to how to use them
    - Comments describing a particular line or block are required *only* where the complexity of the code means that it would be arduous for the reader to interpret, or where they contain pertinent information that is absent from the code.
    - Make including authorship and licensing information a habit.
    - Pick a style guide early and stick to it.

- End-User documentation should guide a user with no pre-existing technical knowledge to a state where they can comfortably use your software.
    - Tutorials are a jumping on point, guiding the user step by step in very granular detail to carrying out basic tasks with the software.
    - How-to Guides provide recipes for applying the simple techniques taught by tutorials to accomplish a much wider range of potential goals.
    - Reference Guides provide a technical reference for each accessible function, method, API, etc. and are intended to be consulted only when needed by accomplished users.
    - Explanations can be scattered throughout and provide in-depth discussions of supporting topics.
- Markdown is a convenient tool for quickly and easily writing documentation that will display correctly and clearly in multiple renderers and contexts.
- Automatic documentation generation can create reference guides from properly documented code, but are most suitable for technical documentation.

Acknowledgements

This work made use of computational support by CoSeC, the Computational Science Centre for Research Communities, through its Software Outlook Activity.

Appendix I. Overview of Markdown Editors

I.i Desktop Editors

Most of us spend most of our time with a single device, and it is therefore convenient to have locally installed tools at your disposal. The editors below all provide the capacity to preview markdown documents with a broad selection of additional bells and whistles for the discerning documentarist.

EDITOR	PLATFORMS	PROS	CONS
VIM	Windows, MacOS, Linux, MS-DOS, Amiga, OS/2, Android, i/OS, QNX, Agenda, Cygwin, Open VMS, MorphOS, probably other	• Utterly ubiquitous • Lightweight, little to no dependency to worry about. • Bundled with most Linux distros and MacOS by default. • Highly customisable, enormous range of shortcuts and mappable commands. • Plugins allow for automation of parts of your workflow. • Runs in terminal – seamless transition from file management to editing. • Free.	• Steep learning curve. • Documentation has a tendency to assume a lot of knowledge on the part of the user, jargon and abbreviations are common. • Enormous range of options can be overwhelming. • A different paradigm - Keybindings and operations that are common to most editors are completely different.
EMACS	[TODO]	•	•
TEXTS	Win, MacOS	• Shallow learning curve - displays rendered version of the document with highlight-and-apply formatting similar to Word or Pages. • Tools to convert to HTML5, PDF, ePub, Word, or presentation formats. • Integration with reference management applications, BibTex bibliography support. • More difficult to directly edit markdown code.	• No Linux support • Expensive – Texts costs $19 per user. • Requires Pandoc to be installed. • No file management system.

TYPORA	Windows, MacOS, Linux	• Integrated file management supporting cloud services such as Google Drive and Dropbox. • Seamless live preview of rendered document with mouse-over initiating a view (and editing functionality) of the raw markdown code. • CSS customisable themes	• Currently in Beta – features may change towards completion. • Free only during the Beta.
HAROOPAD	Windows, MacOS, Linux	• Vim key binding. • Syntax highlighting for >100 languages. • Side-by-side editor and viewer. • Flowchart drawing tools. • Free, but see below.	• Technically in Beta, however progress appears to have stalled for several years. Unlikely to be well-supported. • May become paid if completed.
MARKDOWNPAD2	Windows	• Widely used (on Windows). • Side-by-side editor and viewer. • Free. • CSS customisable themes.	• Windows only • Not actively maintained since the release of markdown extensions for VSCode.
VISUAL STUDIO CODE	Windows, MacOS, Linux	• While not natively supported, markdown extensions can be installed in a few clicks directly from the application. • Extensions provide a high degree of customisability of experience, everything from live preview, live preview in browser, syntax highlighting, spellchecking, formatting suggestions, etc. can be added or removed as needed. • Can also integrate seamlessly with Git. • Similar interface to the widely used Visual Studio – will feel very	• Steeper learning curve as the user needs to be able to find and install extensions before even beginning to make use of markdown functionality.

		familiar if your already used to the latter. • Decent chance that you’re already using VSCode to develop.	
SIMPLEMDE		• Free. • Open Source. • Extensive set of features. • Embeddable.	• Requires you to essentially build or modify your own editor to make use of it.
SUBLIME TEXT 3	Windows, MacOS, Linux	• Windows, Mac OS, and Linux support. • Broad range of useful features for programmers.	• Markdown support is not native • Complex installation process - requires Sublime Package Control and the Markdown Editing Package. • $70 per user.
NOTEPAD++	Windows, MacOS, Linux	• Free • Frequent updates • Highly convenient if you’re already using Notepad or Notepad++	• Less developed as a documentation / software development tool than the other entries on this list
INKDROP	Windows, MacOS, Linux, IOS, Android	• Windows, Mac OS, and Linux support as well as IOS and Android. • Side-by-side live preview. • Code and syntax highlighting. • Key customizations.	• $4.99/month

I.ii Browser-Based Editors

For those occasions where one is separated from one’s beloved machine, there are browser-based tools that can provide the requisite functionality. These have the advantage of being available so long as you have an internet connection, at the cost of a usually more limited suite of functions and often a dependence on browser storage.

EDITOR	PROS	CONS
DILLINGER	• Split-screen preview by default with scroll-sync. • Exports to HTML, Styled HTML, Markdown, PDF. • Direct uploads to GitHub, Google Drive, WordPress, etc.	• Lacks of spell checking.
EDITOR.MD	• Minimalist interface.	• Minimalist Interface

GITHUB	• Real-time saving. • Integrate with (and commit to) your repo. • Broad syntax highlighting options. • Supports Github-specific markdown features.	• Specifically GitHub flavour, limited support for broader flavours.
STACKEDIT	• Broad syntax highlighting options • Easy-to-use interface, handy formatting buttons • Customizable themes • Direct uploads to GitHub, Google Drive, WordPress, etc. • Merge tools • Offline functionality	• Depends on browser's local storage. • Using offline functionality requires the browser and website to have been opened while online. • Publishing to GitHub requires write access to repos, which presents a potential security risk.