Code Review Reference

Originally developed by Mike Jackson

EPCC, The University of Edinburgh

michaelj@epcc.ed.ac.uk

8 December 2017

1 Introduction

Wilson et al. (1) propose a set of Best Practices for Scientific Computing. From these we can pull out a set of questions that we can consider when reviewing code. In this guide, we list these questions, along with additional information, complementing that of Wilson et al. and also relating to documentation. Links to tools to help with code reviewing are also given.

2 Questions for a code review

Is the program broken up into easily understood components, each of which conducts a single, easily understood, task?

Components can include functions, methods, classes, modules, and files. The degree to which such components conduct single, easily understood, tasks is termed cohesion (https://en.wikipedia.org/wiki/Cohesion_(computer_science)).

Are the components loosely coupled?

Coupling (https://en.wikipedia.org/wiki/Coupling_(computer_programming)) refers to the dependencies a component has on other components. Ideally, we seek a loose coupling, so that changes to one component incur minimal, ideally no, changes to other components.

Are names consistent, distinctive, and meaningful?

When implementing mathematical formulae from a paper, we may want to use variable names matching the terms as defined in the paper (e.g. phi, Re, dt etc.) Complementing such an implementation with a comment that provides a citation to the paper helps the reader understand that the names are defined like this for a reason (and not because we're being cryptic or poor developers!)

Is code style and formatting consistent?

While indentation is ignored by C/C++ and Java compilers, developers use indentation to determine what code belongs where, not where the brackets are.

A coding standard is a specification of how code should look, including naming, formatting and use of whitespace. Coding standards can help promote a consistent style and formatting within a project and across projects. Many projects and organisations have coding standards including:

- GNU's Coding standards (http://www.gnu.org/prep/standards/).
- UK Met Office's Fortran 90 Standards (http://research.metoffice.gov.uk/research/nwp/numerical/fortran90/f90_standards.html).
- Google's Java style guide (https://google.github.io/styleguide/javaguide.html).
- Google's C++ style guide (https://google.github.io/styleguide/cppguide.html).

Some languages have generic coding standards including:

- Python community's PEP 0008 – Style Guide for Python Code (https://www.python.org/dev/peps/pep-0008/).
- Java – Oracle's Code Conventions for the Java Programming Language (http://www.oracle.com/technetwork/java/codeconvtoc-136057.html).

Are constants defined exactly once to ensure that the entire program is using the same value?

Does the program have no duplicated code, code that has been copied and pasted?

Does the program reuse existing libraries and packages (e.g. for numerical integration, matrix inversions etc.) instead of re-implementing these?

Does the program use assertions, or perform other checks, to ensure that inputs are valid, outputs are consistent, etc?

Is the program complemented by automated tests?

Are components' interfaces documented?

Documentation of interfaces includes the purpose, inputs, outputs, exceptions, error codes of packages, modules, classes, methods, and functions, everything that describes how a component can be used by another component.

Do comments document reasons why the code has been implemented as it has, not the implementation itself?

Is the documentation embedded in the program?

Tools have been developed to allow structured comments to be written. From these comments, documentation can be automatically generated in HTML, LaTeX or other formats. This documentation can help developers to understand your code without having to look at your source code. These tools include:

- Doxygen (http://www.stack.nl/~dimitri/doxygen/) for C, C++, Fortran or Python.
- Docstrings (https://www.python.org/dev/peps/pep-0257) or Sphinx (http://sphinx-doc.org) for Python.
- JavaDoc (http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html) for Java.

Many others are available, see, for example, Wikipedia's Comparison of documentation generators (http://en.wikipedia.org/wiki/Comparison_of_documentation_generators).

Certain REST frameworks, such as Django (http://www.django-rest-framework.org/), also support auto-generation of API documentation for REST-based APIs.

Is there user documentation which describes how to use the program?

There is a distinction between describing what a program does, how it does it and how to use it.

Does the documentation describe the dependencies it needs and their versions?

Version information is important. Different versions of software, models, tools, libraries, data formats, protocols, interfaces, services, databases, operating systems, languages and browsers, can differ in terms of syntax, behaviour or content. Software written to use one version might not be compatible with earlier or later versions. For example, Python code with the statement `print 'hello'` will not run under Python 3, but code with the statement `print('hello')` will run under both Python 2 and 3. As another example, Scientific Linux 7 or Ubuntu differ in the packages they support.

3 Other issues

Depending upon the purpose of the code review there may be other issues or concerns we may want to identify when reviewing code. For example, Cohen (2) suggests the following categories:

- Algorithm
- Build
- Data Access
- Documentation
- Error-Handling
- Interface
- Maintainability
- Performance
- Robustness
- Style
- Testing
- User Interface

4 Code analysis tools

There are numerous style checking and code analysis tools available which can both review code (for compliance with coding standards for example) and also identify opportunities for redesign. Some of these will also automatically refactor code to correct any stylistic or structural issues. Many integrated development environments also provide these features.

For C/C++:

- Clang Static Analyzer (http://clang-analyzer.llvm.org/), part of Clang (http://clang.llvm.org/).
- ClangFormat code formatter (http://clang.llvm.org/docs/ClangFormat.html), also part of Clang.
- Cpplint (https://github.com/google/styleguide/tree/gh-pages/cpplint) checks code conforms to Google's C++ style.

Python:

- pep8 (https://pypi.python.org/pypi/pep8) checks conformance to Python PEP 0008 style guide.

- pyformat (https://pypi.python.org/pypi/pyformat) reformats code to be PEP 0008-compliant.
- pylint (http://www.pylint.org/).
- PyCharm (http://www.jetbrains.com/pycharm/index.html) IDE.

Java:

- CheckStyle (http://checkstyle.sourceforge.net/).
- Eclipse (https://www.eclipse.org/) IDE.
- google-java-format (https://github.com/google/google-java-format) reformats code to conform to Google's Java style.

Many others are available, see, for example, Wikipedia's List of tools for static code analysis (https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis).

5 How not to lose friends and alienate people

Code is personal work and receiving comments on it can be difficult (as can giving comments). It is important to depersonalise a code review and never attack personal choices in the code, or the author of the code. Code reviews should judge if the code is "good enough", even if it is not written in the way the reviewer would write it. (3) provides advice on giving a constructive, kind, balanced and useful review.

6 References

1. Wilson, G., Aruliah, D.A., Brown, C.T., Chue Hong, N.P. and Davis, M. "Best Practices for Scientific Computing", PLoS Biol 12(1): e1001745, January 2014. doi:10.1371/journal.pbio.1001745. http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001745.
2. Cohen, J. "Crash Course in Lightweight Code Review", March 2009. http://www.drdobbs.com/architecture-and-design/crash-course-in-lightweight-code-review/215800147.
3. Gruel, N., Walker, A., Knight, V. and Jackson, M. "Constructive code critique", The Software Sustainability Institute blog, March 2017. https://www.software.ac.uk/blog/2017-05-11-constructive-code-critique.