Code Inspection Checklist

– Explanations –

For longer explanations, see https://terranostra.one/posts/Principles-of-Software-Development.html

- *Is the code properly formatted?*

  Use indentation to show the level of the current function/conditional/loop body. Lines should be no longer than 100 characters at the most. If a statement is longer than that, set a line break where it makes syntactic and semantic sense.

- *Are variables clearly and succinctly named?*

  Names should be easy to understand, but not too long to type. Use camelCase or underscored_names for multi-word variables (different languages have different conventions).

- *Is the code split up into functions and files of appropriate lengths?*

  Use functions! The most effective functions are probably between 5 and 20 lines of code long. If a function doesn't fit on your screen, break it up. For larger projects, group related functions in files.

- *Is code understandability aided by suitable comments?*

  Comments should primarily be used to explain the *Why* of an implementation, not the *How*. They can also be used as headers for sections of code. Avoid using them to explain difficult code, refactor the code instead to make it easier to understand.

- *Are there no repetitive elements that should be refactored?*

  If you find yourself typing anything more than twice, use a loop or stick it in a function. Avoid repetitive code – it's error-prone and bothersome to maintain.

- *Are there no hard-coded values that should be either named constants or configurable parameters?*

  Hard-coded values may be things like numeric factors or pathnames that are explicitly embedded into the code. This can make the code harder to understand and less adaptable. Avoid hard-coding values by instead storing them in a named constant (ideally in a central location), or making them a configurable parameter that can easily be changed.

- *Does the software check for bad input (defensive programming)?*

  If you ask for user input, make sure the user gives you what you want (e.g. a number where you want a number, or a string without special characters). Also check that external resources of your software (such as input files or network streams) exist before you use them. *Look before you leap!*

- *Are libraries employed where useful, but not unnecessarily?*

  Don't reinvent the wheel, but don't create too many dependencies by relying on other people's libraries for what you could easily do yourself. Dependencies make it harder to get a program up and running on a new system, and may break in future.

- *Does the design make good use of encapsulation and abstraction?*

  Encapsulation means to compartmentalise your code into small, maximally self-contained subsystems. These are easy to develop and debug, and reduce the overall complexity of your software. Subsystems can be constructed on the level of functions, classes, files, or packages.

  Abstraction means to split a system up into layers, and to construct each layer using the building blocks of the layer below. For example, to control a pixel screen, you might have one layer to interface with the hardware and set individual pixels, another layer to draw lines and generic shapes, and a third layer to draw common polygons – whereby each layer calls only the functions provided by the layer below.

- *Is the user interface easy to understand and use?*

  Is it self-explanatory? Is there an inbuilt help function? Does it follow conventions, i.e. does it do what the user expects it to do? Can the user achieve his/her aim with a minimum of effort? Are there sufficient, but not too many features?

- *Is there sufficient documentation to allow an external person to use and modify the software?*

  At the very least, there should be a README file in the top-level directory, explaining how to get the software to run. Further documentation can include a HACKING file on the overall architecture, or HTML documentation of the API generated automatically from source code comments.

- *Can the program be automated?*

  Is it possible to automate the program using a shell script, or does it require interactive user input? Does it offer commandline flags and arguments? Does it accept configuration or input files? In the interest of parallelisation and reproducibility, scientific software should always be automatable.

- *Are there mechanisms for verifying correctness? (Error messages, logging, unit tests…)*

  Does the program emit informative error messages when it crashes, or detects bad input? Does it write a log file so its progress can be traced during and after a run? Is the verbosity level of the output configurable (e.g. using the levels “silent”, “only errors”, “standard”, and “debugging”)? For software requiring high reliability: Is there an automated test suite that checks all important system functions for introduced errors (regression testing)?