Research Software Best Practices

Rafael Mudafort

National Renewable Energy Laboratory

This is a temporary cover page. A final cover page will be generated by comms with current style and formatting.

v2.0.4, November 2022

Executive Summary

Wind energy researchers typically share one key characteristic: a passion for increasing wind energy in the global energy mix. The U.S. Department of Energy (DOE) supports this mission in a number of ways including allocating funding directly to various aspects of wind energy research through the Office of Energy Efficiency and Renewable Energy (EERE) via the Wind Energy Technologies Office (WETO). While the traditional output of research is academic publication, software development efforts are increasingly a major focus. Software tools in the research environment allow researchers to describe an idea and quickly increase the scope and scale as they study it further. As a product of research, these tools represent a direct pipeline from researcher to industry practitioners since they are the implementation of ideas described in academic publications. Given this vital role in wind energy research and commercial development, the broad research software portfolio supported by WETO must maintain a minimum level of quality to support the wind energy field in the growing transition to renewable energy. **This report outlines a series of best practices to be adopted by all WETO-supported software projects, as well as expectations that the communities interacting with these projects should have of the developers and tools themselves.**

Wind energy research software has a unique standing in the field of scientific software. The stakeholders are varied with a subset being:

- DOE EERE leadership
- DOE WETO leadership and program managers
- National lab leadership
- Associated project principle investigators
- Research software engineers
- Wind energy researchers in academia (including graduate students, post docs, and national lab staff)
- Industry researchers and practitioners
- Commercial software developers
- The general public interested in wind energy

These software are typically the end-user of other generic software libraries, so the funding cycles are often tied to applied research rather than the development of the software itself. Since the developers are also wind energy researchers, these tools are typically designed in a way that closely resembles the application in which they're used. Additionally, the expertise and incentives for the developers have a high variability, and often neither are aligned with software engineering or computer science.

Given the unique environment in which wind energy research software is produced and consumed, it is critical for model owners to understand the context of their software. A framework for developing this understanding is to answer the following questions of a given software project:

- What is it's purpose?
- What is it's role in the field of wind energy?
- What is the profile of the expected users?
- For how long will it be relevant?
- What is the expected impact?

These questions allow model owners to identify the appropriate methods for the design, development, and long term maintenance of their software. Additionally, the answer provide context for future planners to understand why particular decisions were made and discern the consequences of changing course.

The information is aggregated from experience within WETO-supported software development groups as well as external organizations and efforts to define the craft of research software engineering. These best practices aim to make the collaborative development process efficient and effective while improving the model understanding across stakeholders. Additionally, the general adoption of a common framework for software quality ensures that the end users of WETO software can trust these tools and accurately understand the risks to workflow integration.

Table of Contents

List of Figures

1 Accessibility

Accessibility is concerned with how practitioners are expected to obtain and integrate a software into their processes. The product that is to be obtained is the executable version of the software. In the case of compiled programming languages, this is a binary executable or library file, while interpreted languages typically require distributing the source code directly.

For guidance on developer accessibility, see **??**.

Summary:

- Determine the expected or targeted user profile up front and address accessibility accordingly
  - Domain experts will need less hand holding than graduate students or the general public
- Automate accessibility

1.1 Prerequisite knowledge

Using a computer in a scientific context is a learned skill and requires years of practice to become proficient. Tools like a "terminal", "shell", or "command prompt" are not initially intuitive, and that these three terms are used interchangeably can lead to further confusion. This is an example of a barrier to entry often encountered by early-career researchers and experienced practitioners alike. In order to improve accessibility, it is important to understand the experience of users and design software to meet their needs.

BP: Identify target user profiles and anticipate their levels of understanding. Accurately understand the complexity of the systems used to access the software, and evaluate whether this matches the expected skills in target users.

Some examples of common barriers to entry are:

- Navigating a "terminal"
- Knowledge of acronyms, jargon, or interchangeable phrases
  - CLI, API, IDE, etc
  - Compile, clone, check out
  - Terminal vs shell vs command prompt
- Extensions: `.exe`, `.so`, `.dll`, `.dylib`
- Installation:
  - Package managers
  - Downloading executable files
  - Configuring an environment

1.2 Distribution

Research computing software often depend on third-party libraries, and many of these dependencies are research software themselves. Therefore, the installation and environment configuration for this type of software can easily become complex.

BP: It is the responsibility of developers to provide a streamlined method of installation using common software distribution systems and automation as much as practical.

Mature package managers are a great resource since they have a distribution system already in place and manage dependencies between software tools.

The ecosystem of open source software package managers has coalesced around a few primary tools:

- Python Package Index (PyPI)
  - Source and binary distribution package manager for Python software
  - Platform: any
- Conda
  - Package, dependency and environment management for any language
  - Platform: any
- Conda-forge
  - A community-led collection of recipes, build infrastructure and distributions for the conda package manager
  - Platform: any
- Homebrew (brew)
  - The Missing Package Manager for macOS (or Linux)
  - Platform: Ubiquitous for macOS, but also available for Linux
- Spack
  - Spack is a package manager for supercomputers supporting any language and distributable product
  - Platform: Ubiquitous for Linux-based supercomputers, and also available for macOS and Linux
- APT
  - A user interface that works with core libraries to handle the installation and removal of software on Debian, and Debian-based Linux distributions
  - Platform: Ubiquitous for Linux for system-level or generic packages
- Fortran package manager (FPM)
  - Fortran-specific executable and library package manager.

The process for including a package in a package management system varies, but all are designed to integrate with automated systems to prepare and distribute the package automatically upon a given event. The practice of releasing a software package after a tagged release (see **??**) or requisite set of changes is called "continuous integration", also known as "CI". Tools for this level of automatic are common, and a practical choice is GitHub Actions. A typical CI pipeline for a Python package is shown below where the square components are GitHub Actions steps. Note that this pipeline includes sub-system areas called "Continuous Testing" and "Continuous Deployment".

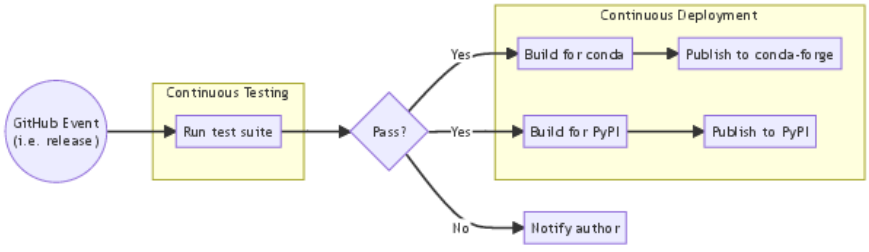

Figure 1. TODO

2 Usability

Usability is concerned with how practitioners are expected to execute the software including creating inputs and managing outputs.

Nuances of research software: - "Design" is typically not a consideration at all - It is typical to adopt "patterns", so there is very little evolution of software interface design and therefore usability - Research software is expected to be predictable and similar to what users already know. The challenge is to understand the existing paradigms and adopt them well.

2.1 User interface

The user interface (UI) is any mechanism through which users interact with the software typically by providing inputs and receiving outputs. Examples of UI's include:

- Graphical user interface (GUI)
- Web-based front ends
- Input and output files
- Command line interface
- Library API's

WETO software UI's should be well defined and predictable. They should adopt the conventions that already exist in the environments and contexts in which they're used. Most importantly, all user interfaces should be well documented.

2.1.1 Command line interface

The command line interface (CLI) is one type of front-end for software. It is the method by which a software is executed via a computer's terminal. WETO software should in general adhere to the following conventions and principles for CLI's. However, these are guidelines and can be skipped when context is clear or another option improves usability.

- Adopt command line syntax requirements from https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html
  - Guideline 1: Utility names should be between two and nine characters, inclusive.
  - Guideline 2: Utility names should include lowercase letters (the lower character classification) and digits only from the portable character set.
  - Guideline 3: Each option name should be a single alphanumeric character (the alnum character classification) from the portable character set. The –W (capital-W) option shall be reserved for vendor options. Multi-digit options should not be allowed.
  - Guideline 4: All options should be preceded by the '-" delimiter character.
  - Guideline 5: One or more options without option-arguments, followed by at most one option that takes an option-argument, should be accepted when grouped behind one – delimiter.
  - Guideline 6: Each option and option-argument should be a separate argument, except as noted in Utility Argument Syntax, item (2).
  - Guideline 7: Option-arguments should not be optional.
  - Guideline 8: When multiple option-arguments are specified to follow a single option, they should be presented as a single argument, using <comma> characters within that argument or <blank> characters within that argument to separate them.
  - Guideline 9: All options should precede operands on the command line.

  - Guideline 10: The first −− argument that is not an option-argument should be accepted as a delimiter indicating the end of options. Any following arguments should be treated as operands, even if they begin with the − character.
  - Guideline 11: The order of different options relative to one another should not matter, unless the options are documented as mutually-exclusive and such an option is documented to override any incompatible options preceding it. If an option that has option-arguments is repeated, the option and option-argument combinations should be interpreted in the order specified on the command line.
  - Guideline 12: The order of operands may matter and position-related interpretations should be determined on a utility-specific basis.
  - Guideline 13: For utilities that use operands to represent files to be opened for either reading or writing, the − operand should be used to mean only standard input (or standard output when it is clear from context that an output file is being specified) or a file named "-".
  - Guideline 14: If an argument can be identified according to Guidelines 3 through 10 as an option, or as a group of options without option-arguments behind one − delimiter, then it should be treated as such.
- Adopt these minimum GNU conventions
  - A short version with one dash and a long version with two dashes
  - −v / −−version to show version information
  - −h / −−help to display help information
  - −i / −−input for input file specification
  - −o / −−output for input file specification
  - −V / −−verbose to include additional output in terminal
  - −q / −−quiet to suppress terminal output
- Use context-specific switches
  - Unix: − or −−
  - Python: − or −−
  - Windows command prompt: /

2.1.2 Input and output files

The ecosystem of tools for processing data files is vast and mature. Therefore, input and output files should adopt a common file type and syntax relevant to the field and context of the software itself. For example, large datasets generated by computational fluid dynamics software are often exported in HDF5 format since robust libraries are available to export the data and load it into post-processing tools. Similarly, input files should retain a ubiquitous human-readable format such as YAML as this allows users to generate input files programmatically using standard libraries. Input and output files required by WETO software should adhere to the following conventions and principles.

- Simple, clear, and predictable structure
- Expressive and concise
- Easy to produce and consume using ubiquitous software tools
- Minimal data consumption
  - For large data sets, option to split into smaller files or binary format
- Typical and predictable data types

2.2 Error messages

Messaging to practioners from within a software can be immensely helpful. At the same time, the infrastructure for communicating messages can be a heavy lift. It is important to find a balance of appropriate levels of messaging while also ensuring that the messages themselves are up to date with the software features and implementations. Too much messaging results in information overload and critical messages can be lost in noise. Additionally, messaging is another develop responsibility and can be overlooked among all of the other responsibilities during the development cycle.

Useful error messages:

- Expect that the reader does not have the context of the author
  - Include a stack trace in all messages
  - At minimum, include the calling function name
- Anticipate the needs of the reader
  - What will they be thinking about when this error pops up?
  - What will they need to do next?
- Include information that will help project maintainers understand the context of the problem
  - Include metadata where relevant; see **??**
  - Include the value of data that is found invalid

2.3 Metadata

Tracking metadata in software projects is a simple way to provide clarity to all users. This greatly improves usability and has the added effect of improving the debugging process. This information can be provided to the user in any structured output from the software. For example, output data files, reports, images, etc can all include a snapshot of the metadata. The objective is to communicate information on the state of the software (version and runtime), the state of the computing environment, and any user decisions.

The following fields are minimum metadata to include:

- Version number in semantic versioning format (MAJOR.MINOR.BUGFIX, i.e. v3.2.1)
- Execution time
- Compile info, if applicable
  - Compiler vendor
  - Compile time
  - Compiler settings
- System information such as OS, relevant hardware (i.e. accelerators) vendor
- Relevant settings enabled

3 Extendability

Extendability is concerned with how improvements such as new features, bug fixes, and general maintenance are added to an existing software project. This covers both the technical aspects as well as the management of multiple developers and development efforts happening concurrently.

The lifecycle of WETO software typically follows a cyclical pattern of funding, development, and release, as depicted below. Note that the "Maintenance" tasks are usually optional and included in future development efforts. Therefore, it is critical to the life of all WETO software to prioritize extendability so that future funding opportunities are attractive to stakeholders and general maintenance and infrastructure upgrades can be introduced with minimal overhead.

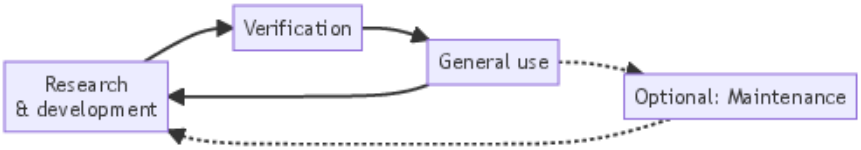

Figure 2. TODO

Is the code open source? If not, is there a build system with modern dependencies? i.e. Makefiles are outdated, use CMake

This is closely tied to **(6) methodologies for communicating intent and verifying implementation** Generally, use ubiquitous infrastructure

- Project managers can rely on mature third party tools to handle complex systems
- Mature third party tools typically come with their own documentation

3.1 Code style

In software development, the word "grok" is often used (see usage in Hacker News, Lobsters, StackOverflow) to communicate about degrees of understanding. This word is described by it's creator below.

> *Grok* means "to understand", of course, but Dr. Mahmoud, who might be termed the leading Terran expert on Martians, explains that it also means, "to drink" and "a hundred other English words, words which we think of as antithetical concepts. 'Grok' means *all* of these. It means 'fear', it means 'love', it means 'hate' – proper hate, for by the Martian 'map' you cannot hate anything unless you grok it, understand it so thoroughly that you merge with it and it merges with you – then you can hate it. By hating yourself. But this implies that you love it, too, and cherish it and would not have it otherwise. Then you can *hate* – and (I think) Martian hate is an emotion so black that the nearest human equivalent could only be called mild distaste.

Source: https://en.wikipedia.org/wiki/Grok.

That such a word exists and is widely used in software development illustrates the high value of clear and understandable code. WETO software should avoid complexity where possible and favor readability over writability. Strive to create software that can be easily grokked by developers who do not have the current context.

The designers of the Python programming language consider readability as a primary priority, and the most famous of the many Python language-development documents is PEP 8 which proposes a style guide for Python code. PEP 8 is summarized into 19 aphorisms (20 including one that's implied) and is referred to as "The Zen of Python". Much of the WETO software portfolio is Python-based, so these guiding principles directly apply. However, these principles are programming language agnostic and eloquently describe the paradigm for developing extendable software.

3.1.1 The Zen of Python

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

3.2 Version control

Version control, typically with `git`, is a tool for tracking the evolution of a project change by change establishing a history of changes. Each change is itself a version of the software, and they provide a snapshot of thought processes and progression of work.

BP: Craft a version control history that communicates the evolution of changes of the software to future developers including the author of current changes. BP: Evolve the software in a logical, linear process with digestible change sizes.

Version control with git is often a secondary consideration in the software development process. It can seem like simply a mechanism to "save" the state of a document. However, it carries far more meaning in the context of software extendability. Since the git system is decentralized, it allows for multiple developers to make changes to a project concurrently. Git also provides a mechanism for resolving differences so that multiple changes can be merged together easily.

In addition to the content of changes themselves, the connectivity between changes is valuable over the lifetime of a project. The connectivity between commits is structured as a directed acyclical graph (DAG). Each commit has a parent and each parent can have multiple children. This provides a mechanism for easily and accurately rolling back to the state of the project at any time in history.

To best leverage the power of git to enable extendability, consider the following guidelines:

- It is reasonable to spend time crafting each commit and a sequence of commits.
- Practice editing a series of commits to ensure that the progress of work is captured accurately.
- Consider whether the commit history is concise and readable to people who are not the authors.
- Become familiar with the following actions:
  - Interactive rebase
  - Cherry-pick
  - Squash
  - Edit a commit message

- Commit messages should be short, and it is a convention to limit them less than 50 characters.
- An additional line can be included as a longer description of the commit beneath the 50 character line. The second line is typically limited to 70 characters, but it is considered reasonable to use as much space as needed.

3.3 Collaborative workflows with GitHub

The processes through which developers interact with a software and other developers is an essential component of extendability. These processes should generally strive for efficiency while minimizing overhead. Automated processes are better than manual processes, and objective is better than subjective. The majority of collaborative software development processes occur on cloud-based resources on the GitHub platform.

BP: Plan and coordinate software development efforts into a collaborative workflow using GitHub

BP: Automate code quality feedback as much as possible via GitHub Actions

GitHub contains some key features for coordinating software development:

- Issue tracking
- Forum-style discussions
- Pull request and code review
- Project boards
- Releases

GitHub and git (see **??**) are tightly connected, but they are different systems and serve different purposes in the development process. Git is a version control system for tracking and merging changes to a software. GitHub is a platform for orchestrating and coordinating the various processes that happen around the development cycle. GitHub activities add context on top of the individual changes captured in commits. Whereas commits often capture low-level information, GitHub activities can map the low level details to high-level efforts.

Become familiar with GitHub features and leverage them to plan and communicate.

GitHub features can be used a many ways. The primary features are described below and a typical sequence of events across these features is described.

- Discussions: This is typically the starting point for any collaboration. Create a discussion topic and engage with other model stakeholders to define the idea and develop a proposed implementation.
- Issues: Document the proposed solution to a problem or implementation of a new feature as outlined in the corresponding Discussion. Finalize the description and outline test cases to verify the idea.
- Projects: Collect Issues, Pull Requests, and generic cards to establish a relationship across all ongoing works in progress. This is typically most useful for large development efforts and prioritizing work for upcoming releases.
- Pull Requests: Pull Requests (PR) are a request to accept a change into a branch. This typically happens across forks of a repository, but it can also happen between branches of the same fork. During the implementation of an Issue, open a pull request to communicate that work is ongoing. This is also the venue for code reviews.
- Releases: A number of accepted pull requests can be aggregated to comprise one release, and this is listed in a project's GitHub Releases page along with release notes to describe the changes and communicate relevant details.

Along with git, GitHub provides a helpful mechanism to capture design intent, factors that lead to particular decisions, and the evolution of a project for future reference. However, it is important carefully craft the messages to avoid washing out information with noise. The following are guidelines to consider when engaging on GitHub.

- Descriptions of any activity should be well scoped and easily understandable.

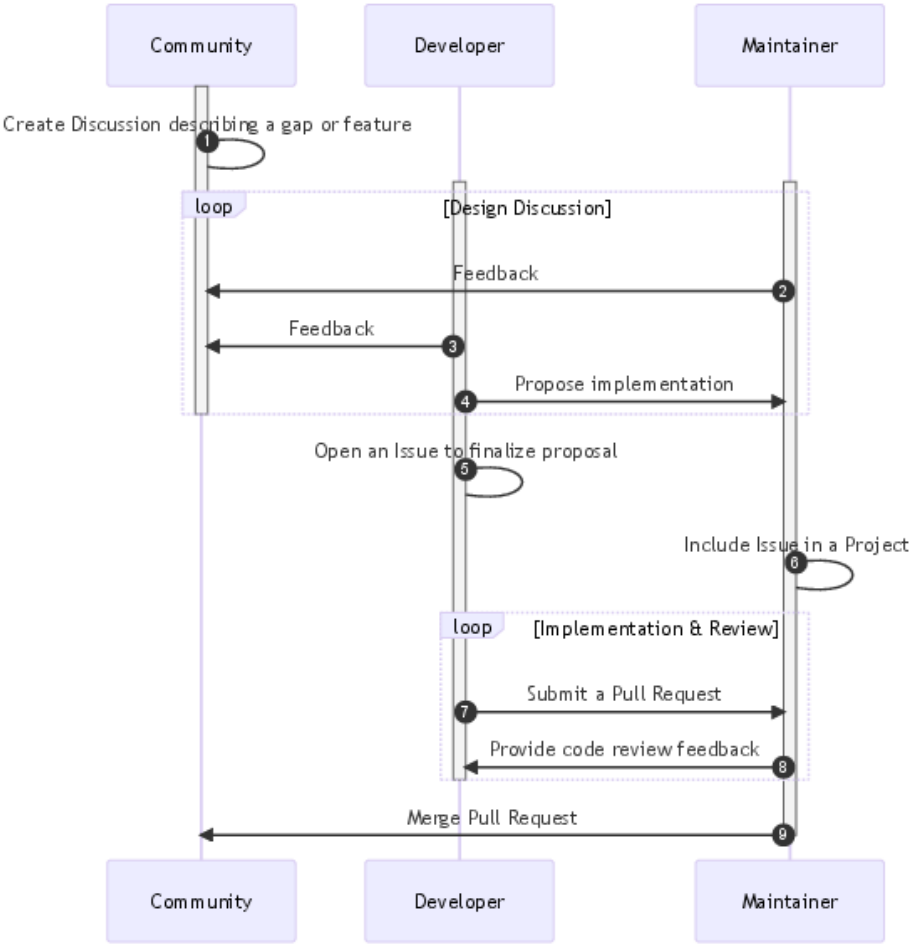

Figure 3. TODO

- Pictures really are worth 1,000 words. Always include a diagram, plot, screenshot, or picture when it will add clarity.
- Prefer actual text over of screenshots of text. GitHub is searchable, so text provides more searchable context whereas screenshots do not. Additionally, text-based code snippets can be copied easily by other users.
- Establish a practice of assigning responsibility for each Issue and Pull Request. Without a person to take ownership, these will remain unaddressed.

3.4 Pull Requests

A pull request is a request to merge a particular set of code changes into another copy of the software, typically an agreed upon "main" version.

Pull requests should include contextual information regarding the code change. The intention is to convince reviewers and maintainers that the new code is in a good state and that it's inclusion would be a benefit to the project. This typically involves a contextual description of the change and a description of why the change is valid and well tested.

Futhermore, GitHub automatically constructs release notes from all of the pull requests merged since the previous release. It automatically takes the titles of each pull request to construct the release notes. "Update XYZ" again provides no context and more work is required to communicate what has changed to users and downstream dependencies.

All version control messages (commits and pull requests) should communicate what the change accomplishes. Vague messages such as "updated solver" are distracting and lack meaning.

3.5 Automated Quality Checks - Continuous-N

Continuous Integration can be mean many things:

- Continuous building
- Continuous testing
- Continuous distribution
- Continuous integration (into the repository)

Appendix A. The engineer behind the research software

Research software exists in a unique environment. The breadth of experience in users and developers is more narrow than in other types of software, and the funding mechanisms are often tied to results from using the software rather than to the software tools themselves. Because of these nuances of the research software environment, the incentives to create high quality software are often lacking, and this can leave software developers with the difficult choice of choosing to prioritize with their work or career but not both. Therefore, it is important to directly consider the needs and expectations of the people responsible for designing and implementing research software projects.

The term research software engineer (RSE) is defined as:

> A Research Software Engineer (RSE) combines professional software engineering expertise with an intimate understanding of research. https://society-rse.org/about/

While modern research typically involves using research software, it is common for research software engineers to focus skill development on either the research domain or the aspects related to software engineering. Additionally, the research environment in academia and government research labs are structured to incentivize academic publication. Therefore, the resulting teams are often comprised of domain researchers and research software engineers. The domain researchers inform the needs of the research software and are the primary users. The RSE's design and develop the software systems as well as manage various IT responsibilities for the group such as creating computer-based workflows, managing data, constructing web-based research artifacts, and training colleagues on best practices in research computing.

In this context, note the difference between computer science and software engineering:

- **Computer science** is the study of computation, information, and automation. Computer science spans theoretical disciplines (such as algorithms, theory of computation, and information theory) to applied disciplines (including the design and implementation of hardware and software). https://en.wikipedia.org/wiki/Computer_science
- **Software engineering** is an engineering-based approach to software development. A software engineer is a person who applies the engineering design process to design, develop, maintain, test, and evaluate computer software. The term programmer is sometimes used as a synonym, but may emphasize software implementation over design and can also lack connotations of engineering education or skills.

  Engineering techniques are used to inform the software development process, which involves the definition, implementation, assessment, measurement, management, change, and improvement of the software life cycle process itself.

  https://en.wikipedia.org/wiki/Software_engineering

Treat a RSE as an engineer

- **Many have engineering or science degrees and treat their work as engineering.** Don't refer to them as "programmers" doing "programming".
- **Writing code and designing software systems are entirely different things, and the latter must be recognized relative to the value that it adds.** Software design and software architecture are very complicated, and the process of developing a design given various requirements is a design process. It involves stating requirements, iterative design, and validation and verification of the design. It often takes years to fully accomplish an objective and at the same time the landscape of computers and software development is changing. Additionally, software is rarely created by one person, so the RSE has to deal with managing multiple people making changes simultaneously while also trying to meet the needs of the project.

In addition to acknowledgement of work and value added, it is important to provide meaningful career guidance to RSE's to serve both the personal goals of associated staff and ensure that the projects have well-rounded contributors. RSE's should have some level of domain experience; that is to say that they should *use* as well as *develop* their software. RSE's should know the context in which their software exists. They should be experts in the implementation and very good in the usage. A typical career trajectory within the national lab environment look something like this:

- Year 1: Implementing models and developing tests and documentation
- Year 2: Second author on analyses, improved modeling, informing AOP, and writing developer-specific documentation
- Year 3: Lead author on analysis work, guiding the direction for future development projects, writing AOP, writing user-specific documentation
- Year 4: Proposing new work and seeking funding to expand the software project

In general, the amount of code written by a developer should peak around year 2 or 3 and then start to drop. It should not go to 0, but the majority of involvement in software development should be code review, design, and project planning.

A.1 Architecture

Support modularity

Design in such a way to allow for future flexibility

- Think about systems in discrete chunks rather than monoliths
- Consider how the system will scale in scope and in problem size

Design for usability

- Trap errors before they manifest into larger problems that are difficult to debug through validation of input files and function inputs
- Adopt a logging framework
  - Funnel messages to a particular place (terminal, file, nowhere) very easily
  - Format messages to call out particular pieces of information
  - Always include context!
    - Users and developers will not know what you mean by "Invalid value" unless they also have some context. Try this: "Invalid value in ClassName.well_named_function(); wind speed must be positive".
- Avoid information overload. It's too easy to ignore messages from a tool when there are too many words. Respect the user's attention, only give messages that are absolutely critical.
- Assume warnings will not be read-trap errors and inconsistencies instead.
- Avoid making decisions on behalf of the user
  - It's fine to provide default values, but its generally better to raise an error when an invalid input is given.
  - It may be tempting to use default values for new inputs to avoid breaking API's, but remember that this adds complexity for the user. These inputs now have the possibility of being null or the value itself. If this input enables / disables a feature, this should be made abundantly clear to the user.