Organizing Small Software Projects

Ben Morgan
What is a “Small Software Project”? 

- In the context of this meeting, software projects that may be
  - **Specific**: e.g. to simulate/process/analyse data from an experiment
  - **General**: e.g. event generators used by many experiments

- The metric here for “Small” is **total developer FTE**
  - Not always related to size of the experiment/community using the software
  - FTE because developers may **only be funded a fraction of their time** to work on the software/experiment
  - This is slightly biased towards specific projects, as general ones often have an active, though transient, population of contributors

- To take the SuperNEMO $0\nu\beta\beta$ search experiment as an example:
  - ~100 researchers, 1-2FTE on software from 2-8 developers
Typical challenges faced in smaller projects

- **Limited FTE**, often with high turnover from limited funding/short contracts
  - “Get it working” pressure can lead to neglect of long term support items
  - Easily leads to a critical Bus Factor of 1-2
- **Not Invented Here** syndrome (reinventing the wheel)
  - Usually driven and exacerbated by the above
  - Fewer developers can mean less breadth of knowledge/experience
- Often a high level of **overthinking/engineering**
  - Not unique to small projects, but creates further long term support issues
  - Developers ≠ Users
- Importance of software work to an experiment or the community may not be recognized by funding bodies (or even experiment members!)
  - Can be particularly challenging for community wide projects
Addressing Challenges

● In all cases, the challenges are best addressed by projects implementing, maintaining and focussing on the foundations
  ○ Documentation (README/CONTRIBUTING, Sphinx, Doxygen)
  ○ Build and testing (CMake/CTest/etc)
  ○ Code style/quality tools (clang-format/tidy, Blackflake8)
  ○ Package management/deployment (Conda/Pip/Spack/CVMFS etc)
  ○ Version Control and Continuous Integration workflows

● Might be obvious (even boring), but neglecting these will build up high interest technical debt that sooner or later will have to be repaid in FTE
  ○ Been there, done/doing that

● There is a growing recognition of the criticality of sustainable and reproducible software for research, but never be afraid to repeat the case!
Tools and Suggestions for Small Projects

- Managing Code and Documents
- Development Workflows
- Release and Deployment
- Caveats:
  - Strong C++ bias
  - I’m just as guilty of not always doing what I suggest...
  - There will no doubt be a lot of “but…” and “what about…” questions (please write them in the Live Notes, they are always welcome!)
Organizing the Project

- **Always** have needed “first contact” docs like README, CONTRIBUTING
- **Never** a need for a “special” layout of source or other code
  - Languages often have a recommended/standard layout
  - Keep it simple, logical otherwise
- Might seem trivial, but you want principle of least surprise!
  - Minimize mental start up and context switching costs
Building the Project

- Always use a standard build tool
  - CMake, Autotools, Setuptools
- Keep scripts simple and standard
  - Avoid bikeshedding and over-engineering!
- Build scripts are code so document and maintain them as such
- A simple, well maintained build system pays dividends in later items!
Testing the Project

- Always use a common/standard testing framework
  - E.g. Catch2/googletest
  - stdout+Mk1 eyeball isn’t!
- Test-driven development can be very useful, both to
  - Ensure tests are written(!)
  - Help clarify interfaces/contracts
- Write tests to triage+fix bugs
- Ensure tests are easily built/run as part of the development workflow
  - E.g. make test
Documenting the Project

- Includes README, CONTRIBUTING, INSTALL files etc.
- Use standard tools like Doxygen, Sphinx for API/User Guides
- Like tests, document-as-you-go
  - Will be clearer at time of writing
  - Helps you to clarify your design
- “Compile” docs as part of build
  - Check for mistakes!
  - Helps in release stage
- Encourage users to contribute - they bring a different perspective
Formatting the Code

- Enforce a coding style to ensure consistency and familiarity
  - E.g. `clang-format` for C/C++, `Flake8` for Python
- Use of a tool reduces the chances of a holy war over spaces/braces
  - An area developers can be pointlessly opinionated about
- Use Git hooks and IDE integrations to format on save/commit
Analysing the Code

- Use static analysers to suggest, or apply, fixes for style, performance
  - E.g. `clang-tidy` (C/C++)
- Usually easy to integrate with the build or test systems
  - CMake makes `clang-tidy` use really easy, for example
- Particularly useful for modernizing or simplifying older projects!
Using Other Software in the Project

- All projects use external packages - dependencies
  - Already seen these with build/test/documentation ones!
- When the project needs an “X”, find a suitable off-the-shelf “X” first
  - “Suitable” means “Meets the requirements”
  - “I don’t like the implementation/style” isn’t a requirement
  - One on the biggest time drains for a small project can be the implementation, maintenance, and development of these new wheels
  - You can (and should!) contribute back to the projects you use!
- Make sure the project is not locked to a specific version of the external
  - All languages have constructs to compile/branch on the version
- A minimum, or range, of supported versions is fine though
  - Most build systems support this, e.g CMake’s find_package
Package Managers

- Always install dependencies with a widely used **package manager**
  - *Don’t write one yourself, please...*
- Choice will depend on languages used and platforms targeted
  - **Language specific**, e.g. pip
  - **Platform specific**, e.g. apt
  - **General(ish)**, e.g. conda, spack
- Use a “requirements” file/package
  - Lists packages and versions for a given “environment”
  - Aids easy setup/reproducibility
Docker/Singularity

- Containers can aid in distributing a development/use environment
  - *Modern IDEs can use them seamlessly (e.g. VSCode)*

- Package manager “environment” files and VCS/Container tags provide great system for reproducible setups

- **Don’t** use containers as a sticking plaster over a bad build/packaging system - **fix these first!**
Hosting the Project

- **Git+GitHub** is the primary VCS and hosting solution, though many GitLab instances at Institutes/Labs
  - *These may be better if the project has specialist needs like access to GPU hardware for CI*
- Git repos are trivial to move, so you can always change later!
- **Never manage your own hosting** - you *cannot* do it as well as GitHub/Lab or Institute/Labs
Use Issues and Boards

- Use **Issues** as primary channel of communication to reduce “where do I go to...” and help engagement
  - Email/Slack fine, **but encourage use of Issues**!
  - Beta “**Discussions**” feature may be good for “How do I” questions
- Use **Projects** to organise core tasks
  - Can add Issues to specific Projects e.g. “Next Release”
  - **Helps focus effort and reduce context switching cost**
Developing the Code: Topic Branches and Pull Requests

Never commit/push directly to the main branch - you will break it at some point!
Organising Topic Branches

- **Forking Workflow** simplest and best
  - Topic Branches on Forks
  - Project repo only for merging
- Tempting to push Topic Branches to project repo itself
  - Fine for 1-3 developers …
  - … but does not scale well
  - Forks don’t stop co-development
- Even lead developers should submit work from Forks!
  - Help separate development work from release/maintenance tasks

18

$ git checkout -b topic
... edit/build/test/commit...
$ git push my-fork topic

$ git fetch|rebase|push shared
Testing Pull Requests

- Always build and test each PR using a CI system
  - *E.g. GitHub Actions, Azure*
- A critical time saver, even if only targeting a single platform/OS
  - *Will quickly reveal any coding oversights or “works on my machine” gremlins*
- This is where earlier work on build and test systems starts to return the time investment!
Real value is the record of build/test results, so PR author can find/fix issues quickly.
CI Scripting

- Like previous recommendations, don’t overthink/engineer these
  - Each system does have slight differences in functionality
- Even complex builds don’t require complex scripting
- They are part of the project code, so develop and maintain them with the same care
  - Including submitting changes through PRs - the CI can test CI!
Reviewing Pull Requests

- Use **PR reviews** as “peer review for code”
  - Discuss the implementation, suggest revisions, improvements
  - Ask for addition of missing items like tests, documentation!
- Can be light touch “fine for me”, to major revisions, even rejection
- Collaborative approach is valuable in increasing the Bus Factor and reducing over-engineering
### Merging Pull Requests

Use Actions and Reviews to build a “pre merge” checklist.
Release & Deployment

● By testing/reviewing PRs before merging to the main branch, each PR merge provides a potential new release for the project
  ○ When/how often to make a release, and whether detailed testing (e.g. physics validation) is required before tagging is highly project dependent

● For each merge to main and new tag (release) of the project, a CI task should be triggered to:
  ○ Create and publish the documentation for the project
  ○ Create package(s)/environment for the project, e.g. conda, Docker, CVMFS

● Last item invaluable if you need to run large validation jobs for releases!
● Might require extra resources (e.g. ReadTheDocs, DockerHub), but earlier investment in standard tools simplifies integration!
  ○ And GitHub/Lab can often handle this for you (gh-pages, Packages)
An easy way to do this is through GitHub Pages.
Example Packaging: Spack

Especially valuable for community wide projects to have a presence in mainline package managers.
Finally: Don’t neglect Training until the last slide!

- “Documentation” should really be “Documentation and Training”!
  - Projects should point to existing Software Carpentry (and HSF!) lessons and others (e.g. GitHub Help) for the basics rather than write their own
  - Software Carpentry template good starting point for developing material for your project
  - Hands-on tutorials at, e.g. experiment collaboration meetings, can be valuable, though you may have to fight for a slot!
  - Link up remote/in-person material as later opportunities can be limited
- Developers should also actively seek training/knowledge opportunities
  - Engage with community meetings/courses like today
  - Institute/National Research Software Engineer groups/organizations
  - Conferences like CHEP, SC, though getting funding can be difficult
In Summary

- Primary challenge for small software projects is developer FTE, so this must be used **effectively**
- **Invest time in simple and standard tools** and methods to reduce developer overhead and points of failure, and increase automation
- **Use GitHub/Lab’s tools** to provide a simple yet highly capable platform for development that returns investment in standard tools
- **In short:** *Invest time in your tools, use them as designed, and let the computer take the strain!*