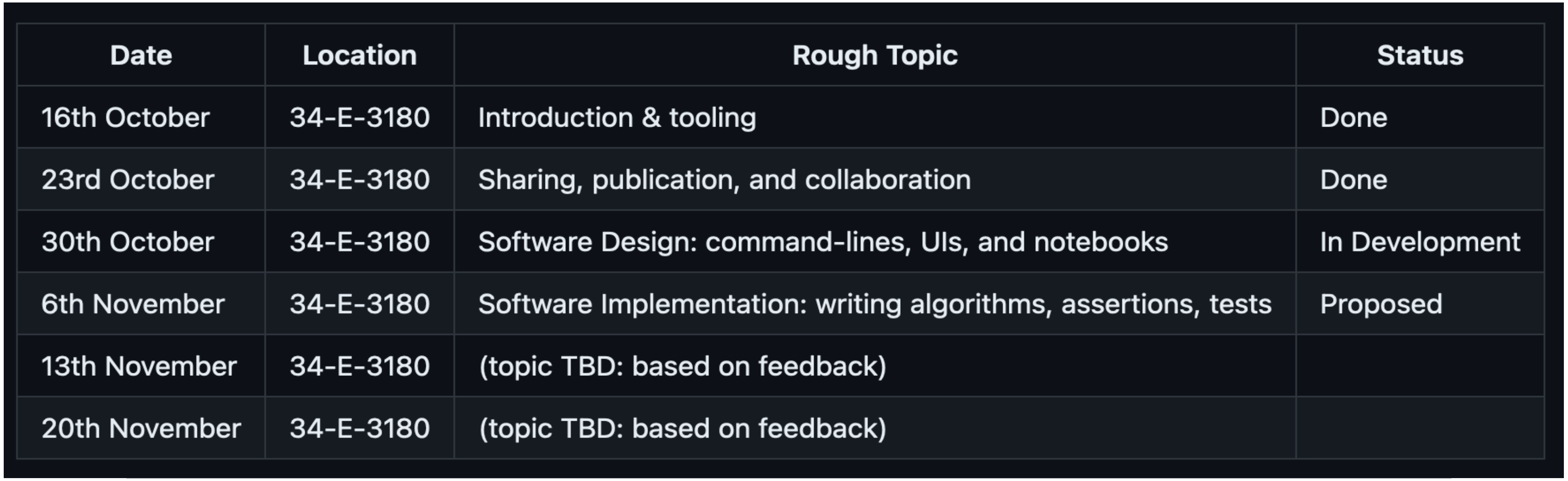
Research Software Engineering Meetups: Software Design

2023/11/30

Overview

- Schedule/upcoming
- A quick note about the topic order
- Software Design/architecture overview
- Interactive part

Schedule (provisional)



Date	Location	Rough Topic	Status
16th October	34-E-3180	Introduction & tooling	Done
23rd October	34-E-3180	Sharing, publication, and collaboration	Done
30th October	34-E-3180	Software Design: command-lines, UIs, and notebooks	In Development
6th November	34-E-3180	Software Implementation: writing algorithms, assertions, tests	Proposed
13th November	34-E-3180	(topic TBD: based on feedback)	
20th November	34-E-3180	(topic TBD: based on feedback)	

***source**: https://github.com/adamkewley/rse-meetups/*

A quick note about the topic order

- You might be wondering why we aren’t studying python coding...
- Or algorithms...
- Or high-level automation/scripting...
- Or low-level hardware coding...
- Or GP-GPU...
- Or Julia…
- Or...

It’s because, when working on software, those things aren’t the most important.
People, Processes, and Design are.

Software Design
(it’s all-encompassing)

- What are you trying to **build**?
- What’s likely to **change in the future**?
- What **processes** will make it easier to handle?
- What’s the desired **user experience**?
- What’s the desired **developer experience**?
- What’s the best way for you to get the most out of your **team/collaborators** (or vice versa)?

Research Software Design

(typical steps)

1) **Figure out what you actually need**
   a) Is it to ship your thesis/paper, where the software is purely a related side-effect?
   b) Is it to write a small, but perhaps heavily reused, script/app that helps work on a standalone research problem?
   c) Is it to write a larger application with multiple, reused, algorithms? Perhaps with multiple developers over a longer period of time?
   d) Is it to write a fundamental system that *many* other researchers will use as a key component of their research, multiple developers, and a very long lifetime?
2) **Try to use the same/similar processes as other researchers in your field***
   a) E.g. programming language, libraries, applications
   b) Remember that people and processes are usually more important than just the code (for better or worse)
3) **Deliver what you need, prioritizing**
   a) Iteration speed
   b) Simplicity/readability
   c) Correctness
   d) Performance

*: Unless you plan on redefining your research area's processes, e.g. by writing a whole new platform.**
**: Even then, you'll have difficulty writing a whole new platform unless you learn the old one first.

Software Architecture

(categories that small-/medium-sized software projects typically fall into)

- **Libraries**
  - Packages
  - Dlls
  - That mathworks script you downloaded to fit a sphere to some datapoints. It works, but you have no idea why, and the comments are just “fit X”, “do Eigen analysis”, etc.
- **Applications**
  - Servers (uncommon in RSE, unless you’re delivering a data processing service)
  - UIs (less-common in RSE)
  - Notebooks (becoming increasingly common – try these out)
  - Command lines (extremely common in SE and RSE)
- **Hybrid (i.e. both of the above)**
  - E.g. write your code as-if it were designed as a modular library, but then separately put command-line interfaces around it so that you can (e.g.) run it from VSCode or run it from a batch script, etc.

Software Architecture

(typical choices for python-based research projects)

- **Libraries – Python packages and modules**
  - A directory containing an __init__.py file is a package
  - Individual python files (optionally, within package directories) are modules
  - Modules define functions and classes that can be called from python
- **Applications – Command-line scripts (what you'd think of as a python script)**
  - Typically starts out as a top-level script that you're running a lot (e.g. `main.py`, `run.py`)
  - You can also add a `__main__.py` file into a package directory to tell python what to run when you "run a package" (e.g. `python -m pip` would run `__main__.py` in `pip/`
  - Python comes with batteries-included for command-line parsing (argparse), configuration files (config), and environment variables (`os.env`)
  - (not covered here, but notebooks usually means "Jupyter Notebook in an Anaconda envrionment")
- **Hybrid - Usually happens naturally as your codebase grows**
  - Have modules/packages for the command-line bit (e.g. __main__.py) and modules/packages for the algorithmic big (e.g. fft.py, fea.py)
  - E.g.
    - Start out with a python script that you can run top-to-bottom (i.e. it's a batch-oriented command-line application)
    - Then break out parts of that script into modules/packages
    - Then "jazz it up" with a nicer command-line interface, arguments, configuration file support, etc.

Discussion: The design of some projects

Python codebases from PLOS Computational Biology:

- https://github.com/John-king-zhou/COVID
- https://github.com/crossley/sensory_uncertainty_fffb
- https://github.com/ZhangGroup-MITChemistry/OpenABC
- https://github.com/ncbi/TranNet
- https://github.com/ZhangGroup-MITChemistry/OpenABC
- https://github.com/serena-aneli/recombulator-x

Jupyter codebases (PLOS Comp. Bio.):

- https://github.com/RuodanL/fixation_probability
- https://gitlab.gwdg.de/nsharma/self_loops_egt

Discussion: Typical Design Observations

Seems basic	I have to read/change code to run the analysis I need	Running it seems quite manual (e.g. clicking through a lab notebook)	It’s not obvious how each part of the code relates to the science
Easy to understand	Seems like there’s a lot going on	Clear feedback that is scientifically relevant	I have to run tertiary analysis on the output to *do* anything interesting
Seems like it would be difficult to develop further	The scientific explanation relates to the code very clearly	It’s not obvious how I’d integrate this code into my codebase	Running it is easy

Interactive: Redesign a Codebase

(overview)

(story) we have obtained a “typical” (small) research script that’s been gradually growing. The following problems have already been identified:

- **It’s hard-coded.** you run the script and it does what it does. It is not possible to (e.g.) provide different data to the script, or modify its behavior, without changing the code
- **The only way to use it is to *run* the whole thing**. Other people in the team can’t just import one part of the codebase. Merely importing the script tries to run the script, rather than just importing the functions for use elsewhere.
- **It’s lacking organization**. All of the code is is one file. There isn’t a “tidy” separation of “here’s the algorithmic bits” and “here’s the utility bits” – everything is in one place and it’s up to the (external) reader to dig through all of the code to find the part they’re interested in.

Interactive: Redesign a Codebase

Rule #1: always ensure that you can run your application/tests after each step.

- Get base research script from **TODO**
- Create *project* directory that contains a *package* directory that contains the script
  - advanced: create the project directory as a GitHub repository, as in previous sessions, so that you can commit each step
- Setup your development environment (e.g. VSCode) to open your project folder. “Running” the project should run the script. Ensure that actually works.
- Study the script and identify what parts of it would be candidates for being broken out into separate modules/packages (discuss this)
- Start breaking it out into modules/packages
- Use `argparse` (google it, ask ChatGPT, etc.) to add arguments to the “main” module. Good candiates for arguments: `num_samples`, `sampling_algorithm_name` etc.
- Use `config` to add configuration file support. Allow users to specify (e.g.) `sampling_algorithm` via a configuration file. Command-line arguments should take priority over anything that can be defined via the config file.

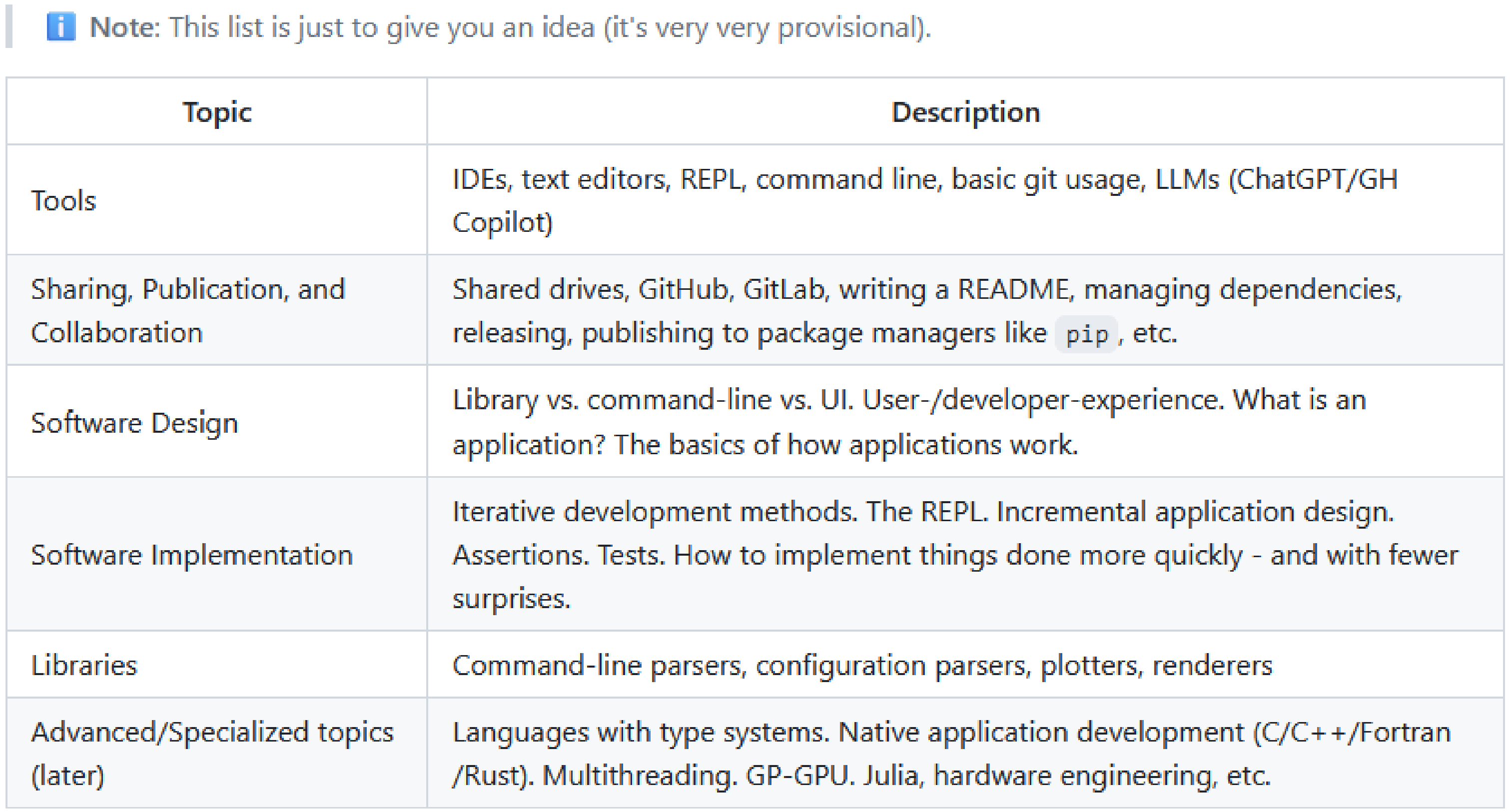
Resources (google)

“Structure of a typical python project”
How to break up code into packages/modules
Python argparse
Python config

Session Checklist

- Design is about understanding a combination of the problem, the people, and the process
- Typically, the easiest thing to do at the start is to study+emulate previous work in the field
- Software architectures can be simplified into a few categories
- There are common patterns/steps for cleaning up or redesigning a codebase to make it ready for the next phase of building

Next Time (up for debate)



> ℹ️ **Note**: This list is just to give you an idea (it's very very provisional).

Topic	Description
Tools	IDEs, text editors, REPL, command line, basic git usage, LLMs (ChatGPT/GH Copilot)
Sharing, Publication, and Collaboration	Shared drives, GitHub, GitLab, writing a README, managing dependencies, releasing, publishing to package managers like `pip`, etc.
Software Design	Library vs. command-line vs. UI. User-/developer-experience. What is an application? The basics of how applications work.
Software Implementation	Iterative development methods. The REPL. Incremental application design. Assertions. Tests. How to implement things done more quickly - and with fewer surprises.
Libraries	Command-line parsers, configuration parsers, plotters, renderers
Advanced/Specialized topics (later)	Languages with type systems. Native application development (C/C++/Fortran /Rust). Multithreading. GP-GPU. Julia, hardware engineering, etc.