Scientific Software Development

Inga Ulusoy, Scientific Software Center, Interdisciplinary Center for Scientific Computing, Heidelberg University

March 2023

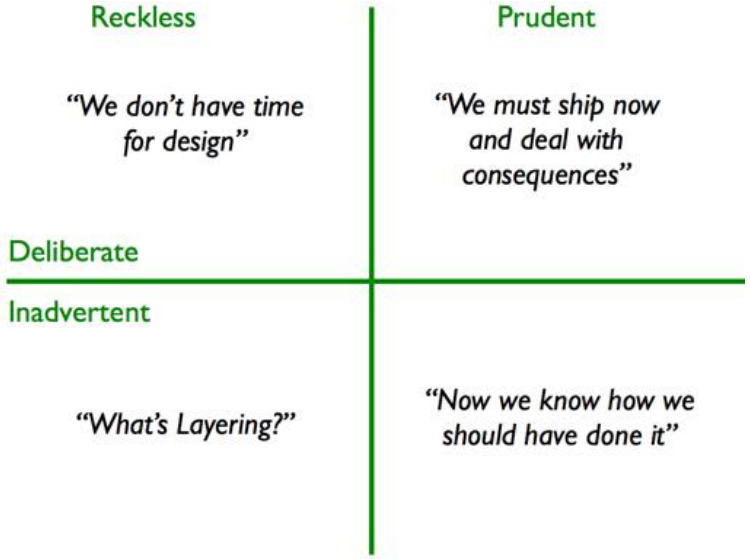
Unit 2: Clean coding as a team

- Technical debt and clean coding
- Style guides
- Working with an IDE
- Linting and performance of code
- More on git: “Clean” repositories
- Pull requests, code review and merging

You will start to work collaboratively with your team.

Technical debt

- The result of prioritizing speed over clean code.

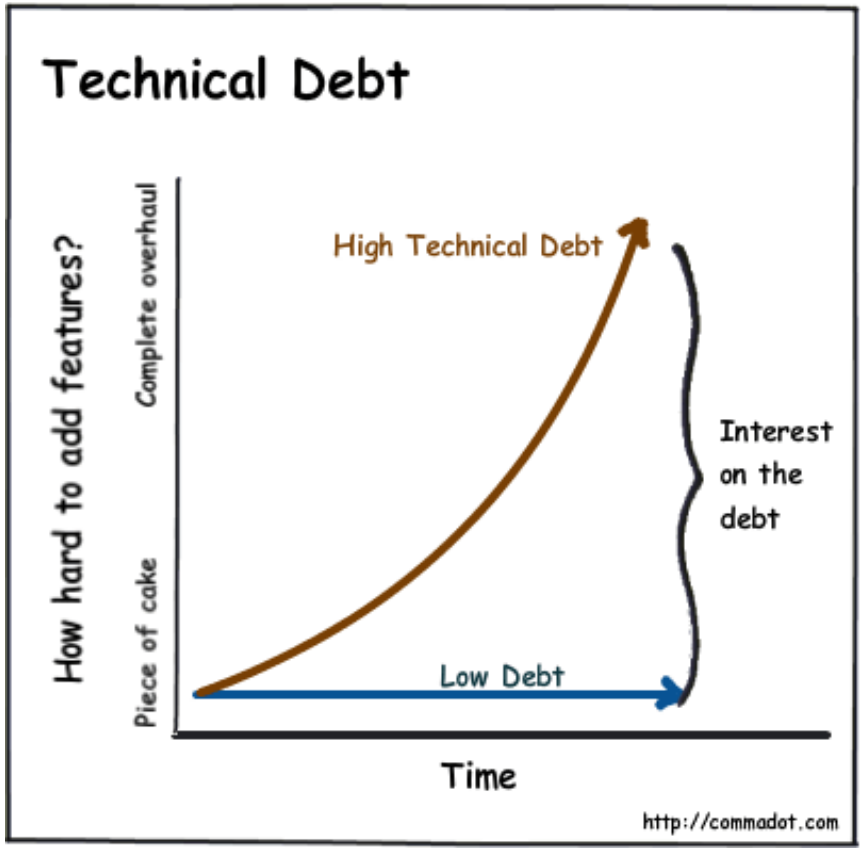


https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

- Refactoring: The process of restructuring existing code and thereby removing technical debt.

Technical debt hinders development

- A high technical debt makes it much more cumbersome to implement new features.



You will save time in the long run by keeping your code clean.

https://commadot.com/the-ux-of-technical-debt/

Clean code: Coding best practices

write comments and documentation

Advice: actively use comments in the documentation through tools like i.e. `sphinx` or `doxygen` – this way your comments will not become outdated!

write efficient but readable code

Making use of functions and classes (methods) can be very efficient, but sometimes hard to read.

proper styling

The indentation and styling should be consistent throughout the project. Avoid deep nesting and too long lines.

proper naming conventions

The names of variables and procedures should be meaningful. *(ideally in English)*

no imaginary future cases

Don't write code for imaginary future scenarios. Only write what you need.

accuracy before speed

Make your piece of code work first, then improve efficiency.

no hardcoding

Those should be constants and declared in the overhead.

reuse functionality

Implement features in a way that enables reuse of functionality in different contexts. Do not repeat yourself.

develop iteratively

Give breaks, don't write all code in one go. Iterate over (pieces of) the code several times.

use helper methods

A method should only do what it is supposed to. Anything else should be contained in other functions/methods.

use existing libraries

You will not be able to code more efficient than the pros.

refactor code

Whenever you have time, refactor parts of your code.

Unit 2: Clean coding as a team

- *Technical debt and clean coding*
- Style guides
- Working with an IDE
- Linting and performance of code
- More on git: “Clean” repositories
- Pull requests, code review and merging

Style guides

- Python is highly readable if code is pythonic (adhering to Code Style guidelines)
- PEP-8: Python Enhancement Proposal – a set of rules for writing pythonic code https://www.python.org/dev/peps/pep-0008/
- Stylized version of PEP-8 https://pep8.org/
- Google style guide for Python https://google.github.io/styleguide/pyguide.html

```
>>> 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!
```

Python styling do's and don’t's

- Go to https://pep8.org/ and read through these sections:
  - https://pep8.org/#code-lay-out
  - https://pep8.org/#naming-conventions
  - You may read more or also peruse the google recommendations at https://google.github.io/styleguide/pygui
- The recommendations for naming, especially naming of modules, is very important if you want to publish your package in the Python Package Index PyPI

Unit 2: Clean coding as a team

- *Technical debt and clean coding*
- *Style guides*
- Working with an IDE
- Linting and performance of code
- More on git: “Clean” repositories
- Pull requests, code review and merging

Working with an IDE = integrated development environment

- I will be using VSCode (Visual Studio Code, https://code.visualstudio.com/) in this course, I recommend that you install and use this one. You may also use others such as Atom etc if you prefer.
- Features: Syntax highlighting, intelligent code completion, embedded git, debugging, …
- Possible to install extensions
- Most popular IDE on stackoverflow

Working with an IDE

- Install VSCode or your chosen IDE and familiarize yourself with its functionalities. Follow the steps as outlined here https://code.visualstudio.com/docs/python/python-tutorial

Unit 2: Clean coding as a team

- *Technical debt and clean coding*
- *Style guides*
- *Working with an IDE*
- Linting and performance of code
- More on git: “Clean” repositories
- Pull requests, code review and merging

Linting

https://pypi.org/project/flake8/
https://simpleisbetterthancomplex.com/packages/2016/08/05/flake8.html

A Linter automatically highlights styling and syntax errors, as well as suspicious constructs in your code.

Most IDEs run linters in the background as you work on your code, and directly highlight problems. Resolve all issues before pushing to your repository.

Additionally, you can check your code in the terminal by running the linter manually (ie. `flake8`). You will also want to run the linter in your GitHub actions (we will get to this in unit 6).

Linting

- I will be demonstrating the usage of a linter in the live session.

Performance of code

https://docs.python.org/3/library/timeit.html
https://docs.python.org/3/library/profile.html

You can check performance by using `timeit` or `cProfile`.

`timeit` will measure the execution time of specified bits of code.

`cProfile` enables deterministic profiling of your code, including how many times functions were called and how long these took to execute, but will add some overhead to the overall run time of your program.

Performance of code

- I will be demonstrating how to assess the performance of Python code in the live session.

Install additional packages

- Install these additional packages for the next session:
  - `pip install flake8`
  - `pip install pytest`

Unit 2: Clean coding as a team

- *Technical debt and clean coding*
- *Style guides*
- *Working with an IDE*
- *Linting and performance of code*
- More on git: “Clean” repositories
- Pull requests, code review and merging

Git: "Clean" repositories

- Only track the files that are essential and cannot be recreated.
- Others should be ignored via `.gitignore` to keep your repo clean.

Source files: *.py
Documentation files:
*.md
Configuration files: *.yml
...

Temporary files:
.ipynb_checkpoints/
__pycache__/
.pytest_cache/
Output files

.gitignore

- Look at the `.gitignore` file in your repo.
- Look at this page https://git-scm.com/docs/gitignore for the `.gitignore` syntax. Are there additional files in your folder that you should ignore via .gitignore?

Unit 2: Clean coding as a team

- *Technical debt and clean coding*
- *Style guides*
- *Working with an IDE*
- *Linting and performance of code*
- *More on git: “Clean” repositories*
- Pull requests, code review and merging

Pull requests (PR)

- Used to tell others that your added changes are ready for review
- You can discuss the changes in the open pull request
- You may make changes to your code and commit based on the feedback
- GitHub can run actions such as automated tests, style checks, coverage etc upon PR

Pull requests

- Once you finish a line of implementation, open a PR
- Tag your teammates as reviewers
- In the review, you can comment on lines of code
- The PR allows to close issues
- After everything is satisfactory: Squash and merge -> keeps your commit history clean
- Delete your branch on the remote and also locally

  `fetch -p` and `branch -d yourbranchname`

Unit 2: Clean coding as a team

- *Technical debt and clean coding*
- *Style guides*
- *Working with an IDE*
- *Linting and performance of code*
- *More on git: “Clean” repositories*
- *Pull requests, code review and merging*

Live lesson

- In the beginning of the live lesson, you will discuss with your ***team*** and decide on naming conventions and general coding style.
- Merge your efforts into one Jupyter notebook in one repo. Each person in your team will continue to work on their part of the problem set, and you will use PRs and code review to increase the quality of your implementation.
- Use a linter to check your notebook for coding style.
- Adhere to the ***coding best practices***!

Live lesson - Demonstrations

- The following demonstrations will take place in the beginning of the live session:
  - IDE with Jupyter notebook;
  - how to use a linter; linter for Jupyter notebooks;
  - linter as pre-commit hook;
  - performance of code;
  - PR, code review, and merge conflicts.