EECS 201: FOSS

Authored by Zalan Shah

Some Context

Free and Open Source Software (FOSS) refers to any program or library that is freely licensed. In simple terms, that means that anybody is free to copy, change, or use the program without paying (and in many cases, crediting) the creator.

FOSS exists at the core of software engineering: most of the things you interact with are FOSS. In fact, Linux, VS Code, and even Microsoft Calculator fall under the category of FOSS. In fact, every program we've utilized in this class (from Python to make to head to sed) has been FOSS software.

Beyond just actual programs, there's also libraries. A library is just a bit of code that somebody else has written for you (think of C++ std::vector or Python's NumPy). These libraries also fall under the umbrella term of FOSS, too: while they're not full programs in their own right, these programs can be used by anybody in any way one might please.

This second type of FOSS (libraries) often goes unrecognized, but they are often the unspoken heroes of the world. To give you an idea of *just* how important the contributions of FOSS developers and maintainers are, here's a funny story of a time where one package almost destroyed the internet.

All this is to say that FOSS developers are the backbone of modern technology. It also leads into a very important moral lesson: as an engineer (a software engineer, if we're being specific), the things you create and build have real world impacts and influence real world people. As such, you as an engineer should have a moral code to do what's right by the world. There's many moral codes for engineers that exist already, but ultimately it's on you to bring good to the world with the skills you have in programming. One of the easiest ways to do this is to contribute to the wealth of knowledge and create programs that others can use for free.

Part 0: So what am I actually doing? (0 points)

Good question. In this project, you will be:

- coming up with a novel idea for a program that others can use **(Part 0)**
- building a development roadmap **(Part 1)**
- Creating V1.0.0 of your program **(Part 2)**
- Writing all the documentation associated with your project **(Part 3)**
- Publishing it to a package manager **(Part 4)**

Let's start easy. We're going to start off by brainstorming. Here's some examples of FOSS ideas that would be acceptable for this project (had they not existed already):

- Runes
  - Many emojis aren't just a single emoji, but instead are a combination of emojis that come together to form the relevant emoji (this is how emojis can have skin color, for example). In Javascript, a string can be split using the .split() function, but it doesn't work correctly for strings that contain emojis. This package allows you to do so correctly.
- Fakerjs
  - This emoji generates random junk data (such as names, addresses, etc) for people that want large sets of user data for testing.
- Jackson
  - A Java library that can parse through JSON. Not much to say here >.>
- sl
  - A CLI program that will run a steam locomotive (i.e. an sl) across your screen when you mistype ls. There's no way to kill it; SIGINT doesn't stop it; you just have to let it run across your screen for 10 seconds as punishment. This one's sort of a joke, but it's helped me get better at navigating the terminal, so there's that.
- Figlet
  - A program that can generate large ASCII-art style text.

Obviously, there's a lot of variety in both use-cases and languages. You can pretty much come up with anything, so long as it's interesting to you and useful in some shape or form. Spend some time brainstorming ideas. Again, this can be anything, from a fully-fledged program that does something to a library that others can use in their code. There are two conditions that your idea must fulfill:

- It must be complicated enough to require multiple updates. This shouldn't be the type of program you can write a 50-line program for and call it done (this is entirely subjective; see the checkpoint for more information).
- It must solve a problem. This doesn't need to be a crazy big problem (i.e. while you could, your project doesn't need to be you creating and publishing a whole new Linux flavor), and could be "a program that can pull data from a website and do something with that data" or "a program that allows you to play Sudoku in the terminal."

CHECKPOINT:

Once you have come up with an idea you think is suitable, do research online. Ensure that this is not a program that already exists. Then, talk to the overseer of this project. Pitch it to them: explain your vision, what problem you're trying to achieve, and how you see yourself going about doing it! You can be as creative as you want. You can just talk to us, but we'd love to see you go the extra mile of creating a slideshow and giving a real "investor pitch."

You'll notice as you work through this project that things will feel very hands-off. That is intentional – this isn't meant to be a project you submit to the Autograder and don't think about again. Our intention for this project is to both give you some real-world experience you can put on your resume and let you explore the programming skills you've developed in college by doing something that you find interesting.

Part 1: Roadmap (5 points)

IMPORTANT: **if you did NOT complete the checkpoint in Part 0, you can NOT receive credit for completing this part. No exceptions will be provided under any circumstances.**

Alright, no more games. This is the big leagues now. There is no spec for exactly what your program should do. It's on you to figure out exactly how your program works and how it should do that. More importantly, you should have an idea of where you're planning on going with this project (keep in mind, our intention is not that this will be a project you're making for the sake of this class, but one that you're interested in enough that you'll continue working on it after the class ends).

First, let's talk about versions. If you've ever looked at the update information for apps, games, and other software, you might've noticed three numbers (if not, try running the command `Python3 --version` in the terminal). Generally, versions of programs are in the form of V{MAJOR}.{MINOR}.PATCH} (i.e. V1.2.3).

- MAJOR
  - The first number typically represents the iteration of some software. If there is a massive overhaul that significantly changes how software interacts with items related to a previous version and renders previous data unusable, this is typically indicative of a massive version change.
  - Take Python: not all Python3 code will work with Python2 or Python1 code, hence why each one of them is at a different version. The number next to them refers to this same first number here.
- MINOR
  - The second number indicates significant new features.
  - If you take Minecraft as an example, this number changes every time new blocks, mobs, etc. are added, since each one of these updates adds a ton of new features to the game.
- PATCH
  - The third number is indicative of bug updates, or other minor changes that don't have a significant impact on usage.

For more details on how this works, see here.

Obviously, you shouldn't aim to update the major or patch numbers frequently, nor can you reasonably expect when they should come up. You can, however, plan the middle ones.

Create a reasonable roadmap of versions. A roadmap is a schedule that developers follow to know when what code changes are expected by. Write up a roadmap of reasonable updates. This roadmap should include what each version should change. Optionally, you could add expected release dates to your roadmap. We are expecting you to create a roadmap of updates and features that you could work on for at least 6 months to a year.

For example, your roadmap could look like:

- V1.1.1:
  - Add first feature that allows basic use
- V1.2.1:
  - Add flag option -A
  - Add flag option -B
- V1.3.1
  - Add flag option -C
  - Add main menu
- Etc.

Note that we aren't expecting you to work on this project for a full year. We do, however, want you to have this roadmap and encourage you to keep working on this project even after the class has ended.

Your roadmap can be done in a graph, or even on a text document – the format is up to you.

CHECKPOINT:

Once you have finished creating your roadmap, meet with the overseer of this project for approval and the associated points with this part.

Part 2: Build the dang thing (20 points)

IMPORTANT: if you did NOT complete the checkpoint in Part 1, you can NOT receive credit for completing this part. No exceptions will be provided under any circumstances.

This part is, thankfully, a lot less reading. Build V1.1.1. Use git for version control and use a public GitHub repository to upload your work.

CHECKPOINT:

Once you have finished building V1.1.1, meet with the overseer of this project for approval and the associated points with this part.

Part 3: Now how do you use it? (5 points)

IMPORTANT: **if you did NOT complete the checkpoint in Part 2, you can NOT receive credit for completing this part. No exceptions will be provided under any circumstances.**

At this point, you're almost ready to start letting users, well, *use your software.* All we have to do now is tell people how to actually use it. You will need:

- README.md
  - This should outline exactly how to use your software, including, but not limited to:
    - How to install it
    - How to run it
    - Any arguments, if a terminal-based program
    - How to utilize it in programs, if a library
    - How to contact the creator (i.e. you)
    - Acknowledgements (I'd appreciate it ;)), and anything else you might want to add
- CHANGELOG.md
  - This will be a running list of every single version. Every version should contain a changelog of what was changed in that version. See here for an example.
- LICENSE
  - A file that dictates who can use your software, and how it can be used. There are many different licenses you can choose from, but we encourage you to be as non-restrictive as possible with the license you choose. Generally, the MIT License is generally regarded as one of the least restrictive.

The files that end in .md are markdown files, and will require you to use LaTeX.

CHECKPOINT:

Once you have finished writing these three files, upload them to GitHub. Then, meet with the overseer of this project for approval and the associated points with this part.

Part 4: Publishing time (10 points)

IMPORTANT: if you did NOT complete the checkpoint in Part 3, you can NOT receive credit for completing this part. No exceptions will be provided under any circumstances.

Now, people are almost ready to use your masterpiece. All you have to do is ensure that they can actually install it.

Here’s the thing: because you could have used anything to write your library/program, this will be entirely a *you* thing. We can give you this much guidance, though:

- Terminal-based program
  - You want to upload this to apt-get and brew
- Python package
  - You want to upload this to pip
- JavaScript package
  - You want to upload this to npm
- Rust package
  - You want to upload this to the crate registry
- All others
  - There’s probably some package manager out there that you can make use of. So long as it allows a person to install a package and use it, it’s fair game.

CHECKPOINT:

Once you have finished publishing V1.1.1, meet with the overseer for this project for approval and the associated points with this part.

Congratulations! You’ve just went through the stages of planning, building, and publishing FOSS for the world to use. Now that you’re done, don’t just stop here. You’ve got a roadmap; you can keep working on this project and building on it for people to keep using.