Programming Project Structure

This document contains a brief description of structuring a programming project as part of the Systems Development Life Cycle, which every software developer should follow without exceptions to reduce doubts and improve estimates, as well as for effective management, monitoring, and quality control throughout all stages of project development.

In this document, I will share each stage that should be considered before starting a new programming project and before diving into coding it.

As a software developer, I have been working on many projects since 2014 and have learned several valuable tips along the way. I want to share some of these tips that I have learned through my experience as a software developer, as they are relevant to the Systems Development Life Cycle. I strongly recommend following each stage described in this document step by step.

Refer to the table of contents to access details and further information about each stage on the internet. This document serves as a brief introduction to some of the stages you should consider in the Systems Development Life Cycle before embarking on any programming project.

https://about.me/sebaxtian

https://www.youtube.com/watch?v=CAeWjoP525M

Table of Content

1. Identify the Problem

The first thing you need to do before you start worrying about the structure of your code is to understand the problem that your project is going to solve. If you are in an interview, the best way to present your project is to begin by discussing a problem you encountered and how you solved it with code.

Even if the problem is not the most complicated one in the world, the ones that stand out the most are the simpler ones that solve real-world problems, which are relatable to other people.

Make sure you understand the problem and can explain it clearly.

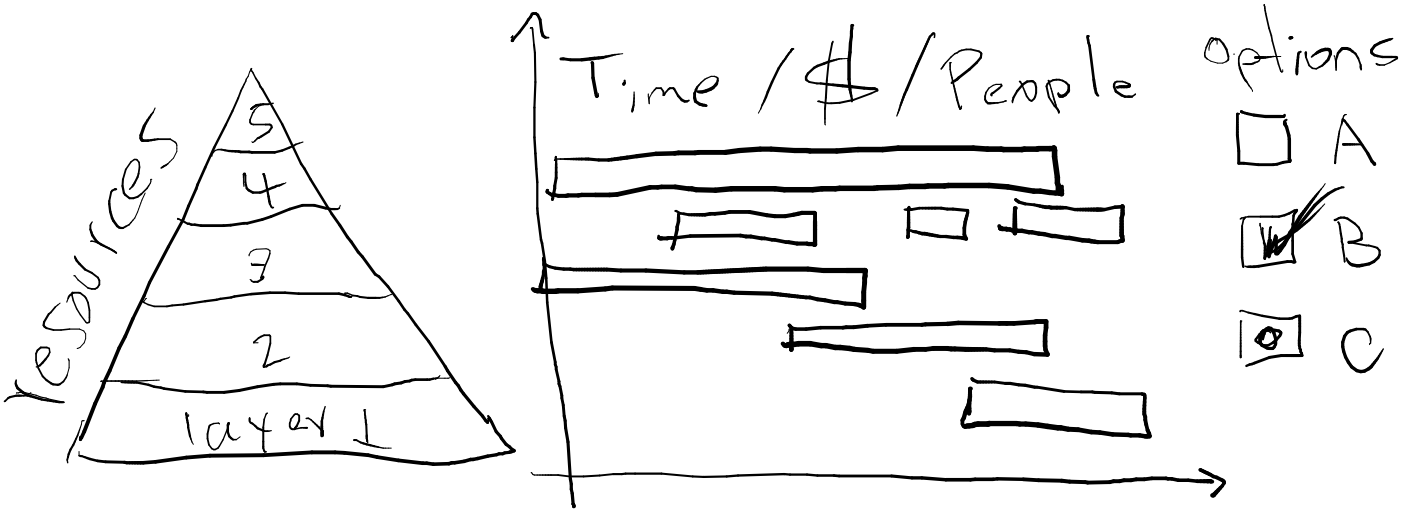
2. Have a Plan

Let's move on to the next step you need to take when working on a programming project. You don't need to go overboard by creating an excessive number of system diagrams and sequence diagrams or any fancy stuff you might come across online. However, it's essential to invest some time before diving into your project and develop a basic plan.

I recommend starting with the fundamentals. Decide on your tech stack, choose a database, outline your backend and frontend structures, and identify the key components your system will require. This is a good practice, and if you can discuss it during an interview or include these design documents in your GitHub repository, it will make a strong impression.

If you don't want to go into such detail, at the very least, jot down a few key considerations and start thinking about the major systems you'll need to build before diving into the project.

At this point, you've understood the problem, created a plan, and now it's time to work on your directory and folder structure.



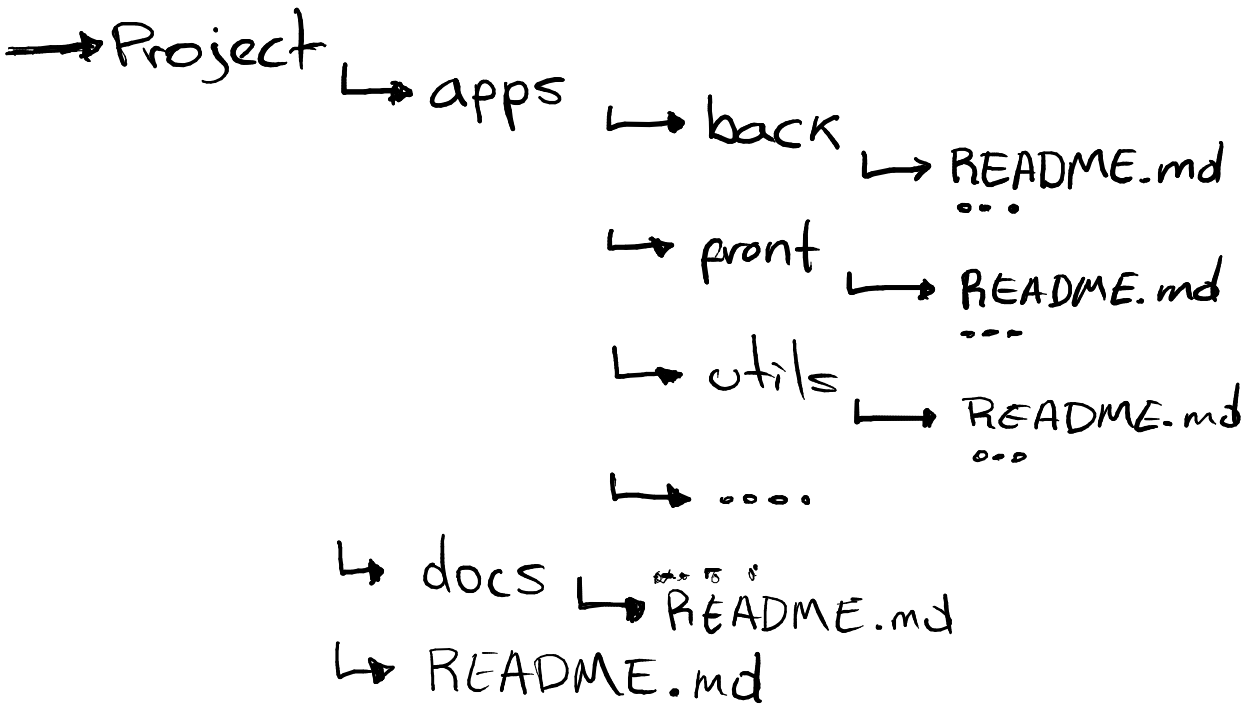
3. Structure your Directories

Now it's time to work on your directory and folder structure. Before you dive headfirst into writing a ton of code, it's essential to set up your repository, establish your codebase, and have a general idea of where you'll be placing different parts of the system within your code.

This is the stage that can get a bit tricky, and it often evolves over time. It also depends significantly on the type of framework you're using. For instance, frameworks like React, Django, or Flask come with their own directory structures and conventions that you should typically adhere to.

If you're building something entirely from scratch, you have more freedom to structure it as you see fit. However, as a general rule, it's a good idea to follow the conventions of the framework you're using.

If you're unsure about these conventions, you can examine examples of large applications that utilize the same framework to see how they've structured their projects. Additionally, you can turn to resources like ChatGPT for guidance, or simply trust your intuition.



3. Structure your Directories

The first thing to keep in mind is that, however you decide to structure your project, consistency is key. Even if you think you might be doing it incorrectly, stick to your chosen approach consistently every single time.

For instance, if you decide to always start your folder names with a capital letter, then make sure to do that consistently. If you opt for using ".test" to indicate test files, always use ".test". Whether you use underscores instead of spaces or dashes, maintain that consistency throughout your project.

This is a crucial lesson I've learned over the years. It's much easier to make adjustments when everything follows the same pattern, rather than dealing with a mix of inconsistent file names and conventions. So, regardless of whether your chosen method seems flawed, always stick with it and keep it consistent.

We have our directory structure here, and it really depends on the type of project you're working on. The structure will depend on how large and complicated the project is.

At the root of my repository, I keep general folders, and I try not to clutter it with too many files. The reason for this is that I want quick access to items like a README file, design documents, and other important information that may not necessarily be code. These are placed right at the root of my repository.

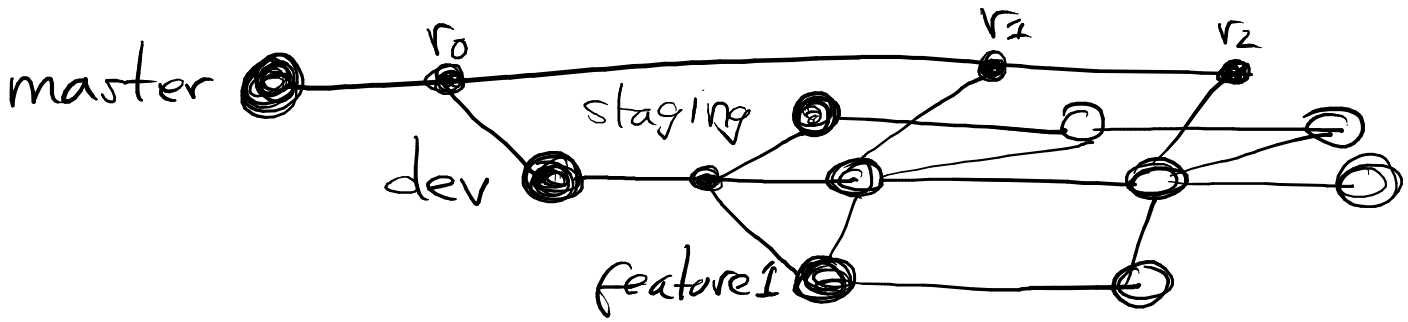
When we dig in inside something like 'apps,' I typically organize my backend, frontend, and any utilities into separate directories. This is an important point to note because many times you'll need to share code between different systems.

4. Use Version Control System

Now that we've established our directory structure, the next step is to start using a version control system immediately; don't wait until the very end. Begin using it right from the beginning. This practice will allow you to become proficient in creating commits, making pull requests, and pushing changes to remote repositories. It will also be evident to anyone reviewing your commit history that you know how to separate different tasks into individual commit messages, making your commit history meaningful.

Effective use of Git, a version control system, is particularly valuable as your projects grow larger. Oftentimes, you'll need to revert to previous versions of the codebase, and having well-crafted commit messages and logically organized code changes within those commits will make this process smoother.

To get started, you'll need to initialize a Git repository, add your various files, and write concise, meaningful commit messages. These small, well-documented commits will create a valuable history. When you look back at your project's history, you'll be able to see everything you did, how long it took, and easily return to any previous state of the codebase. This is especially valuable in large projects.



5. Modularize and Componentize your Code

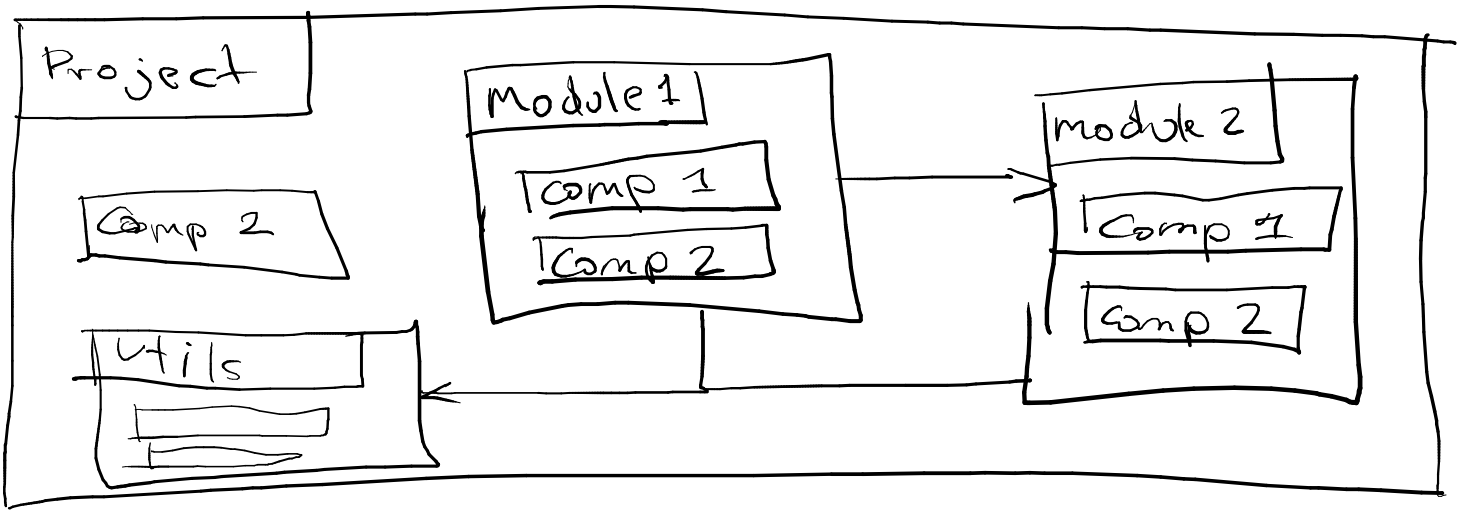
Now, at this point, you can start coding, but it's essential to ensure that you're separating everything and taking a modular approach. It's better to have more files or components, making your code more reusable, rather than having fewer.

Often, people create massive files with 400-500 thousand lines of code, which is not advisable. This makes it very challenging for someone to review or read the code, and it becomes difficult for you later on to understand what's happening.

In general, aim for short, reusable, and easy-to-understand components that belong to cohesive groups. This means that if you have a helper's file or a module, like a math module, you should only include code that logically fits together in that file. If something doesn't belong, move it to another file.

Maintain organization and keep code in logical places where it makes sense, ensuring it's easy to navigate. If you can't locate a function simply by reading the directory and file names, you might have placed it in the wrong location.

At this stage, if you've followed these principles, you're on the right track. You've written clean, beautiful code that scales fantastically.



6. Documentation

The next thing we need is some documentation. I'm not a fan of over-documenting, and I don't recommend writing tons and tons of documentation for your code, but having some is very helpful.

First, consider creating an installation guide along with a troubleshooting guide. Many times, you'll need to run your code on multiple platforms, in various locations, and across different parts of the world. As a result, you're likely to encounter various issues. I recommend documenting these issues, along with their solutions.

In your installation guide, provide instructions for someone who has never run this code before. Explain how they can set it up and what prerequisites they need. Do they require certain keys or special access? Should they contact an administrator? Cover all installation steps comprehensively. Additionally, create a separate document that lists the issues you've encountered or others have faced, along with their respective solutions.

For open-source projects, you should also consider having contributing guidelines. This informs potential contributors about what is expected and what a pull request (PR) should look like. Provide general tips and advice to help them navigate the process. You may also want to create a template for issues to streamline problem reporting and resolution.

- Installation / troubleshooting guide
- Readme.md
- requirements.txt
- example.env
- .gitignore
- issues template
- PR guidelines

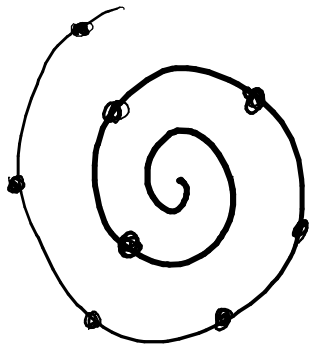
7. Testing

Testing may not always seem necessary for a hobby project. After all, you're not developing a critical application that could lead to financial losses or catastrophic failures like a rocket ship exploding. However, including tests in your project demonstrates a commitment to proper software development and methodology.

When I see someone has taken the effort to write decent tests, even if they only cover a small portion of the code, it impresses me. I know that writing those tests may not have been enjoyable, but they did it because they understood its importance. That speaks volumes and stands out.

Whether you choose to write a few unit tests, an integration test, or an end-to-end test, having some tests and demonstrating your commitment by organizing them in test folders, giving them appropriate names, and writing them correctly can make a significant impact.

- Unit test
- Integration test
- End-to-End test



8. Dependency Management

This aspect is highly dependent on the type of project you're working on, but it's essential to keep in mind the use of secure and well-maintained dependencies.

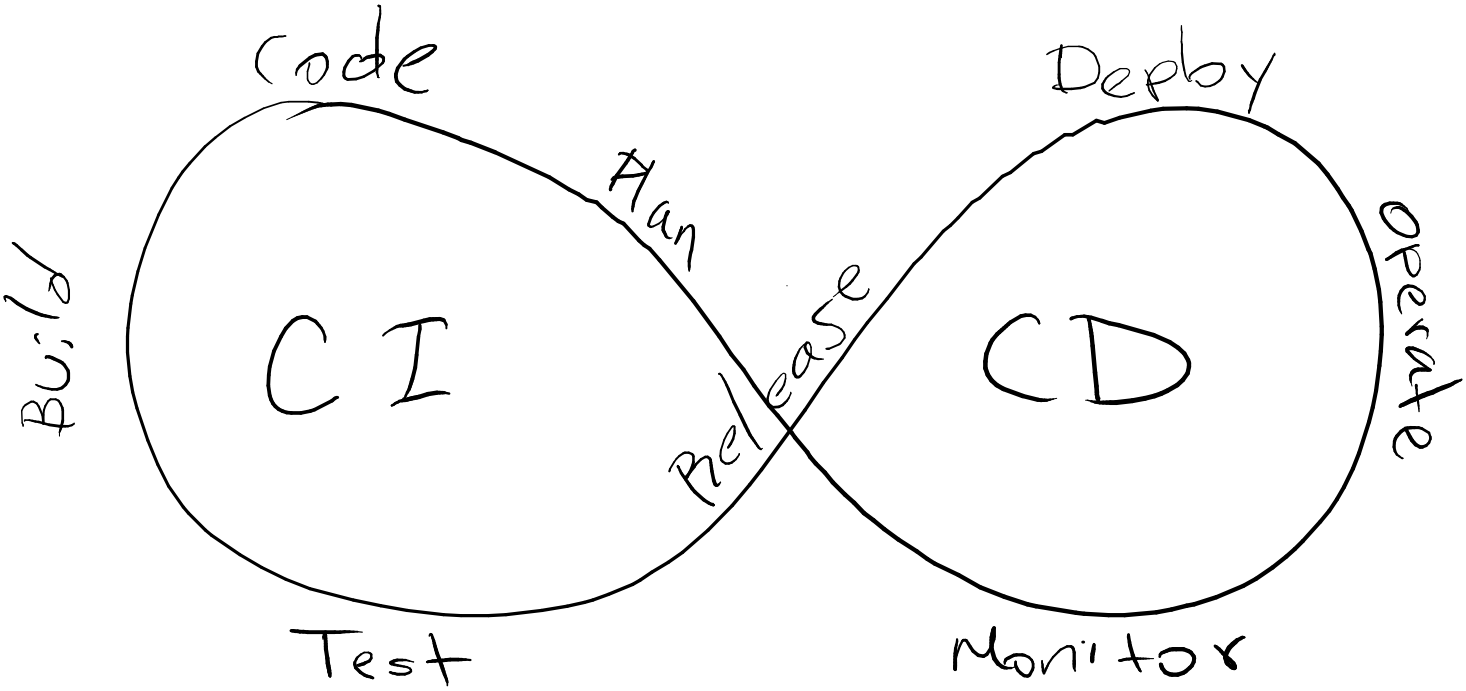
Properly managing dependencies with tools like pip, npm, or any other you prefer is crucial. Ensure you have the correct files for dependency management and, ideally, try to utilize dependencies that are up-to-date, actively maintained, not deprecated, and known for their security.

- well-maintained
- up-to-date
- Actively maintained
- Not deprecated
- Secure

9. CI & CD

Something you'll likely want to implement is a form of continuous integration and continuous deployment (CI/CD). Essentially, this involves automated testing using tools like GitHub Actions and automated deployment through the deployment service of your choice. The goal is to make it as easy as possible to run your application, typically in some kind of environment.

CI and CD demonstrate your understanding of the other side of software development: how to deploy applications and ensure they are tested and secure. This is an aspect that many junior developers may not be familiar with or may overlook, and it can truly set you apart.



10. Code Review, Refactoring

The last thing to mention here is the importance of reviewing and refactoring your code in the future. In any software project, code changes are inevitable—code is modified, removed, and added over time.

Once you've completed the project, take a few days to relax and step away from the code. Don't look at it for a while. Instead, focus on something else.

After a few days, return to your code and assess where you can make improvements and changes. There are almost certainly areas that can be modified or enhanced.

These improvements can range from simple tasks like adjusting indentation levels, organizing variables, or making certain variables constants. Sometimes, you'll only notice these opportunities if you take a step back from the codebase and don't obsessively dive back into it immediately.