Remove complexity of coding

Let's start with a simple truth - coding can be complex. As developers, we often find ourselves tangled in webs of dependencies, intricate algorithms, and lengthy codebases. But does it always have to be this way? The answer is no.

The core of coding is problem-solving, and the best solutions are often the simplest ones. This doesn't mean they are easy to find, but once found, they are easy to understand, implement, and maintain. As the renowned computer scientist Edsger Dijkstra once said, "Simplicity is a prerequisite for reliability."
So, why do we aim for simplicity?

First, simple code is easier to understand. When you or someone else revisits your code in the future, it will be much easier to grasp if it's simple and clean. This leads to less time spent on figuring out what the code does and more time spent on improving or expanding it.

Second, simple code is easier to debug and maintain. If something goes wrong, it's much quicker to identify and fix the problem in a simple codebase than in a complex one.
But how do we remove complexity in our code?

One key method is to approach problems from first principles. This means breaking down a problem to its fundamental truths and reasoning up from there. This approach can often reveal simpler solutions that might not be apparent if we just follow established patterns or preconceived notions.

For example, if you're tasked with creating a feature to upload photos, instead of immediately considering complex solutions involving large libraries or intricate data structures, start with the basics. What is the bare minimum needed to achieve this task? You need a way for users to select a file, a method to upload that file to a server, and a way to handle potential errors. Starting with these basic requirements and building up from there often leads to a simpler and more elegant solution.

Another way to remove complexity is to let go of personal bias. As developers, we often have favorite tools, languages, or methods that we're comfortable with.

But these may not always be the best fit for the problem at hand. By being open to other options, we can often find simpler and more effective solutions.

Remember, code is a tool to solve problems, not a canvas to showcase complexity. Strive for simplicity and clarity, and you'll not only become a better coder, but you'll also make life easier for yourself and those who work with your code.

How to remove (reduce) complexity of coding:

1. **Remove Personal Bias**. As developers, we often develop preferences for certain tools, languages, or approaches based on our previous experiences. While this is

natural, it's important to keep an open mind when approaching a new problem. Here's how you can work on removing personal bias.

   a. Recognize your bias. The first step is to acknowledge that you have a bias. Are you inclined towards a certain programming language or a specific library? Do you prefer certain methods over others?
   b. Question your bias. Once you've identified your bias, ask yourself why you have it. Is it because you're more comfortable with it, or is it truly the best tool for the job?
   c. Explore other options. Make a conscious effort to explore other tools or approaches. Read about them, try them out in small projects, and compare them objectively to your preferred tools.

2. **What is Easiest to Implement?** When considering different solutions, it can be helpful to think about what would be easiest to implement. Here's how you can do that.
   a. Break down the problem. We have an upcoming module about this and step by step on how to break down complex problems.
   b. Evaluate each step. Look at each step individually. What would it take to implement this step? What tools or techniques would you need?
   c. Consider the overall complexity. After evaluating each step, consider the overall complexity of implementing the solution. How many steps are there? How complex is each step? How long does it take me to implement it? What are my key missing gaps in implementing this?
3. **Looking at the Problem from First Principles.** This approach involves breaking down a problem to its most basic elements and reasoning up from there. Here's how to apply it.
   a. Identify the fundamentals. What are the basic truths or elements of the problem? What is absolutely necessary for a solution?
   b. Reason from the fundamentals. Once you've identified the fundamentals, start building up a solution from there. How can you achieve the necessary outcomes using the simplest methods?
   c. Challenge assumptions. Along the way, you might identify common practices or established patterns. Challenge these assumptions. Are they truly the best way to solve the problem, or is there a simpler approach?

This would be my thought process of building a web application for a logistics company. My go-to language for backend development had always been Python because of my familiarity and comfort with it. However, in this case, I recognized that bias and decided to evaluate other options too. Node.js was another viable option because of its non-blocking I/O and event-driven architecture, which could potentially handle real-time data more efficiently. I spent some time researching and comparing the two, and eventually chose Node.js as it seemed to offer better performance for this particular application. The problem could be divided into a few key steps: gathering location data from vehicles, sending this data to a server, and updating the client-side application in real-time. I evaluated several methods for each step. For example, for the real-time update, I considered polling the server at regular intervals, long-polling, and using WebSockets. After weighing the complexity, scalability, and reliability of each method, I decided on WebSockets because it was relatively straightforward to implement with Node.js and offered good performance. The system needed to receive location updates from each vehicle and display these updates in real-time to the user. I

reasoned that I needed a way to maintain a constant connection between the server and each vehicle, as well as between the server and the client-side application. This led me to choose a WebSocket-based solution over traditional HTTP requests. I also realized that I didn't need a heavy front-end framework for the client-side application. Instead, I could use vanilla JavaScript with a map library like Leaflet.js to keep it simple and lightweight.

Throughout this process, I was able to set aside my personal bias towards Python, choose the simplest and most efficient solutions for the task.

This is one of the most crucial things to do as a remote developer.