Street Coder Chapter 3 Notes - Ozan Sazak

- **3.1 - If it ain't broke, break it**
  - I mostly agree on the points of the author on abstraction. However the strictness and the granularity of the abstraction can be project-dependent, we should not practice over-abstraction
    - **Discussion point**: Where should we draw the line?
  - Strictly separated service layers introduce linearly increasing codebase update time for new functionalities / it also reduces the exponentially increasing refactor need and developer cognitive load
    - **Discussion point**: are these linear/exponential or different? how is your experience in your current/past companies?
    - I think the cognitive load is pretty important on scale (multiple teams, 50+ engineers etc.), what do you think about that?
- **3.1 (p. 60)** *"Every small bad decision will add seconds to your and your team's performance. Your throughput will degrade cumulatively over time. You will get slower and slower, getting less satisfaction from your work and less positive feedback from management. By being the wrong kind of lazy, you are dooming yourself to failure. Be the right kind of lazy: serve your future laziness."*
  - Most of the ideas discussed in this subtitle are mostly applied to companies on scale. There is a tradeoff in applying abstractions: you sacrifice an amount of architecture design and modularity maintenance time for easier functionality additions/removals. As the scale grows from a few engineers to thousand engineers, this tradeoff becomes more profitable IMO.
    - **Discussion point** (again): Where should we draw the line of the need of abstraction?
- **3.2.1 (p. 62)** *"Programming isn't about crafting things as much as it's about navigating the maze of a complex decision tree."*
  - This can be generalized to engineering. Engineering is the craft of moving in that complex decision tree for a specific problem, optimizing for a special function (runtime, memory, money, etc.) while trying to sacrifice as less as possible. It's better the problem is more granular and specific. Trying to find general solutions to general problems is the hardest part of this craft.
- **3.2.1 (p. 62)** *"...most of the time, you won't even need to look at your previous work. It's already in your mind, guiding you much faster, and without going into the same spiral of desperation this time."*
  - This is true in my personal experience. Building a solution for the same problem while even using the same algorithm yields better outputs in every trial, **but with diminishing**

improvements. At some point the improvements pass a certain point that another rewrite won't worth the time.

- **3.3 (p. 62)** *"Certain changes can be performed on code that aren't necessary but can help you in the long term. You can make it a regular habit to keep your dependencies up to date, keeping your app fluid, and identify the most rigid parts that are hard to change."*
  - Agree. Companies don't usually prioritize this stuff but this should be put to a calendar maybe as a regular engineering practice.
- **3.3.1 (p. 64)** *"My solution to this problem is to join the race toward the future. Keep the libraries up to date. Make it a regular habit to upgrade libraries. This will break your code occasionally, and thanks to that, you'll find out which part of your code is more fragile, and you can add more test coverage."*
  - I don't have too much code breaking experience on that, but it can be considered a good engineering practice to keep an eye on the change logs of major and minor versions of your dependencies. This can be programming languages, databases, important libraries, etc.
  - I disagree on actively upgrading all the libraries your project uses, of you take a look at even the most simple Go or JS project's `vendor` or `node_modules` folder you will see tons of libraries. Actively monitoring library versions and upgrading them and fixing related breaking code will introduce a burden.
  - **Discussion point**: Do you have a similar experience where a dependency broke the vendor versioning in your project? How do you prevent it?
- **3.6 (p. 74)** *"Then how do you reuse code? How do you inherit your class from an abstraction? It's simple—it's called composition. Instead of inheriting from a class, you receive its abstraction as a parameter in your constructor. Think of your components as lego pieces that support each other rather than as a hierarchy of objects."*
  - I agree. My takes on this:
    1. Multiple inheritance and over-abstraction seems cool when you start a small project, but it starts to become a burden again on scale. (increasing abstraction levels and number of abstractions depending on each other
    2. Depending on concrete implementation instead of abstract interface is a pretty bad anti-pattern. This is also considered a bad practice in Go: you should return concrete structs and accept interfaces (abstractions in Go). More info in book: 101 Go Mistakes and How to Avoid Them
- **3.7.2 (p. 79)** *"Structs are lightweight classes. They are allocated on stacks because they are value types. That means that assigning a struct value to a variable means copying its contents since no single reference represents it."*
  - That's not true for other languages and seems like a C#-bound information. You can (and mostly do) allocate structs on heap in other languages as C, Go or Rust (Go is a little

different though still heap alloc exists).

- **3.8.1 - Don't use If/else**
  - This is also a best practice in Go. Using else-less if statements where you early return on error conditions provides 10x more readable code. You can also see that the indented code is error processing code, while the no indent code is the safe path.
- **3.8.2 - Use goto**
  - Strong disagree. Goto is too open to be abused anywhere in the codebase. I personally would consider it useful only in a state machine implementation, and probably nothing else.
  - Also there are multiple clean goto-less ways to implement the example the author gave, such as a constantly defined common error value, an anonymous error handler function, etc. He also mentioned the `defer` keyword in Go
  - **Discussion point**: Any examples you used `goto` in a production system and it was beneficial?
- **3.9 - Don't write code comments**
  - My take on comments is the limit should be the need of context, or reasoning. Your code should be self descriptive as much as possible but also not as much as it starts becoming too verbose (more than normal length variable names for example).
  - An example of reasoning is: Let's say you have a timeout duration defined as a constant:

    ```go
    const (
        defaultTimeout = 5 * time.Minute
    )
    ```

    Another engineer may ask "Why 5 minutes is the default timeout?" and may need a concrete reasoning if that specific value causes some functionality to fail. Is there an upper limit? What was the needs of other systems depending on this one while this timeout value is put? These can be written as code comments.
  - **Discussion point**: What is your take on code comments? When should/shouldn't we use them? Is the project documentation enough? Should the code be self explanatory?