# Chapter 1 - Act with Prudence by Seb Rose

This part really stresses how crucial it is to deal with technical debt in software development. Basically, it's about resisting the urge to take shortcuts just to get things done fast. The author compares technical debt to a loan, it might help in the short term, but you'll end up paying more later. And the longer you ignore it, the worse it gets. It messes with adding new stuff, makes fixing things a headache, and leads to more bugs.

The author gets that sometimes you have to cut corners to meet deadlines or add important features. But they're all about keeping track of that debt and paying it off as soon as possible. If you do decide to cut corners, make sure to note it down and plan to fix it later. By budgeting time to pay off that debt and making sure everyone knows the cost, you can figure out if it's worth it and prioritize accordingly.

This chapter is saying it's smarter to do things right even if it takes longer, and you need to stay on top of fixing any shortcuts to avoid problems later on.

# Chapter 2 - Apply Functional Programming Principles by Edward Garson

This chapter is about how understanding and using functional programming principles can make software development better. It says that even though functional programming is often seen as something for multicore, it's actually super helpful for writing cleaner code in general.

One big thing it talks about is this idea called referential transparency, which basically means that functions always give the same results when you give them the same stuff, no matter when or where you use them. This is cool because it helps cut down on messy mutable stuff that can cause bugs in code.

This chapter mentions how a lot of bugs in code come from using too much mutable stuff, which is common in object-oriented programming. But it says you can get around that by using test-driven design and focusing on making functions that don't change stuff unnecessarily. By using smaller functions that work on inputs instead of changing variables all over the place, you can make your code cleaner and easier to debug. And apparently, learning a functional programming language can really help you get the hang of this. But it's not saying that functional programming is always the best. Sometimes, mixing it with object-oriented stuff works better, especially when you're dealing with complex business rules.

The chapter just says that learning about functional programming can make you a better programmer and that combining it with object-oriented stuff can give you the best of both worlds.

**Chapter 3 - Ask, “What would the User Do?” (You Are Not the User) by Giles Colborne**

This chapter is talking about how important it is to understand how regular people think and act when you're designing software or interfaces. It points out that us programmers don't think the same way as users do, and assuming they do can mess up our designs.

To get around this, the chapter suggests watching users directly. By seeing how they use similar software and listening to what they're thinking while they do it, designers can figure out how to make interfaces that work better for them. This helps spot common problems and guides the design process. It also talks about how users can get stuck when things aren't clear in an interface. Putting helpful hints or instructions where they're needed can really help them out. And sometimes, users come up with weird ways to do things, which shows the importance of making interfaces that are easy to understand and use.

Another thing it brings up is that what users say they want isn't always what they actually need. Instead of just asking them directly, it's better to watch what they do and learn from that. It's all about making sure software and interfaces are made with the users in mind. Watching users and really understanding how they do things can help make stuff that works better for everyone.

**Chapter 4 - “Automate Your Coding Standard” by Filip van Laenen**

This chapter is all about how tough it can be to keep everyone on the same page when it comes to writing code during a project. Basically, even if you start off with good intentions about how the code should look, things can get messy as the project goes on. The author talks about why it's important to have a set of rules for writing code, like making sure it all looks consistent and doesn't have common mistakes that lead to bugs. Following these rules should make it easier for everyone to work together and keep things moving smoothly.

To deal with the problem of keeping everyone following the rules, the author suggests using automation. This means setting up tools that automatically check the code as it's being written. They can fix the formatting, point out any mistakes, and even stop the whole project if something really bad is found. It's like having a helpful robot watching over your shoulder while you code. But the author also says that not everything can be automated, so there might still be some rules that need to be followed manually. The key is to automate as much as possible to keep things consistent and efficient.

They also mention that the rules should be able to change as the project goes on. What made sense at the beginning might not be the best way later on, so the rules need to be flexible enough to adapt. The main idea is to use automation to make sure everyone's following the same rules when writing code, which should lead to better quality code and make it easier for everyone to work together.