**LOUIE JAY A. CENIZA BSIT 4-1 FEBRUARY 21, 2024**

**(CHAPTER 21-30)**

# **DISTINGUISHED BUSINESS EXCEPTIONS FROM TECHNICAL**

In this chapter it talks about the importance of distinguishing between technical problems that prevent us from using the application and business logic that prevents us from misusing the application. particularly when it comes to exception handling.

It differentiates the two situations where technical problems refer to issues that arise due to programming errors, infrastructure failures, on another hand, business logic issues pertain to violations of domain-specific rules or constraints that prevent users from performing certain actions within the application. It's crucial to handle business logic exceptions separately from technical problems to ensure that they are addressed in a manner consistent with the application's intended behavior and domain requirements.

By distinguishing between technical problems and business logic issues in exception handling, developers can create more robust and maintainable software systems. Separating these concerns allows for clearer error handling strategies, better adherence to application requirements, and improved overall reliability and usability of the software.

# **DO LOTS OF DELIBERATE PRACTICE**

In this chapter it talks about practicing challenging yourself, explore something new where you are not good at, it tells us that doing deliberate practice means repeating the code until you know what is the purpose, why did you want to make it, is it to just complete the given task? or you want to improve your performance as a developer?

Also, they say it takes minimum of 10,000 hours to be called an elite in this industry, don’t think too much that you’re not good enough for this industry, you just need to practice deliberately, Practice makes it perfect is what it wants to say.

# **DOMAIN-SPECIFIC LANGUAGES**

In this chapter it talks about domain-specific languages (DSLs), which are specialized languages designed to address the needs and terminology of a specific domain, such as chess, education, or insurance. it tells us that every domain has its specific actions and expressions that are readable and understandable in this specific domain.

Also, discussed that there are 2 types of DSLs which are internal using its syntax to mimic natural language. They are often used to wrap existing APIs, libraries, or business code, providing a more user-friendly interface for accessing functionality on the other hand External DSLs, on the other hand, are separate languages expressed either textually or graphically.

By familiarizing DSLs, you need to consider the target audience, whether developers, managers, business customers, or end-users. The technical level of the language, available tools, syntax help, early validation, visualization, and representation should be related to the intended domain.

# **DONT BE AFRAID TO BREAK THINGS**

In this chapter it talks about the importance of do the necessary changes even if its temporarily breaking things, The analogy of the codebase being sick and in need of a doctor is used to highlight the idea that avoiding changes due to fear only allows the system's problems to worsen over time. utilize the refactoring and restructuring, even you can disrupt the flow of the system. Refactoring, redefining interfaces, and simplifying design can help reduce complexity, eliminate corner cases, and improve the overall health of the codebase.

It also tells us that break your whole code into small pieces into a function instead of large-scale changes all at once, so that you can test and reduce the risk of causing major problems to the system.

By adopting to this attitude, it will serve the system to be more maintained and cleaner. it will help your collaborators to prioritize the overall health of the codebase.

# **DONT BE CUTE WITH YOUR TEST DATA**

In this chapter it talks about how can your test data be harmful when it becomes public, it says that test data can be amusing and enjoyable to your part but when it comes visible to others it would likely receive a critique and you’ll bear the consequences.

Also, the author says that in todays interconnected world, mistakes or inappropriate content can quickly spread and become public knowledge.

By being mindful and maintaining professionalism on creating test data, comments or any text within the code you can overcome this problem.

# **DONT IGNORE THAT ERROR!**

In this chapter it talks about how errors can harm your codebase system if you ignore it always, it can be very harmful to the point that you can’t locate the bugs and errors and poor structure.

By giving attentions to your errors instead of ignoring it we need to find a way as soon as possible for the error to be fix not just I’ll do it later on because it will accumulate to the point where it is beyond saving.

# **DONT JUST LEARN THE LANGUAGE, UNDERSTAND ITS CULTURE**

In this chapter it talks about how important to understand the language and its uses, in short continuous learning but you need to grasp the language you want to learn not just memorizing the syntax you need to go beyond more that, you need to study their culture, its idioms, design patterns, and best practices. The author emphasizes that learning a new programming language each year, as suggested in "The Pragmatic Programmer," is valuable not only for expanding technical skills but also for gaining insights into different approaches to problem-solving and design.

By understanding and learning the language you need to immerse yourself on that language not just memorizing the syntax and formats, it can also help you understand the language that you already knew to enhance and broadening your perspective.

# **DONT NAIL YOUR PROGRAM INTO THE UPRIGHT POSITION**

In this chapter it talks about how the importance of simplicity and clarity in error handling strategies. It underscores the need to avoid overengineering solutions and instead choose a straightforward and effective approaches to manage exceptions. By replacing the overly complicated exception handling mechanism with a more streamlined and robust reporting system, the author and their team were able to enhance the overall reliability and maintainability of the codebase.

It serves as a reminder of the pitfalls of excessive complexity and the value of learning from past mistakes. It highlights the importance of continuously refining and improving software development practices, emphasizing the need for pragmatic and efficient approaches to error handling.

# **DONT RELY ON "MAGIC HAPPENS HERE"**

In this chapter it talks about how they view your work industry and think it is simple it also applies to you which you can that their work is simple. it also tells that thinking is the least appreciated, danger of assuming that certain aspects of application development or project management are simple or happen "by magic" when one is not actively involved in them.

By understanding your application that you develop you can say that in creating a codebase system you need to understand the code that you have created, not only that it works just fine so I won’t try to explore it more to understand what are the complex works within that function.

# **DONT REPEAT YOURSELF**

In this chapter it talks about the "Don't Repeat Yourself" (DRY) principle, which emphasizes the importance of avoiding duplication in software development. The main idea is that every piece of knowledge should have a single, authoritative representation within a system, whether it's in code, processes, or logic. the duplication can impact a negative consequence as it will cause a bloating to your codebase, increasing the potential for bugs, and making it harder to understand and maintain the system.

So, by applying the DRY principle to your codebase system, it can optimize and enhance your maintainability, readability, and overall software quality. Eliminating duplication reduces the risk of introducing bugs and inconsistencies, making it easier to understand and modify the codebase. Automation of repetitive processes streamlines development workflows, improving efficiency and reducing errors.