Nicole P. Satiembre **March 25, 2024**

Cebu Technological University-Main Campus

Clean Code

Chapter 7 - Error Handling

Some codes can be completely dominated by error handling and it is almost impossible to see what the code does because of all of the scattered error handling. This chapter is in connection to clean code, our purpose it make our codes as clear as possible.

Use Exceptions Rather than Return Codes. Returning an error code clutters up the caller. The caller must check for errors immediately after the call, unfortunately, it's easy to forget. This is why it is better to throw an exception when you encounter an error. The calling code is cleaner.

Write Your Try-Catch-Finally Statement First. From this passage, I've learned about the significance of exception handling in defining the scope and transactional behavior of code. The try-catch-finally statement establishes a transaction-like structure, ensuring that execution can recover from exceptions and leave the program in a consistent state. By starting with a try-catch-finally block during code development, developers can clarify expected behavior in case of exceptions and incrementally build and test the transaction scope, maintaining robust error handling throughout the development process.

Use Unchecked Exceptions. While checked exceptions initially seemed beneficial for enforcing error handling, it's now clear that they can introduce complexities and violate the Open/Closed Principle by requiring cascading changes to method signatures. This can lead to a cascade of updates throughout a codebase, breaking encapsulation and increasing dependency costs. While checked exceptions may be useful in critical libraries, for general application development, the drawbacks often outweigh the benefits.

Provide Context with Exceptions. Each exception that you throw should provide enough context to determine the source and location of an error.

Define Exception Classes in Terms of a Caller's Needs. Instead of duplicating exception handling code for various specific exception types, it's more efficient to wrap third-party APIs and translate their exceptions into a common exception type. This simplifies the code and makes it more maintainable. Wrapping third-party APIs also reduces dependencies and provides flexibility in choosing alternative libraries in the future. Additionally, by defining a common exception type for a specific area of code, the code becomes cleaner and easier to manage, unless there are specific scenarios where different exception types need to be distinguished and handled differently.

Define the Normal Flow. I've learned about the concept of the Special Case Pattern in error handling, which aims to simplify code by encapsulating exceptional behavior within special case objects. By designing classes or configuring objects to handle specific edge cases, such as when certain data is not available, the need for explicit error handling and cluttered logic can be minimized. This approach promotes cleaner and more straightforward code, where the bulk of the logic focuses on the main algorithm, with error detection pushed to the edges of the program.

Don’t Return Null. I've learned about the pitfalls of using null in code and how it can lead to error-prone and cluttered logic. Returning null from methods or handling null checks throughout the codebase can create maintenance headaches and increase the risk of NullPointerExceptions at runtime. Instead of returning null, it's recommended to consider alternatives such as throwing exceptions or returning special case objects. Additionally, utilizing methods like Collections.emptyList() in Java can help simplify code and minimize the chance of NullPointerExceptions by returning an empty collection rather than null. This approach promotes cleaner, more robust code and reduces the likelihood of runtime errors.

Don’t Pass Null. Passing null into methods is just as bad as returning null from methods. When you pass null as an argument, you’ll get a NullPointerException. A better approach would be to create a new exception type and throw it instead.

Conclusion. Clean code doesn’t only mean readable but it should also be efficient. One way to do this is if we see error handling as a separate concern, something that is viewable independently of our main logic.