Notas sobre o capitulo 7 de Clean Code – by Lucas Trevizan

Tratamento de erros

Diversos erros podem acontecer em projetos, uma entrada errada, um nullPointer, um SQL TimeOut entre outros.

É comum em um projeto ver um monte de tratamentos de erro espalhados pelo código todo, esse excesso de tratamento de erros pode obscurecer a lógica?

Use exceções em vez de retornar códigos

Antigamente algumas linguagens não tinham suporte a exceções, então era necessário tratar erros na "mão", durante toda a execução do código tinha que ser chamado uma função para informar erro se a flag de erro fosse acionada.

Isso é um problema de antigamente, então não tive contato com nada parecido, programando em projeto de Java eu sempre vi um bom uso da Exceção e como ela facilita nossa vida, então use exceções.

Crie primeiro sua estrutura try-catch-finally

Estruturas de try-cathc-finally ajudam seu código a se manter "coeso" durante a execução de uma operação que pode retornar uma exception.

O bloco try-catch-finally é basicamente uma definição de escopo para te ajudar a lidar com os erros e quem sabe até te dar um caminho na lógica e implementação, um bom exemplo dado no livro é se forçar a escrever testes que lançam exceções.

Use exceções não verificadas

Java tem bastante uso de exceções verificadas, de certa forma isso te ajuda a implementar métodos que tem erros previsíveis e que precisamos "tratar" ou "jogar" ao utilizar esses métodos.

A provocação que rola aqui é que esses erros verificados tem um preço, que preço? **A VIOLAÇÃO DO PRINCIPIO OPENCLOSED** *(Aberto fechado, é um princípio de SOLID).*

E essa provocação faz total sentido, se implementarmos um método que lança uma exceção, mas o catch dele está 3 níveis acima será necessário **DECLARAR UM TRHOWS** em todos os níveis entre o método e o catch. Isso quer dizer que uma modificação em um nível mais baixo do código pode forçar alteração em assinaturas no nível mais alto ***(Uma classe deve estar ABERTA PARA EXTENSÃO, MAS FECHADA PARA MODIFICAÇÃO).***

Exceções verificas são úteis para processos críticos, mas rola um questionamento se as dependências dela e essa "violação" de encapsulamento valem a pena. Eu sinceramente não sei se concordo tanto com esse tópico.

Forneça Exceções com contexto

Devemos fornecer exceções com informações suficientes para identificar onde foi o erro e qual o propósito da operação que falhou.

Defina as classes de exceções segundo as necessidades do chamador

Aqui temos um conceito de ***Wrapper(Empacotamento)*** a respeito de exceções, o exemplo se dá numa chamada de uma API externa ao projeto. Essa chamada pode jogar diversas exceções, então por que não as empacotar em uma única classe de exceção maior?

Isso faz com que o código fique mais limpo e cria facilidade em trocar essa API no futuro caso seja essa a vontade.

Defina o fluxo normal

Empacotando as exceptions e deixando que um "cara" acima cuide do que estourar mais abaixo você consegue criar um bom código que é capaz de dizer o caminho "correto", e o caminho do que é uma exceção.

Entretanto, e se eu tiver uma exceção onde eu queira não parar o programa, mas sim fazer outro tipo de processamento? Talvez não seja necessário deixar que sua exceção seja o fator que muda o comportamento do código, esse tipo de implementação é confuso (***Se der uma exceção aqui ou eu tiver um caso especial então faz um processamento diferente).***

Para esse tipo de situação seria melhor implementar o padrão "special case". Basicamente você deixa que um objeto ou classe trate o caso especial, assim o código não precisa lidar com esse processamento diferente, ele fica encapsulado em um objeto especial.

Não retorne null

Se você não deve retornar null então obviamente não deve aceitar valores nullos como parâmetro A NÃO SER que você esteja trabalhando com uma API que precisa de null.

O ponto é que se você receber null provavelmente vai ter problemas, então não aceite valor null nos seus métodos, ou no MINÍMO trate eles de uma maneira clara ao invés de só deixar um nullPointer estourar.

Conclusão

A robustez que tratamentos de erro trazem ao código não significa que ele deixou de ser limpo, na verdade é o contrário. Um código limpo tem que tratar erros com robustez, é isso que garante a manutenção dele.