CAPÍTULO 13 CONCORRÊNCIA.

RESUMO OBJETIVO:

O que é concorrência? - Técnica utilizada para executar tarefas em paralelo

Objetivo central: Melhorar desempenho e responsividade em sistemas.

Desafio: Garantir segurança e eficiência ao lidar com compartilhamento de recursos.

O objetivo primário da concorrência é permitir que múltiplas tarefas sejam executadas simultaneamente, otimizando recursos (ex.: CPU, tempo de resposta). Porém, como isso introduz riscos (race conditions, deadlocks), a segurança e a eficiência tornam-se requisitos críticos para alcançar esse objetivo.

No contexto de "Código Limpo", Uncle Bob destaca que, sem segurança e clareza, a concorrência se torna um pesadelo de manutenção. Logo, o foco do capítulo é:

* Evitar bugs difíceis de reproduzir (ex.: inconsistências em multithreads).
* Escrever código concorrente legível, mesmo que isso exija trade-offs de desempenho.

Um exemplo prático de uso de concorrência:

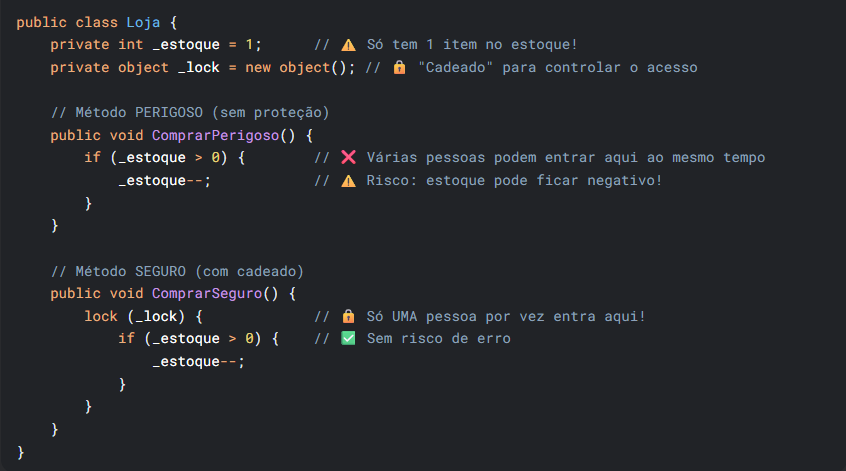
Uma loja online precisa processar pagamentos e atualizar o estoque simultaneamente.

Sem utilizar concorrência: As tarefas são lentas e sequenciais.

Utilizando concorrência mal implementada: Risco de vender produtos sem estoque (*race condition*).

Utilizando concorrência limpa: Tarefas rápidas, seguras e sincronizadas.

Cenário: Imagine uma última camiseta à venda em uma loja online.



Método sem proteção(ComprarPerigoso): Vários clientes clicam "Comprar" ao mesmo tempo. Todos veem "1 em estoque" e finalizam a compra → estoque vai para -2!

Método seguro(ComprarSeguro): O cadeado (lock) faz os clientes entrarem um de cada vez. O primeiro compra, o estoque vai para 0, e os outros veem "Esgotado!"

CONCLUSÃO CAPÍTULO 13: A concorrência no contexto do capítulo 13 tem como objetivo prático permitir a execução paralela de forma segura e eficiente, mas seu propósito mais amplo (na computação) é habilitar a simultaneidade de tarefas. O capítulo ensina como alcançar isso sem comprometer a qualidade do código.

**"O lock é como um semáforo que controla quem acessa o estoque, evitando caos!"**

CAPÍTULO 14: REFINAMENTO SUCESSIVO

RESUMO OBJETIVO:

O que é o Refinamento Sucessivo? - O objetivo do refinamento sucessivo é transformar código funcional, porém confuso (código sujo), em código limpo, legível e fácil de manter, por meio de melhorias incrementais e contínuas. É um processo de "lapidação" do código, onde pequenas alterações são feitas iterativamente para elevar a qualidade sem comprometer sua funcionalidade.

Principais Objetivos e ideias:

* Legibilidade: Tornar o código autodocumentado, com nomes claros e estrutura lógica.
* Redução de Complexidade: Dividir funções longas em métodos menores e especializados.
* Eliminar Duplicações: Aplicar o princípio DRY (*Don’t Repeat Yourself*) para evitar repetições.
* Facilitar Manutenção: Garantir que futuras alterações sejam rápidas e seguras, sem introduzir bugs
* Separação de Responsabilidades: Isolar regras de negócio, validações, parsing de dados e tratamentos de erro em componentes distintos.

Refinamento Sucessivo na prática:

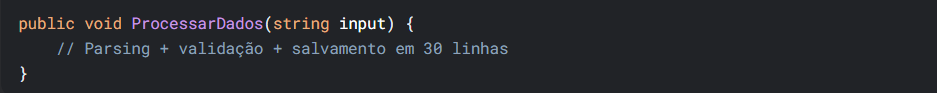
1- Identificar um trecho de código difícil de entender (ex.: uma função de 50 linhas).

2- Fazer pequenas mudanças (ex.: extrair um método, renomear variáveis).

3- Testar após cada alteração para garantir que nada quebrou.

4- Repetir até que o código esteja claro e organizado.

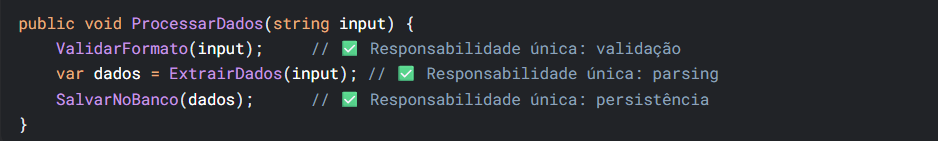
Código Sujo:



Problemas:

* Mistura de responsabilidades, o método faz tudo: parsing, validação e salvamento no banco. Viola o “Princípio da Responsabilidade Única”
* Complexidade Alta, mais de 30 linhas em um único método, difícil de ler e manter.
* Dificuldade para testar, testar validação separadamente do salvamento é impossível.

Código Refinado:



Melhorias:

* Separação de Responsabilidade: Cada método faz uma coisa bem feita. ValidarFormato: Verifica se o input está correto

ExtrairDados: Converte o input em um objeto estruturado. SalvarNoBanco: Persiste os dados no banco.

* Legibilidade: O código se autodocumenta: qualquer pessoa entende o fluxo em segundos.
* Facilidade de Manutenção: Alterar a validação não afeta o salvamento (e vice-versa)
* Testabilidade: Cada método pode ser testado isoladamente (ex.: testar validação sem precisar do banco).

Principais Benefícios do Refinamento:

* Legibilidade: Qualquer desenvolvedor entende o código rapidamente.
* Manutenibilidade: Alterar uma regra não afeta outras partes.
* Testabilidade: Métodos pequenos são fáceis de testar individualmente.
* Agiliza o Trabalho em Equipe: Todos entendem o código rapidamente.

***“Ninguém escreve código limpo de primeira. Código bom é resultado de múltiplas iterações de refinamento.”***

**– Robert C. Martin**

CAPÍTULO 15: CARACTERÍSTICAS INTERNAS DO JUNIT

OBJETIVO:  
Apresentar uma análise interna do módulo ComparisonCompactor do JUnit, demonstrando como código bem projetado pode ser testado, compreendido e mantido com clareza, mesmo tratando de lógica aparentemente complexa.

Origem do JUnit:

* O JUnit nasceu dentro de um avião, durante uma conversa entre Kent Beck e Erich Gamma.
* Em poucas horas, criaram os fundamentos do JUnit, combinando as ideias de testes de Kent em Smalltalk com a linguagem Java.

O que é o ComparisonCompactor?

* É uma parte engenhosa do JUnit usada para destacar diferenças entre duas strings que falham em um teste.
* Por exemplo, ao comparar "ABCDE" com "ABXDE", o módulo gera uma mensagem como:

<...B[X]D...> (Tenta separar esse trecho)

* Isso ajuda o programador a ver rapidamente onde está o erro, especialmente quando as strings são longas e parecidas.

Importância dos Casos de Teste:

* Em vez de explicar o comportamento do módulo com documentação extensa, o capítulo mostra que os próprios testes automatizados são a melhor forma de especificação.
* A Listagem 15-1 traz diversos testes que revelam os requisitos implícitos e os casos de uso reais do ComparisonCompactor.

CAPÍTULO 16: REFATORANDO O SERIALDATE

OBJETIVOS:

Demonstrar como realizar uma revisão e refatoração profissional de código legado (no caso, a classe SerialDate), utilizando testes automatizados como suporte para compreensão, validação e melhoria progressiva da base de código.

CONTEXTO INICIAL:

* O autor deixa claro que não se trata de crítica pessoal ao criador da classe (David Gilbert), mas sim de uma revisão técnica construtiva — como médicos, pilotos ou advogados fazem entre si.
* David, ao compartilhar a SerialDate publicamente, prestou um grande serviço à comunidade, criando uma classe útil que foca em datas, não em horas — um problema comum em APIs como java.util.Date.

PROBLEMA INICIAL:

* Embora existam testes unitários (SerialDateTests), eles cobrem apenas ~50% das instruções de código.
* Um exemplo claro é o método MonthCodeToQuarter(), que não é testado nem utilizado, sendo identificado por ferramentas como Clover.

ABORDAGEM DO AUTOR:

* Executar e validar os testes existentes — que passam, mas cobrem pouco.
* Usar Clover para obter uma análise de cobertura real da classe.
* Criar uma nova suíte de testes própria (Listagem B-4), com comportamento esperado e maior abrangência (atingindo ~92% de cobertura).
* Alguns testes foram inicialmente comentados, pois falhavam — representavam o comportamento idealizado, que o autor pretendia alcançar com a refatoração.

FILOSOFIA DA REFATORAÇÃO:

* Antes de modificar código legado, é fundamental escrevê-lo de forma testável e coberta por testes confiáveis.
* Compreensão precede refatoração: não se pode melhorar algo que você não entende completamente.
* Testes também podem expressar expectativas de comportamento, mesmo antes de estarem implementadas.

VANTAGENS

* A refatoração começa com testes abrangentes.
* Cobertura de testes é fundamental para segurança durante mudanças.
* Refatorar código legado é um processo que exige cautela, compreensão e iteração.
* Testes que inicialmente falham podem servir como especificação do comportamento desejado.

DESVANTAGENS

* Criar testes para código legado pode ser difícil se ele for mal estruturado.
* Pode haver resistência a modificar código que “já funciona”.
* Requer tempo e ferramentas adequadas (como Clover ou cobertura em .NET).

CAPÍTULO 17: MAUS CHEIROS E HEURÍSTICAS

Reúne uma lista prática de **"code smells" (maus cheiros de código)** e **heurísticas** organizadas por categorias. Esses "cheiros" são indícios de que o código pode estar mal projetado, difícil de entender ou manter. As **heurísticas**, por outro lado, são boas práticas que ajudam a evitar esses problemas.

O autor destaca que **bons desenvolvedores treinam sua sensibilidade** para detectar esses sinais e corrigir o código antes que ele se torne problemático.

### **1. Comentários**

* **Comentário bom explica o porquê**, não o como. O código já deve ser claro o suficiente para explicar como funciona.
* **Comentários desnecessários ou enganosos** são piores do que nenhum.
* **Comentários obsoletos** são perigosos, pois passam informações erradas.
* **Evite redundância** (comentários que repetem o que o código já diz).
* Use comentários para:  
  + Justificar decisões complexas.
  + Marcar partes do código que exigem revisão (ex: TODO).
  + Explicar regras de negócio específicas que não são óbvias.

**2. Ambiente**

* Um **sistema de build** precisa ser rápido e confiável.
* Builds complicados ou lentos desencorajam testes e integração contínua.
* Um bom ambiente de desenvolvimento deve permitir:  
  + Rodar todos os testes com um único comando.
  + Gerar a versão final com facilidade.
  + Feedback rápido sobre erros.

**3. Funções**

* **Funções devem ser pequenas** e fazer apenas uma coisa.
* Nomeie as funções de maneira clara e objetiva.
* Evite passar **muitos parâmetros** – isso indica responsabilidade mal distribuída.
* **Funções com efeitos colaterais** inesperados (alterar variáveis externas, por exemplo) são perigosas.
* Prefira **funções puras**, que recebem entradas e retornam saídas previsíveis.
* **4. Geral**
* **Evite duplicação de código**: sempre que encontrar partes repetidas, pense em como unificá-las.
* **Código morto** (que não é mais usado) deve ser removido.
* **Estilo consistente** (formatação, nomeação, espaçamento) facilita a leitura por qualquer membro da equipe.
* Prefira **clareza à complexidade**: uma solução simples e compreensível é melhor do que uma solução inteligente e obscura.

**5. Nomes**

* Escolha **nomes descritivos**: que revelem a intenção da variável, função ou classe.
* Nomes devem ser **pronunciáveis** e **pesquisáveis** (fáceis de encontrar no projeto).
* Evite abreviações sem sentido.
* Use um padrão consistente: se você usa get para acessar valores, use set para atribuir, e assim por diante.

**6. Testes**

* Testes devem ser:  
  + **Rápidos** – para rodar com frequência.
  + **Independentes** – não devem depender uns dos outros.
  + **Repetíveis** – devem sempre produzir o mesmo resultado.
  + **Automatizados** – para garantir consistência e agilidade.
* Testes mal escritos ou frágeis indicam **problemas de design** no sistema.
* Um bom conjunto de testes permite que você refatore com confiança.

**Conclusão**

O capítulo ensina que, assim como bons autores reescrevem textos para melhorar a clareza, **bons programadores refatoram continuamente seu código**. Os maus cheiros são pistas que algo pode estar errado, mesmo que o código "funcione". O objetivo é desenvolver **um faro para detectar esses sinais** e aplicar heurísticas que mantenham o código limpo, simples e sustentável a longo prazo.