Resumo - Código Limpo

Sumário

1. História Código Limpo

O *clean code* [1]é uma filosofia, uma proposta de atitudes e técnicas comportamentais e procedimentais que visam a construção de um código legível, simples e de fácil manutenção e escalabilidade.

Com base em sua experiência de mais de cinco décadas no trabalho de desenvolvimento de softwares, Robert Cecil Martin escreve e publica, em 2008, o livro que dá origem ao termo e cria um novo paradigma, o qual apregoa, entre outras, a máxima da simplicidade para a confecção de códigos com maior entrega de valor. Além disso, “Uncle Bob”, como é conhecido, é coautor do manifesto ágil, que foi uma das motivações para a escrita do livro.

Após os anos 2000, a maioria das linguagens de programação que conhecemos já havia sido criada, no entanto ainda não existia a preocupação de criar uma estrutura de normas para estabelecer um padrão de escrita de códigos. Dessa forma, essa demanda foi gerada com o surgimento de grandes times de desenvolvedores, ou seja, muitas pessoas trabalhando em um mesmo código. Abaixo estão alguns dos problemas enfrentados na época que motivaram a criação do livro:

- cerca de 80% do tempo de um desenvolvedor é usado para dar manutenção à códigos e não para criar novos.
- Códigos bagunçados dificultam a leitura e diminuem a produtividade.
- Códigos ruins são difíceis de gerenciar e geram custos desnecessários para as empresas.

Ainda mais, no livro “Clean Code”, Martin traz além da sua opinião profissional e percepção pessoal do que seria um código limpo, pontos de vista de importantes desenvolvedores, dessa forma o livro é separado em tópicos de problemas relacionados com código, e ainda mais, por subtópicos dentro desses tópicos. Portanto, entre a variedade de atributos que um código limpo precisa conter para ser considerado como tal, a seguir estão alguns:

Um Código limpo possui várias definições:

- Proporciona uma leitura natural;
- Faz bem apenas uma coisa;

- Elegante e eficiente;
- Bastante atenção aos detalhes;
- Simples e direto, deve expor claramente as questões do problema a ser solucionado;
- Facilita para que outras pessoas o melhorem;
- Possui testes unitários e poucas dependências;
- Sem duplicações, uma tarefa, expressividade e pequenas abstrações

Por fim, mesmo depois de mais de uma década da sua criação o livro se mantém como um grande sucesso na área, utilizado por desenvolvedores menos experientes até aqueles com anos de carreira. Isso se deve ao fato de o livro ser considerado como um código de conduta e boas práticas para a profissão. A seguir, será abordado de uma forma mais detalhadas os tópicos tratados no livro.

[1] “*Clean Code: a Handbook of Agile Software Craftsmanship”.* Ed. Prentice Hall PTR, 2008.

2. Criando um Código Limpo

2.1 Atitudes para alcançar um código limpo

> 💡 “Quem não faz bem pequenas coisas não faz bem coisa alguma.”

Antes de demonstrar tecnicamente como construir um código mais legível, *Clean Code* apresenta a atitude ideal que os envolvidos no trabalho com softwares devem ter para alcançar esse objetivo. As atitudes compõem-se de valores e concepções importantes que impactam e são impactados uns pelos outros, constituindo um ciclo vicioso da dívida técnica ou um ciclo da eficácia.

Ciclo vicioso da dívida técnica

O ciclo vicioso da dívida técnica gera custos para a empresa e o profissional. O tempo e recursos despendidos em tentativas de interpretação e refatoração tardia, além de representarem uma perda econômica, revelam a ocorrência de entregas de menor valor.

Ciclo da Eficácia

A atitude que leva ao ciclo da eficácia, por sua vez, é alcançada por meio de preceitos filosóficos, entre os quais pode-se mencionar:

A *regra do escoteiro*: Incentiva uma atitude proativa e colaborativa ao sugerir que programadores devem, à semelhança dos escoteiros, deixar o ambiente em que estão mais limpos ao saírem. Pode ser retirando uma duplicidade, realizando um teste ou melhorando a legibilidade do código, por exemplo.

A máxima de que *pequenas coisas importam*, por sua vez, sugere uma reflexão sobre os impactos positivos ou negativos da atitude do desenvolvedor. O código é, segundo esse princípio, uma forma de comunicação. Ele dever ser expressivo, simples e conter abstrações compreensíveis. Pequenas tarefas, realizadas com consistência, levarão à eficácia.

Na teoria da janela quebrada, é construída a analogia de que pessoas tendem a cuidar melhor de algo que já está sendo cuidado. Assim, ao encontrar um código limpo, o programador se esforçará para manter o padrão. Portanto, o bom desenvolvedor deve começar a limpar.

Esses e outros princípios permeiam as explicações práticas e técnicas sobre a construção de um código compreensível e de fácil manutenção que serão mencionadas nos próximos tópicos.

2.2 Nomes Significativos

> 💡 “Comunique. Não codifique.”

Os nomes têm o intuito de revelar o propósito de uma variável, função ou classe. Para tanto, sugere-se que sejam significativos. O nome de uma variável, função ou classe deve responder a todas as grandes questões. Isso facilita o entendimento e manutenção do código.

```
public class Banco {

    private String n;
    private List<Conta> c;
```

No primeiro exemplo, os nomes das variáveis não revelam seu propósito, o contexto não está explícito. Códigos escritos de tal forma são de difícil compreensão e não são passíveis de buscas.

```
public class Banco {

    private String nome;
    private List<Conta> contas;
```

Após refatorarmos, obtemos nomes legíveis, pronunciáveis, passíveis de busca e que revelam seu propósito.

Algumas dicas para escolher bons nomes:

Evitar:

- palavras cujos significados podem se desviar daquele que desejamos.
- codificação húngara, nomes codificados raramente são pronunciáveis, além da possibilidade de escrevê-los incorretamente.
- nomes parecidos para variáveis de contextos diferentes.

- números sequenciais em nomes.

Usar:

- nomes pronunciáveis, com palavras que representem o conceito proposto.
- nomes passíveis de busca: nomes de uma letra só ou números não são fáceis de localizar ao longo de um texto.
- substantivos para nomear classes e objetos.
- verbos para nomear métodos.

▼ **Exemplo**

```
public void depositar(double valor) {
    saldo += valor;
}
```

Nomeando adequadamente o método:

- Use nomes a partir do domínio da solução: não é prudente nomear a partir do domínio do problema, para não ter que consultar o cliente sempre para saber o significado de um nome, o qual o desenvolvedor já sabe o conceito, só que por outro nome.
- Use nomes de domínio do problema: quando não houver uma solução mais adequada.
- Contexto significativo: usar nomes que façam parte do contexto para o leitor. Para isso você os coloca em classes, funções e namespaces bem nomeados. Caso isso não seja suficiente, como último recurso adicionar prefixos aos nomes.

2.3 Funções

Se até esse momento as ações resolviam questões estéticas e didáticas do desenvolvimento de softwares, em funções elas começam a impactar o funcionamento do programa. Afinal, funções - e métodos - mal escritos entregam resultados técnicos ruins. Por isso, debruçar-se sobre como construir e descrever boas funções e dialogar com os preceitos abordados pela filosofia de código limpo tem sido o tema recorrente em livros posteriores.

Aqui, faz-se um recorte do que pode ser pensado ou repensado sobre o assunto.

O termo “função” já é conhecido por sua aplicação matemática, portanto o seu uso na linguagem de códigos é empregado justamente para descrever um método que se comporta como uma função matemática.
Uma função deve ser pequena (<= 20 linhas) e fazer apenas uma coisa. Às vezes, uma função precisa conter validações e etapas antes de realizar a ação proposta, nesse caso, avaliar quais são essas ações anteriores e destrinchá-las em outras funções atende mais adequadamente ao princípio da simplicidade e das pequenas abstrações.

```
public void verificarSaldoDoUsuarioAutenticado() {
    buscarUsuarioAutenticado();
    calcularSaldo();
}
```

Os blocos de condicionais e loopings devem conter poucas linhas, preferencialmente uma e que esteja fazendo a chamada de outra função. Evita-se, de tal modo, a construção de estruturas aninhadas e de difícil compreensão.

▼ **Não fazer**

```
String pokemon = "pikachu";
String[] ataques = {"Choque do trovão", "Calda de ferro"};

public void verificarPokemonELancarAtaque() {
    if (pokemon.equals("Pikachu")) {
        for (int i = 0; i < ataques.length; i++) {
            if (ataques[i].equals("Choque do trovão")) {
                System.out.println("Pikachu usou Choque do trovão!");
            } else {
                if(ataques[i].equals("Calda de ferro")) {
                    System.out.println("Pikachu usou Calda de ferro!");
                }
            }
        }
    }
}
```

▼ **Fazer**

```
String pokemon = "pikachu";
String[] ataques = {"Choque do trovão", "Calda de ferro"};

public void verificarPokemonELancarAtaque() {
    if (pokemon.equals("Pikachu")) {
        lancarAtaque();
    }
}

private void lancarAtaque() {
    for (int i = 0; i < ataques.length; i++) {
        String ataqueLancado = ataques[i];
        System.out.println("Pikachu usou " + ataqueLancado);
    }
}
```

Um nível de abstração por função é uma pequena abstração adequada, uma vez que diferentes níveis de abstração dentro de uma mesma função podem causar confusão no leitor, que não saberá identificar se um conceito é essencial ou apenas um detalhe.

Em todos os casos mencionados está a máxima de que o código eficiente deve se constituir em uma narrativa, na qual conforme lemos, avançamos para um nível mais baixo de abstração das funções.

Algumas dicas sobre funções:

- Nomes descritivos:

  → Ser consistente com os demais nomes de variáveis e funções que estão dentro do módulo, mantendo um padrão; → Para funções, o uso de verbos no infinitivo é adequado. Ex.: cadastrarPessoa().

- Parâmetros de funções:

  → A quantidade ideal de parâmetros de entrada é zero, seguido de um parâmetro (nômade), dois parâmetros (díade) e a quantidade que se deve evitar: três parâmetros (tríade);

→ Os parâmetros de entrada dificultam os testes já que, quanto mais parâmetros, maior será a variação de testes a serem realizados;

→ Parâmetros lógicos não devem ser usados pois mostram, explicitamente, que a função faz mais de uma coisa e fere, portanto, o princípio da responsabilidade única;

→ Funções díades podem ser criadas quando os parâmetros estejam se complementando ou possuam uma ordem pré-determinada natural (como em funções para criar um ponto em eixos cartesianos, por exemplo);

→ Se possível, alterar objetos dentro da função ao invés de retorná-los para depois fazer um tratamento.

- Efeitos colaterais:

  Efeitos colaterais, como alterações desnecessárias ou incompletas na aplicação, podem ocorrer ao chamar uma função que não faz apenas uma coisa. Nomear a função chamada de modo a revelar seu propósito e contexto e dar a ela uma única responsabilidade são algumas das soluções possíveis.

- Exceções e blocos try/catch:

  Prefira exceções a retorno de código de erro, desta forma, a exceção poderá ser tratada em outro código, simplificando a função.
  Não deve haver continuação de código após os blocos catch/finally.
  Blocos try/catch devem ser extraídos, isso é, devem conter somente chamadas das funções que terão responsabilidades, inclusive das exceções.

```
public void deletarPaginaEReferencias(Pagina pagina) {
    try {
        deletarPagina(pagina);
        deletarRegistro(pagina.nome);
        deletarConfiguracao(pagina.chave);
    } catch (Exception e) {
        exibirErro();
    }
}
```

- Evite repetições:

  Códigos repetidos geram problemas ao tentarmos reutilizá-los, pois para cada alteração feita no sistema será necessário modificar diferentes locais que

utilizam aquele trecho de código, bem como, há a possibilidade de omissão de erros.

2.4 Comentários

> 💡 “Qualquer um pode escrever código que o computador entenda. Bons programadores escrevem códigos que humanos entendem”
>
> Martin Fowler

De acordo com esse paradigma, um código confuso a ponto de ser necessário um comentário para interpretá-lo é indício da necessidade de refatoração.
Isso não significa que comentários são proibidos, ao contrário, alguns são relevantes e até mesmo indispensáveis para tornar a interpretação do código mais fluente. Cabe ao desenvolvedor entender o que é importante e o que é poluição em seu código.

Comentários relevantes:

- Comentários legais: Alguns padrões de programação corporativa tornam necessário o comentário de direitos autorais e/ou autoria.
- Comentários informativos: É preferível que um método informe a sua função com o nome, mas existem casos em que o comentário se torna vantajoso.

```
//Formato (string + @ + dominio)
Pattern pattern = Pattern.compile(regex);
```

- Explicação da intenção: Os comentários podem ir além de explicar apenas a funcionalidade do código, também é possível usá-lo para esclarecer a tomada de decisão por parte do desenvolvedor.
- Esclarecimentos: Quando o retorno de parâmetro ou valor não for claro ou fizer parte de uma biblioteca, o comentário guiará a uma interpretação adequada.
- Alerta sobre consequências: Um comentário visando alertar sobre alguma particularidade é relevante quando aplicado de modo lógico, como um aviso sobre

um teste que demore para ser executado, por exemplo.

- Destaques: Um comentário pode ser usado para reforçar a importância de algum trecho ou método que, à primeira vista, possa parecer irrelevante.

```
//A função salvarFilme é essencial para colocar as informações no banco de dados.
filmeService.salvarFilme(filme);
if (novo) {
    mv.addObject("filme", new Filme());
} else {
    mv.addObject("filme", filme);
}
```

- Comentário *"TO DO"*: Serve como uma bandeira das metas seguintes, podendo ser utilizado por desenvolvedores para criar tarefas a serem realizadas

```
//TODO Atualizar o método utilizando DTO
@RequestMapping
public ModelAndView listarFilme(String nome, String genero) {
    ModelAndView mv = new ModelAndView("filme/listar.html");
    if (genero != null) {
        mv.addObject("lista", filmeService.listarFilmePorGenero(genero));
    } else {
        mv.addObject("lista", filmeService.listarFilme(nome));
    }
    return mv;
}
```

Comentários irrelevantes:

- Murmúrio: Não explicativos, como se estivesse comentando para si mesmo.
- Redundâncias: Não adicionam nada à interpretação do código.

```
//Cria uma lista de emails
List<String> emails = new ArrayList<>();

//Adiciona emails na lista
emails.add("user@domain.com");
emails.add("user@domain.co.in");
emails.add("user1@domain.com");
emails.add("user.name@domain.com");
emails.add("user#@domain.co.in");
emails.add("user@domaincom");
```

- Comentários enganadores: Não informam com precisão o funcionamento do código, ou informam erroneamente.

- Comentários imperativos: Em uma tentativa de manter uma documentação (javadoc) em todas as funções, termina gerando comentários confusos, desorganizados e até mesmo errôneos.
- Marcadores de posição: Comentários utilizados para unir funções, porém, normalmente, são chamativos e se utilizados com frequência passarão a ser vistos como ruído e ignorados.

```
//////////////////////////
//Métodos
//////////////////////////
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}
```

2.5 Formatação

O trabalho do desenvolvedor é contar uma história. Para conseguir esse efeito, no entanto, um código deve ser bem formatado e seguir o planejamento prévio. A formatação, assim como a legibilidade, pode afetar a capacidade de herança e manutenção, mesmo tempos após o código original ter sido alterado. Por isso, desenvolver estilo, disciplina e ser consistente na construção de códigos podem torná-lo um profissional melhor.

Algumas questões sobre formatação:

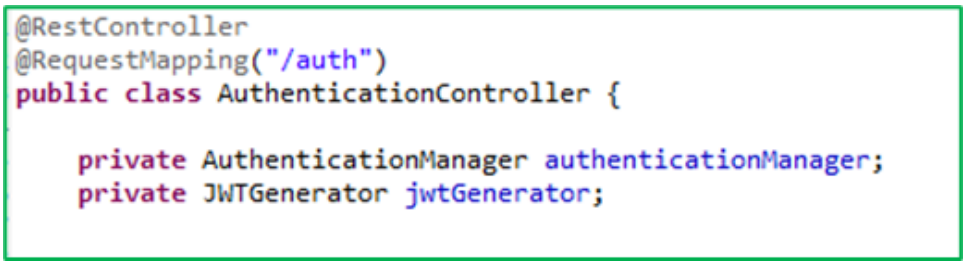
Formatação vertical

- Espaçamento vertical entre conceitos: cada linha representa uma expressão ou estrutura, e cada grupo de linhas representa um pensamento completo. Esses pensamentos devem ficar separados por linhas em branco.

```
@Transactional
public void excluir(Long id) {
    Produto produto = this.buscarProduto(id);
    repository.delete(produto);
}

@Transactional
public Produto atualizar(Produto produto, Long id) {
    Produto produtoOriginal = this.buscarProduto(id);
    produto.setId(produtoOriginal.getId());
    return repository.save(produto);
}
```

- Continuidade Vertical: linhas de código que estão intimamente relacionadas devem aparecer verticalmente unidas.
- Distância vertical:
  - Os conceitos intimamente relacionados devem ficar juntos verticalmente.
  - Não se deve separar em arquivos distintos conceitos intimamente relacionados, a menos que tenha uma boa razão para isso.
  - As variáveis devem ser declaradas o mais próximo possível de onde serão usadas.
  - Variáveis locais devem ficar no topo de cada função, já as variáveis de instância devem ser declaradas no início da classe, pois serão usadas por muitos, senão todos, os métodos da classe.



```java
@RestController
@RequestMapping("/auth")
public class AuthenticationController {

    private AuthenticationManager authenticationManager;
    private JWTGenerator jwtGenerator;
```

```java
@PostMapping("login")
public ResponseEntity<AuthResponseDTO> login(@RequestBody LoginDTO loginDTO){

    Authentication authentication = authenticationManager
                        .authenticate(new UsernamePasswordAuthenticationToken
                                (loginDTO.getUsername(), loginDTO.getPassword()));
    SecurityContextHolder.getContext().setAuthentication(authentication);
    String token = jwtGenerator.generateToken(authentication);


    return new ResponseEntity<>(new AuthResponseDTO(token), HttpStatus.OK);
}
```

Nos exemplos acima podemos ver que as variáveis de instâncias estão sendo declaradas no início da classe, e as variáveis locais no topo da função.

- Se uma função chama outra, elas devem ficar verticalmente próximas, e a que chamar deve ficar acima da que for chamada. Isso dá um fluxo natural ao programa.

- Funções com afinidades conceituais devem ficar próximas, por exemplo, funções que efetuam operações parecidas, ou que compartilham a mesma convenção de nomes.

```java
public Optional<Role> findByName(String name) {
    return roleRepository.findByName(name);
}

public Role findById(Long id) {

    Optional<Role> optional = roleRepository.findById(id);

    return optional.orElseThrow(() -> new EntityNotFoundException("Role not found!"));
}

public Page<Role> findAll(Pageable pageable) {
    return roleRepository.findAll(pageable);
}
```

Formatação horizontal:

- Recomenda-se usar 120 caracteres no máximo por linha.
- Usamos o espaço em branco horizontal para associar coisas que estão intimamente relacionadas e para desassociar outras fracamente relacionadas.
- Não colocar espaço entre os nomes das funções e os parênteses de abertura, pois estão intimamente relacionados.
- Um arquivo fonte é como uma hierarquia. Cada nível dessa hierarquia é um escopo, usamos a indentação para tornar visível a hierarquia desses escopos.

```java
public String getUsernameFromJWT(String token) {
    Claims claims = Jwts.parser()
                        .setSigningKey(SecurityConstants.JWT_SECRET)
                        .parseClaimsJws(token)
                        .getBody();
    return claims.getSubject();
}
```

Regras de equipe:

- Uma equipe de desenvolvedores deve escolher um estilo de formatação, e todos os membros devem usá-lo.

2.6 Objetos e Estruturas de Dados

- Abstração de dados: não bastam apenas interfaces ou métodos de acesso (escrita, ou setters, e leitura, ou getters).
- Anti-simetria data/objeto:
  - Os objetos usam abstrações para esconder seus dados e expõem as funções que os operam.
  - As estruturas de dados expõem seus dados e não possuem funções significativas.
  - Ambas as definições possuem natureza complementar, pois são praticamente opostas
  - O código orientado a objetos facilita a adição de novas classes sem precisar alterar as funções existentes.

```java
public interface Figura {
	public double area();
}


public class Retangulo implements Figura {
	private double largura;
	private double altura;

	public Retangulo(double largura, double altura) {
		this.largura = largura;
		this.altura = altura;
	}

	@Override
	public double area() {
		return largura * altura;
	}
}

public class Circulo implements Figura {
	private double raio;

	public Circulo(double raio) {
		this.raio = raio;
	}

	@Override
	public double area() {
		return Math.PI * raio * raio;
	}
}
```

  - O código procedimental, usado em estruturas de dados, facilita a adição de novas funções, sem precisar alterar as estruturas de dados existentes.

```java
public class Figura {
	public double area(Object figura) {
		if(figura instanceof Retangulo) {
			Retangulo retangulo = (Retangulo)figura;
			return retangulo.largura * retangulo.altura;
		} else {
			Circulo circulo = (Circulo)figura;
			return Math.PI * circulo.raio * circulo.raio;
		}
	}
}


public class Retangulo {
	public double largura;
	public double altura;
}


public class Circulo extends Figura {
	public double raio;
}
```

    - A situação estabelece a melhor abordagem a ser seguida.

- Lei de Demeter:
    - Um objeto não deve expor sua estrutura interna por meio de métodos acessores.
    - A Lei de Demeter diz que um método M de uma classe C só deve chamar os métodos de:
        - C
        - Um objeto criado por M
        - Um objeto passado como parâmetro para M
        - Um objeto dentro de uma variável de instância C

```java
String saidaDiretorio = contexto.getOpcoes().getRascunhoDiretorio().getCaminhoAbsoluto();
```

    - O código acima viola a Lei de Demeter.

- Carrinhos de trem
    - Cadeias de chamadas geralmente são consideradas descuidadas e devem ser evitadas
    - Na maioria das vezes é melhor separar assim:

```
Opcoes opcoes = contexto.getOpcoes();
Arquivo rascunhoDiretorio = opcoes.getRascunhoDiretorio();
String saidaDiretorio = rascunhoDiretorio.getCaminhoAbsoluto();
```

- Para saber se estamos violando a Lei de Demeter, depende se o contexto, opcoes e rascunhoDiretorio são objetos ou estruturas de dados. Se forem objetos, o conhecimento de seu interior é uma violação. Por outro lado, se forem estruturas de dado, é natural expor suas estruturas internas, então não é uma violação.
- O uso de funções de acesso confunde o tópico acima. Abaixo temos um exemplo de como poderíamos melhorar essa questão:

```
String saidaDiretorio = contexto.opcoes.rascunhoDiretorio.caminhoAbsoluto;
```

2.7 Tratamento de Erro

- Use exceções em vez de retornar códigos:
  - No passado, como muitas linguagens não suportavam exceções, você criava uma flag de erro ou retornava um código de erro. Essas técnicas sobrecarregavam as chamadas, pois deviam verificar os erros imediatamente após a chamada.
  - É uma boa prática lançar exceções quando o erro for encontrado.

▼ Não fazer

```
if(deletarUsuario(usuario) == E_OK) {
    if(deletarEnderecoUsuario(usuario) == E_OK) {
        sucessoAoDeletar();
    } else {
        logger.log("Endereço do usuário não deletado");
    }
} else {
    logger.log("Erro ao deletar usuário");
}
```

▼ **Fazer**

```
try {
    deletarUsuario(usuario);
    deletarEnderecoUsuario(usuario);
    sucessoAoDeletar();
} catch(Exception e) {
    logger.log(e.getMessage());
}
```

- Conforme exemplo acima, o código fica mais claro, pois o algoritmo para deletar o usuário e o tratamento de erro estão separados.

- Crie primeiro sua estrutura try-catch-finally:
  - Exceções definem um escopo dentro do seu código: o bloco try funciona como uma transação que pode ser cancelada e deve continuar no bloco catch.
  - É uma boa prática que as funcionalidades que podem gerar erros sejam iniciadas por blocos try-catch, auxiliando ao definir o que o usuário do código deve esperar, independente do que ocorre no bloco
- Use exceções não verificadas:
  - As exceções verificadas surgiram com a primeira versão do Java e a assinatura de todos os métodos listaria todas as exceções que seriam passadas ao chamador.
  - Verificar exceções viola o Princípio Aberto-Fechado (Open-Closed Principle) do SOLID e quebra o encapsulamento, pois no tratamento do *throw* é preciso enxergar detalhes da implementação da exceção.
- Forneça exceções com contexto:
  - Em Java, é possível capturar um stack trace de qualquer exceção, no entanto, ele não consegue dizer o objetivo da operação que falhou.
  - É uma boa prática criar mensagens de erros informativas, mencionando a operação que falhou e o tipo da falha, junto com as exceções.
- Defina as classes de exceções segundo as necessidades do chamador:

- A melhor prática que existe é empacotar APIs de terceiros. Você minimiza as dependências nelas, pode escolher migrar para uma biblioteca diferente no futuro sem grandes problemas e não fica preso às escolhas de um modelo de API de um fornecedor em particular.

- Defina o fluxo normal:
  - Às vezes você pode não querer cancelar quando ocorrer uma exceção. Para isso, criamos uma classe ou configuramos um objeto para que ele trate este comportamento de uma maneira especial, sem que o código do cliente precise lidar com o comportamento diferente. Isso é chamado de Special Case Pattern (Padrão de Caso Especial), de Fowler.

- Não retorne null:
  - Quando retornamos null, basta uma verificação ausente para uma aplicação agir fora do controle.
  - É uma boa prática lançar uma exceção ou retornar um objeto Special Case Pattern.

  ▼ **Não fazer**

```
List<Empregado> empregados = getEmpregados();
if(empregados != null) {
    for(Empregado e : empregados) {
        salarioTotal += e.getSalario();
    }
}
```

  ▼ **Fazer**

```
List<Empregado> empregados = getEmpregados();
for(Empregado e : empregados) {
    salarioTotal += e.getSalario();
}
```

  - No exemplo acima será retornado uma lista vazia em vez de poder retornar null e lançar NullPointerException.

    - O Java possui o Collections.emptyList() que retorna uma lista predefinida de imutável e pode ser usado para retornar uma lista vazia.

- Não passe null:
    - A menos que, trabalhando com uma API, espere receber null, evite passá-lo sempre que possível.
    - A abordagem lógica para lidar com um valor nulo passado acidentalmente seria proibir, por padrão, a passagem dele.

2.8 Testes de Unidade

As Três Leis do TDD (Test Driven Development):

1ª: não se deve escrever o código de produção até criar um teste de unidade de falhas.

2ª: não se deve escrever mais de um teste de unidade do que o necessário para falhar, e não compilar é falhar.

3ª: não se deve escrever mais códigos de produção do que o necessário para aplicar o teste de falha atual.

▼ **Saiba mais**

https://www.youtube.com/watch?v=qkblc5WRn-U

Os testes e o código de produção são escritos juntos, com os testes apenas alguns segundos mais adiantados. Devemos manter os testes limpos, pois são eles que mantém o código flexível, reutilizável e passível de manutenção. Quando um código possui testes, o desenvolvedor não possui medo de alterá-lo.

Uma das vantagens em se fazer testes é que criamos os códigos de uma forma para serem simples de testar, consequentemente os códigos são mais desacoplados, mais fáceis de ler e entender, o que facilita muito a sua manutenção e reduz o tempo gasto com depuração.

O que torna um código limpo? Legibilidade!

Para um código legível ele deve possuir essas três características:

- Clareza;

- Simplicidade; e
- Consistência de expressão.

Cada função de teste deve possuir apenas uma instrução de afirmação (assert), e testar apenas um conceito, dessa forma, os testes chegam a uma única conclusão que é fácil e rápida de entender.

▼ **Não fazer**

```
@Test
void deveFrearVeiculoSeEleEstiverLigadoEComVelocidadeAcimaDeZero() throws Exception {
    veiculo.setVelocidade(100);
    int velocidadeAtual = veiculoService.frear(veiculo);
    assertEquals(80,velocidadeAtual );

    veiculo.setVelocidade(10);
    int velocidadeAtual2 = veiculoService.frear(veiculo);
    assertEquals(0, velocidadeAtual2);

}
```

▼ **Fazer**

```
@DisplayName(value = "O veículo pode acelerar somente se estiver ligado.")
@Test
void deveAcelerarSeVeiculoEstiverLigado() {
    int velocidadeAtual = veiculoService.acelerar(veiculo);
    assertEquals(20, velocidadeAtual);
}
```

Testes limpos seguem outras cinco regras que formam o acrônimo em inglês F.I.R.ST. (Fast, Independent, Repeatable, Self-validating, Timely).

- **Fast – Rapidez:** os testes devem ser executados com rapidez.
- **Independent – Independência**: os testes não devem depender uns dos outros. Você deve ser capaz de executar cada teste de forma independente e na ordem que desejar.
- **Repeatable – Repetitividade**: deve-se poder repetir os testes em qualquer ambiente, seja no de produção, no de garantia de qualidade.

- **Self-Validating – Autovalidação**: os testes devem ter uma saída booleana. Se os testes não possuírem autovalidação, então uma falha pode ser tornar subjetiva e executá-los pode exigir uma longa validação manual.
- **Timely – Pontualidade**: os testes precisam ser escritos em tempo hábil, imediatamente antes do código de produção. Se criá-los depois, o teste do código de produção poderá ficar mais difícil.

2.9 Classes

- Organização de classes:

Por convenção uma classe deve começar com uma lista de variáveis, se houver constantes estáticas públicas, devem ser declaradas primeiro. E depois as variáveis estáticas privadas. As funções devem ser criadas de acordo com a lista de varáveis declaradas, utilizar variáveis publicas apenas quando necessário.

- Encapsulamento em relação aos testes:

Gostaríamos de manter as variáveis e as funções de utilidades privadas, porém não devemos ser tão radicais quanto a isso, pois pode interferir no teste, que é uma parte importante do sistema. Devemos dar um jeito de testar, por vezes afrouxando o encapsulamento, mas esse afrouxamento deve ser usado em último caso.

- Tamanho das classes:

As classes devem ser pequenas, essa é a regra que devemos adotar. Porém, diferente do tamanho das funções que é medido em quantidade de linhas, as classes são medidas por quantidade de responsabilidades. Então para sabermos o tamanho de uma classe devemos contar quantas responsabilidades ela tem.

O nome de uma classe deve descrever suas responsabilidades. Um bom nome irá definir bem essas responsabilidades. Devemos ter em mente que quanto mais ambíguo for o nome da classe maior será a probabilidade de aumento de responsabilidades da mesma.

- Princípio da responsabilidade única (SRP):

Esse princípio afirma que uma classe deve ter apenas um objetivo e apenas um motivo para mudar. Esse princípio pode ser usado como norteador para definirmos o tamanho das classes.

Muitos desenvolvedores pensam que o programa está pronto assim que funciona, e não se preocupam com a limpeza do código e com a divisão de responsabilidades das classes. Muitos por medo de que o entendimento do quadro geral ficar mais difíceis com muitas classes com propósito único. Porém deve-se fazer a seguinte pergunta “você quer que suas ferramentas sejam organizadas em caixas de ferramentas com muitas gavetas pequenas, cada uma contendo componentes bem definidos e rotulados? Ou você quer algumas gavetas nas quais você simplesmente joga tudo?”

Todo sistema tem suas complexidades, a proposta é organizar essas complexidades do sistema. Cada pequena classe terá um motivo para mudar. Cada pequena classe irá encapsular uma função.

- Coesão:

As classes devem ter um pequeno número de variáveis de instância. Cada um dos métodos de uma classe deve manipular uma ou mais dessas variáveis. Quanto mais variáveis um método manipula, mais coeso esse método é para sua classe. Uma classe na qual cada variável é usada por cada método é maximamente coesa.

Não é aconselhável nem possível criar classes tão coesas ao máximo, porém deve-se manter a coesão alta. Quando a coesão é alta, significa que os métodos e variáveis da classe são codependentes e permanecem juntos como um todo lógico.

A estratégia de manter as funções pequenas e as listas de parâmetros curtas pode às vezes levar a uma proliferação de variáveis de instância que são usadas por um subconjunto de métodos. Quando isso acontece, quase sempre significa que há, pelo menos, uma outra classe tentando sair da classe maior.

- Isolando para a Mudança:

Qualquer aplicação está sujeita a mudanças, e, portanto, seus algoritmos também. E por isso que devemos ter em mente que, uma classe que depende de classes concretas está em risco quando esses algoritmos mudam. Para isso podemos introduzir interfaces e classes abstratas para ajudar a isolar o impacto que esses detalhes causam.

Um sistema deve ser desacoplado o suficiente para ser bem testado. E com isso ele também será mais flexível e promoverá mais reutilização. A falta de acoplamento significa que os elementos do nosso sistema estão mais bem isolados uns dos outros e da mudança. Esse isolamento facilita a compreensão de cada elemento do sistema.

Ao implementar essa prática estaremos aplicando automaticamente um outro importante princípio o de design de classe conhecido como Princípio de Inversão de Dependência (DIP em inglês).

> 💡 “Qualquer um pode escrever código que o computador entenda. Bons programadores escrevem códigos que humanos entendem”
>
> Martin Fowler

3. Referências

MARTIN, C. Robert. et al. Código Limpo. Código Limpo: habilidades práticas do agile software. Rio de Janeiro, RJ: Alta Books, 2011.

HORN, Michelle. Clean code: o que é, porque usar e principais regras! Betrybe, 2021. Disponível em: https://blog.betrybe.com/tecnologia/clean-code/. Acesso em: 30/11/2022.

IntelliJ IDEA by JetBrains. The Three Laws of TDD (Featuring Kotlin). Youtube, 2017. Disponível em: https://www.youtube.com/watch?v=qkblc5WRn-U. Acesso em: 30/11/2022.

MATURANO, Lorenzo. Código Limpo: Classes. Medium, 2020. Disponível em: https://medium.com/@maturanolorenzo/c%C3%B3digo-limpo-classes-731f3db96016. Acesso em: 01/12/2022.

LISURA, Leandro. Clean Code (Código limpo) – Classes. Leandro Lisura, 2021. Disponível em: http://leandrolisura.com.br/clean-code-codigo-limpo-classes/. Acesso em: 01/12/2022.

BALTIERI, André. Clean Code - Guia e Exemplos. Balta.io. Disponível em: https://balta.io/artigos/clean-code#regras-para-fun%C3%A7%C3%B5es-ou-m%C3%A9todo. Acesso em: 02/12/2022.