Código Limpo:

Habilidades Práticas do Agile Software

Capítulo 10: Classes

Este material foi desenvolvido pelo grupo PET/ADS do IFSP São Carlos

Classes: código limpo para classes

Os capítulos anteriores mostram importantes elementos de um código limpo, tais como: nomes significativos para variáveis, escrever funções pequenas e claras, etc.

Entretanto, não se faz um código limpo sem aprender como escrever classes limpas. Os próximos slides mostram um dos padrões mais aceitos para a organização de classes.

Organização das classes

Recomenda-se seguir a estruturação de classes do padrão Java. Para utilizar esse padrão, escreva os componentes de sua classe na seguinte ordem:

1. Variáveis públicas, estáticas e finais (constantes);
2. Variáveis privadas estáticas;
3. Instâncias privadas;
4. Funções públicas;
5. Funções privadas.

Com a ordem “de cima para baixo”, o entendimento da classes se torna mais fácil. O programa será lido como um artigo de jornal.

Organização das classes: exemplo

Exemplo de classe seguindo o padrão Java

```
public class ExemploClasse {

    public static final double CONSTANTE = 3.14;
    private static String variavelEstaticaPrivada = "VariavelEstaticaPrivada";
    private String instanciaPrivada;

    public void funcaoPublica() {...}

    private void funcaoPrivada() {...}
}
```

Encapsulamento

O ideal é que todas variáveis e funções fossem privadas, mas há momentos em é preciso torná-las protegidas (protected) para permitir testes (no Clean Code, os testes têm prioridade).

É importante ressaltar que raramente há um bom motivo para existirem variáveis públicas em uma classe. Perder o encapsulamento sempre deve ser um último recurso!

As classes devem ser pequenas: o quão pequenas?

Assim como as funções, as classes devem ser pequenas! Porém, a definição do tamanho das classe não se dá por quantidade de linhas, mas pelo número de responsabilidades.

As classes devem possuir apenas uma responsabilidade, e essa responsabilidade deve ser bem descrita pelo nome da classe a qual pertence.

As classes devem ser pequenas : orientações

Uma classe pequena não deve ter um nome ambíguo. Deve ser possível descrevê-la brevemente usando em torno de 25 palavras, sem conectivos como “e”, “se”, “mas”, etc.

Se essas regras não se aplicarem, sua classe ainda não está pequena o suficiente. Você precisará reescrevê-la até que seja possível comprir com as orientações acima.

As classes devem ser pequenas: exemplo de classe

A classe DashBoard está grande?

```
public class Dashboard extends JFrame implements MetaDataUser {
   public void addWidget()
   public void removeWidget()
   public void listWidgets()
   public void showInfos()
   public void saveData()
   public void recoverData()
   public void setCurrentUser()
   public void saveEvent()
   public void saveError()
}
```

As classes devem ser pequenas: analisando a classe

Sim, a classe DashBoard é muito grande! Ela tem cinco responsabilidades:

- Gerenciar widgets do painel;
- Exibir informações do painel;
- Manipular dados no DataStore;
- Gerenciar informações do usuário;
- Registar eventos e erros.

Logo, é preciso dividi-la em cinco diferentes classes.

O *Single Responsibility Principle (SRP)* estabelece que as classes devem ter uma única responsabilidade, um único motivo para ser mudar.

O princípio da responsabilidade única: classe melhorada

Agora temos uma classes com responsabilidades únicas.

```
public class  WidgetManager {
   public void  addWidget(){}
   private void removeWidget(){}
   private List<Widget> removeWidget(){}
}

public class DashboardUI{
   private void showInfos(){}
}

public class DataStore{
   private void saveData(){}
   private void recoverData(){}
}
```

```
public class ManageUserInfo {
   public void setCurrentUser() {}
}

public class RegisterErrorOrEvents{
   public void saveEvent(){}
   public void saveError(){}
}
```

O princípio da responsabilidade única: por que utilizar?

O SRP é frequentemente deixado de lado por causa da pressão por prazos, não realização das revisões e refatorações ou por medo de se lidar com muitas classes.

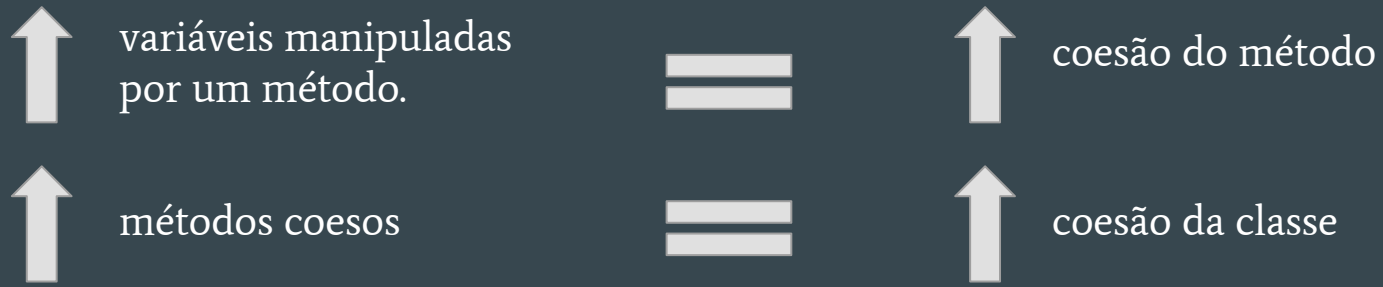
Entretanto, o princípio facilita trabalhar com programas em expansão, pois isola as responsabilidade das partes, melhorando a abstração do conjunto do programa.

Com o SRP, tem-se um código mais organizado e compreensível, que facilita manutenções e alterações futuras.

Você prefere um caixa de ferramentas sem repartições, com tudo misturado, ou uma caixa de ferramentas com várias repartições, onde tudo está organizado?

Coesão: o que significa?

Classes coesas possuem um número pequeno de atributos, que são muito utilizados por todos os métodos da classe.



A baixa coesão de uma classe é um ótimo indicador de que é necessário desmembrá-la em classes menores.

Coesão: código de exemplo (parte 1)

A classe Stack, que representa o tipo abstrato de dado Pilha, é uma classe bastante coesa. Dos seus três métodos, apenas o método *size()* não utiliza as duas variáveis.

```
public class  Stack {
  private int topOfStack = 0;
  private List<Integer> elements = new LinkedList<Integer>();

  public int size() {
     return topOfStack;
  }

// continua...
```

Coesão: código de exemplo (parte 2)

```
// continuação

   public void push(int element) {
     topOfStack++;
     elements.add(element);
   }

   public int pop() {
     if (topOfStack == 0)return;
        int element = elements.get(--topOfStack);
     elements.remove(topOfStack);
     return element;
   }
}
```

Manutenção de resultados coesos

Há momentos do desenvolvimento no qual há necessidade de refatorar um classe do sistema. Por exemplo, quando esta possui uma função muito grande que deve ser dividida em várias menores.

Contudo, essas refatorações devem ser feitas com calma e aos poucos, para manter-se a validade dos resultados. Para isso, é necessário realizar testes a cada pequena alteração no programa.

Esse cautela se faz necessária, pois, é possível que suas classes também precisem ser reescritas, deixando seu código mais extenso.

Não se deve ter medo dessas situações de refatoração, se forem bem feitas. Elas proporcionam classes mais coesas, com estruturas mais simples e inteligíveis.

Como organizar para alterar

A maioria dos programas é constantemente alterada com o passar do tempo. Assim, bons sistemas têm classes estruturadas para reduzir possíveis riscos decorrentes das futuras alterações.

Siga o Princípio Aberto-Fechado - OCP (Open/Close Principle), no qual um sistema ideal deve ser aberto à extensão e fechado para alterações.

Construa classes que exijam apenas adição de novos métodos e novas subclasses. Assim, não será necessário reorganizar variáveis e desmembrar classes velhas em novas classes.

Para construir classes com o OCP, precisa-se respeitar o SRP, mantendo a melhor coesão possível. Ter entidades simples, interfaces diminutas e testes unitários facilitados também é necessário.

Como isolar as alterações

Para isolar alterações, é necessário criar sistemas fracamente acoplados.

O princípio da Inversão da Dependência (DIP) prega que as classes devem depender de abstrações ao invés de implementações concretas.

Para que o DIP seja implementado efetivamente, será preciso usar interfaces e classes abstratas nos programas.

Como isolar das alterações: exemplo - Parte 1

A classe “Portfolio” teria problemas para ser testada caso fosse vinculada diretamente a variável “exchange” de uma API de bolsa de valores.

```
public Portfolio {
   private StockExchange exchange;
   public Portfolio(StockExchange exchange) {
      this.exchange = exchange;
   }
}
```

Por adotar a Interface “StockExchange”, Portfolio não depende mais de valores voláteis de uma API externa. Com isso, é possível escrever testes confiáveis e isolar alterações provenientes da API.

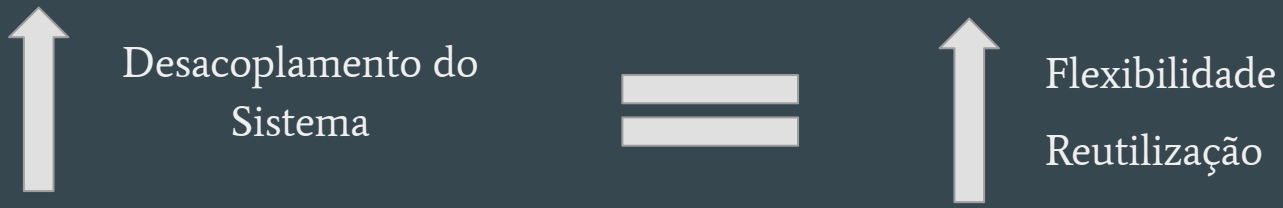
Como isolar das alterações: exemplo - Parte 2

```
public class PortfolioTest {
   private FixedStockExchange exchange;
   private Portfolio portfolio;

   @BeforeEach
   private void setUp() throws Exception {
      exchange = new FixedStockExchangeStub();
      exchange.fix("MSFT", 100)
      portfolio = new Portfolio(exchange);
   }
   @Teste
   private void GivenFiveMSFTTotalShouldBe500() throws Exception {
      portfolio.add(5, "MSFT");
      Assert.assertEquals(500, portfolio.value());
   }
}
```

Como isolar das alterações: facilitador de testes

Os artifícios anteriores (OCP e DIP), além de isolarem impactos de alterações, também facilitam os testes.

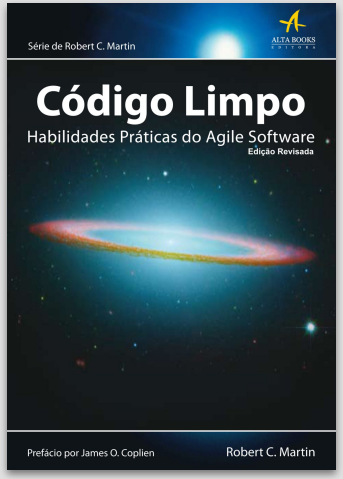


Sobre o autor

Robert C. “Uncle Bob” Martin é desenvolvedor e consultor de software desde 1990. Ele é o fundador e o presidente da Object Mentor, Inc., uma equipe de consultores experientes que orientam seus clientes no mundo todo em C++, Java, C#, Ruby, OO, Padrões de Projeto, UML, Metodologias Agile e eXtreme Programming.

Este conjunto de slides foi elaborado a partir da obra:

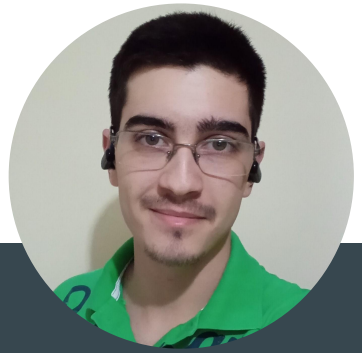
MARTIN, Robert. **Código Limpo:**
Habilidades Práticas do Agile Software. 1. ed.
Rio de Janeiro: Alta Books, 2009.



A equipe

Eduardo Derisso, Autor

Aluno do segundo período, é bolsista do PET/ADS desde abril de 2023.
LinkedIn



Lucas Almeida, Revisor

Aluno e bolsista do PET/ADS desde julho de 2023
LinkedIn

Lucas Oliveira, Tutor

Professor de Computação, é tutor do PET/ADS desde janeiro de 2023.
LinkedIn