Universidade de Brasília Faculdade UnB Gama

Curso: Engenharia de Software
Disciplina: Técnicas de Programação em Plataformas Emergentes **Turma: 01**
Semestre: 2022-02
Professor: André Luiz Peron Martins Lana

Alunos

Arthur Manuel Florêncio Sena	**180030345**
Isadora Soares	**180122606**
Jaime Juan de Castro Feliciano Damasceno	**180102711**
Victor Samuel dos Santos Lucas	**180028685**
Vinícius Vieira de Souza	**170115500**
Wagner Martins da Cunha	**180029177**

TRABALHO 03

1. Simplicidade

Uma das características mais importantes de um código bem escrito é a Simplicidade. Ela permite que um código seja fácil de ler, facilitando o processo de manutenção. Em síntese, essa propriedade permite que um código seja coerente, consistente, escrito de maneira enxuta e de fácil legibilidade.

Relação da característica com mau-cheiros: Esta característica está relacionada diretamente ao mau-cheiro de código muito longo, em inglês, *bloaters*, ou seja, é um código que cresceu demais. Além disso, outros maus-cheiros são relacionados a essa característica, sendo eles:

- Método longo;
- Classes grandes;
- Longa lista de parâmetros;
- Aglomerados de dados;
- Cadeias de mensagens;

Operação de refatoração capaz de levar o projeto de código a ter a característica em análise:

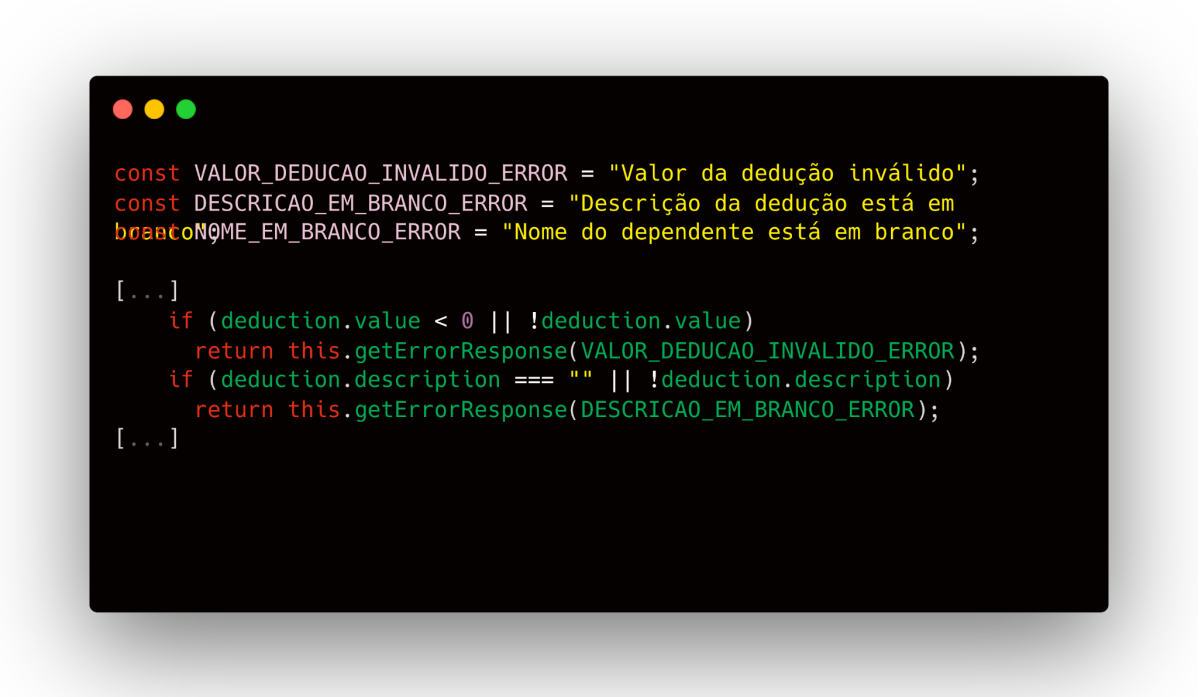
- Extrair métodos para tornar o código mais claro, como o "calculateDeductionTotals":

```
export class Deduction {
  ...
  createDeduction(deduction: IDeduction){
    ...
    const total =
this.calculateDeductionTotals(this.deductionlist);
    ...

    return res_data;
  }

  calculateDeductionTotals(deductionlist: IDeduction[]) {
    const totalDeduction = deductionlist.reduce(function (
      accumulator,
      deduction
    ) {
      return accumulator + deduction.value;
    },
    0);
    return totalDeduction;
  }
  ...
}
```

- Extrair mensagens de erro para constantes ao invés de repetir a string de erro nas várias exceções, evitando erros de digitação e facilitando mudar as mensagens de erro no futuro:



```
const VALOR_DEDUCAO_INVALIDO_ERROR = "Valor da dedução inválido";
const DESCRICAO_EM_BRANCO_ERROR = "Descrição da dedução está em
branco";
const NOME_EM_BRANCO_ERROR = "Nome do dependente está em branco";

[...]
    if (deduction.value < 0 || !deduction.value)
      return this.getErrorResponse(VALOR_DEDUCAO_INVALIDO_ERROR);
    if (deduction.description === "" || !deduction.description)
      return this.getErrorResponse(DESCRICAO_EM_BRANCO_ERROR);
[...]
```

2. Elegância:

Dizer que um projeto de software é elegante pode ter vários sentidos já que há muitas interpretações sobre o que seria esse aspecto. No entanto, é comum observar alguns certos padrões que facilitam identificá-lo como elegante: Ser de fácil interpretação, ele possui um código correto no sentido de fazer o que deve fazer sem situações previstas e sabe lidar com situações imprevistas. Também observamos esse aspecto de elegância muitas vezes relacionado com a simplicidade. Por mais que um problema seja complexo, a elegância vem em resolver esse problema de forma simples, limpa e clara.

Relação da característica com mau-cheiros: Como foi explicado na descrição desta característica, elegância se encontra em vários aspectos no código, se relacionando com diversos tipos de maus cheiros também. Um deles é a chamada complexidade artificial. Este mau cheiro se trata do uso de Design Patterns (Padrões de Design) extremamente complicados para um problema mais simples.

Operação de refatoração capaz de levar o projeto de código a ter a característica em análise: Um exemplo de técnica de refatoração que contribui para essa característica é *Parametrizar Método.*

Antes:

```
export interface IIncome {
    value: number;
    description: string;
}
const data: IIncome[] = [];
export interface IResponse {
    statusCode: number;
    response: IIncome[];
    totalIncomes: number;
}

export function calculateIncomeTotals(data: IIncome[]) {
    try {
        const totalIncome = data.reduce(function (accumulator, income) {
            return accumulator + income.value
        }, 0);
        return totalIncome;
    } catch (error) {
        return error
    }
}

export function registerIncome(income: IIncome) {
    try {
        if (income.value <= 0 || !income.value) throw new Error('ValorRendimentoInvalidoException');
        if (income.description === "" || !income.description) throw new
Error('DescricaoEmBrancoException');
        data.push(income);
        const total = calculateIncomeTotals(data);
        const res_data: IResponse = {
            statusCode: 200,
            response: data,
            totalIncomes: Number(total)
        }
        return res_data;
    } catch (error) {
        return {
            statusCode: 400,
            response: error
        }
    }
}
```

Depois:

```
export interface IIncome {
  value: number;
  description: string;
}

export interface IResponse {
  statusCode: number;
  response: IIncome[];
  totalIncomes: number;
}

export function calculateIncomeTotals(data: IIncome[]) {
  try {
    const totalIncome = data.reduce((accumulator, income) => accumulator + income.value, 0);
    return totalIncome;
  } catch (error) {
    return error;
  }
}

export function registerIncome(data: IIncome[], income: IIncome) {
  try {
    if (income.value <= 0 || !income.value) throw new Error("ValorRendimentoInvalidoException");
    if (income.description === "" || !income.description) throw new
Error("DescricaoEmBrancoException");
    data.push(income);
    const total = calculateIncomeTotals(data);
    const resData: IResponse = {
      statusCode: 200,
      response: data,
      totalIncomes: Number(total),
    };
    return resData;
  } catch (error) {
    return {
      statusCode: 400,
      response: error,
    };
  }
}
```

Aqui transformamos as funções do reduce para arrow functions, o que é mais conciso e elegante e também parametrizamos a função registerIncome para que ela receba o array data como um parâmetro, em vez de estar diretamente ligada a ele. Isso permite trabalhar com múltiplos conjuntos de dados independentes, caso seja necessário.

3. Boas interfaces:

Uma interface pode representar uma biblioteca, uma classe, um módulo ou qualquer estrutura que apresenta operações públicas com a implementação internalizada. Para se criar uma boa interface, identifique o que o cliente quer(ou precisa) fazer, o que a implementação pode de fato fazer, qual o tipo adequado de interface e quais operações são realmente necessárias. Boas interfaces apresentam uma **separação** clara entre o que o cliente pode acessar e a implementação da mesma, **abstraem** o “o que fazer” de “como fazer”, **comprimem** código, para transformar operações inteiras em poucas chamadas, e são **substituíveis** ao nível de implementação.

Considere, por exemplo, uma biblioteca para interação com a rede. Bibliotecas deste tipo apresentam interfaces com funções ou métodos para abrir conexões, se conectar a outro computador, enviar e receber mensagens. O programa que utiliza estas bibliotecas só pode usá-las através destas funções disponíveis em suas

interfaces. A implementação é uma caixa preta, que inclusive muda dependendo do sistema operacional.

Relação da característica com mau-cheiros: Interfaces podem evitar ou apresentar maus cheiros de código, a listar:

- Longa Lista de Parâmetros: Funções de interfaces com muitos parâmetros podem diminuir a qualidade da interface, uma vez que a função se torna difícil de usar.
- Cirurgia com rifle: Boas interfaces devem modificar a implementação sem modificar o que está disponível ao cliente. Se a interface não for consistente para o cliente, o código sofrerá mudanças em várias partes.
- Generalidade especulativa: Pode acontecer de uma interface apresentar mais operações do que é realmente necessário, em tentativa a ter a maior generalização possível. É importante considerar o que é realmente importante de se ter implementado.
- Biblioteca de classes incompleta: Pode acontecer de uma interface não apresentar alguma funcionalidade desejada. Uma interface é considerada boa se atende as necessidades de seus clientes.

Operação de refatoração capaz de levar o projeto de código a ter a característica em análise: Uma operação de refatoração que contribui para a característica de Boas Interfaces é a operação *Incorporar classe*.

4. Ausência de Duplicidades:

A ausência de duplicidades no código significa que não há código repetido ou duplicado em todo o projeto. Isso é importante porque torna o código mais fácil de ler, manter e testar. Além disso, a ausência de duplicidade ajuda a garantir que as correções feitas em um lugar não afetem outro lugar onde o código repetido está sendo usado. A ausência de duplicidades no código tem vários efeitos positivos na qualidade do código. Primeiro, torna o código mais fácil de ler e entender, pois não há partes repetidas que possam causar confusão. Além disso, torna o código mais fácil de manter e testar, pois todas as correções precisam ser feitas apenas em um lugar, ao invés de em vários lugares onde o código repetido está sendo usado. A ausência de duplicidade também ajuda a garantir a coesão do código, pois todas as partes relacionadas a uma determinada funcionalidade estão juntas e não espalhadas pelo projeto. Isso torna o código mais modular e menos acoplado, facilitando a manutenção do código a longo prazo.

Relação com Maus-Cheiros de Código: A ausência de duplicidade está relacionada ao mau cheiro de código "Copy and Paste" de Martin Fowler. O mau cheiro "Copy and Paste" refere-se ao uso excessivo de código repetido, que prejudica a clareza, a manutenção e a coesão do código.

Operações de Refatoração:

- Extrair Método: Quando o código repetido é encontrado, é possível extrair o código repetido em um método separado e chamá-lo a partir de vários lugares no código.
- Extrair Classe: Quando o código repetido é muito complexo e pode ser visto como uma entidade separada, é possível extrair esse código repetido em uma classe separada e usá-la a partir de vários lugares no código.

5. Portabilidade:

A portabilidade é um aspecto importante no desenvolvimento de um software, pois determina sua capacidade de ser executado em diferentes ambientes, sistemas operacionais ou plataformas hardware sem necessidade de alterações significativas no código. Isso significa que o software pode ser usado em uma ampla gama de dispositivos e sistemas, tornando-o mais acessível e útil para um maior número de usuários.

Além disso, a portabilidade pode ajudar a prolongar a vida útil de um software, uma vez que ele pode ser usado em sistemas atuais e futuros sem precisar ser completamente reescrito ou atualizado. Isso pode ser especialmente importante para software crítico que precisa funcionar sem interrupções ou erros.

A portabilidade também pode ser uma vantagem competitiva para o desenvolvedor do software, pois pode aumentar a base de usuários e, consequentemente, a receita gerada. Além disso, a portabilidade pode tornar o software mais atraente para potenciais investidores ou parceiros comerciais, já que mostra que ele é escalável e preparado para o futuro.

Conforme Martin Fowler, “maus cheiros” são pontos em que há possibilidades para aplicação de refatoração, e como supracitado, a portabilidade é um fator importantíssimo para se atentar para não haver tais “maus cheiros”. Necessitando assim de uma análise prévia para o desenvolvimento de um bom projeto.

Em resumo, a portabilidade é crucial para o sucesso de um software, pois permite que ele seja acessível para um público mais amplo, tenha uma vida útil mais longa e seja mais competitivo no mercado, além de se encaixar nas características propostas por Martin Fowler.

Antes:

```
  createDeduction(deduction: IDeduction){
    try {
      if (deduction.value < 0 || !deduction.value)
        throw new Error("ValorDeducaoInvalidoException");
      if (deduction.description === "" || !deduction.description)
        throw new Error("DescricaoEmBrancoException");

      this.deductionlist.push(deduction);
      const total =
this.calculateDeductionTotals(this.deductionlist);
      const res_data = {
        statusCode: 200,
        response: this.deductionlist,
        totalDeduction: Number(total),
      };

      return res_data;
    } catch (error: any) {
      return {
        statusCode: 400,
        error: error,
      };
    }
  }
```

Depois:

```
  createDeduction(deduction: IDeduction){
    if (deduction.value < 0 || !deduction.value
        || deduction.description === "" || !deduction.description)
{     return {
        statusCode: 400,
        error: new Error("Input inválido"),
      };
    }

    this.deductionlist.push(deduction);
    const total =
this.calculateDeductionTotals(this.deductionlist);
    return {
      statusCode: 200,
      response: this.deductionlist,
      totalDeduction: total,
    };
  }
```

- A validação de entrada foi simplificada e agrupada em uma única condicional.

- A conversão do total de dedução para Number não é mais necessária, pois o resultado da função calculateDeductionTotals já é do tipo number.
- Foi eliminado o uso do try-catch, que não é uma boa prática para validação de entrada e pode dificultar a identificação do erro. Em seu lugar, utilizamos uma condicional para retornar um objeto de erro com statusCode 400 em caso de entrada inválida.