## INSTITUTO INFNET

### ESCOLA SUPERIOR DE TECNOLOGIA

### GRADUAÇÃO EM ENGENHARIA DE SOFTWARE


# Teste de Performance 1

# Refatoração

### [25E2_4]

**Professor Rafael Cruz**
**Estudante João Victor Cícero de M. T. Ramos**
**Maio de 2025**

---

# Repositório

> O repositorio referenciado por todas as questões pode ser encontrado atravéz do seguinte link do **Github** :
> https://github.com/jvcmtr/Refactoring_DR4_AT

---

# 1 Refatoração prática de código simples

Você foi designado para revisar um componente antigo de um sistema de classificação que imprime mensagens com base em um número inteiro fornecido. O código legado apresenta nomes obscuros e mistura de responsabilidades, dificultando a manutenção.

```java
public class X {
    public void y(int z) {
        if (z > 10) {
            System.out.println("ALTO");
        } else if (z == -9999) {
            System.out.println("CASO RARO");
        } else {
            System.out.println("BAIXO");
        }
```

```
        int temp = z * 0 + 42;
        System.out.println("DEBUG: z = " + z);
        if (z > 10 && z > 5) {
            System.out.println("ALTO");
        }
    }
}
```

1. Refatore o código acima aplicando boas práticas de nomes, clareza e responsabilidade única. Use nomes semânticos, extração de métodos e evite decisões de negócio embutidas em métodos genéricos.
2. Adicione um novo comportamento que imprima "MÉDIO" quando o valor for exatamente 10, mantendo a clareza do código e testabilidade. Identifique ao menos dois *bad smells* presentes no código original e indique como a sua refatoração os resolveu.
3. Implemente um teste automatizado que valide os três comportamentos. Apresente o código do teste e evidencie sua execução com uma screenshot ou saída do terminal.

## 1 Refatoração

- Remove variavel temporaria, condicional redundante e print de Debug
- Renomeia classe `X` para `classificator` e `y()` para `ClassifyInt()`
- Extrai logica para classes separadas ( estrutura de `Classification` e demais classes filhas )
- Implementa estrutura de controle **Declarativa** para ordenar os diferentes tipos de classificação (propriedade `classifications` )

questao1 > src > main > java > questao1 > Classificator.java > ...

You, 2 minutes ago | 1 author (You)

```java
public class Classificator{

    // ⚠ WARNING
    /// Be aware that the order of the list also defines the priority of each classification in case of conflict
    private final List<Classification> classifications = Arrays.asList(
        new RareCaseClassification(),
        new HighClassification(),
        new LowClassification()
    );

    public void ClassifyInt(int integer) {

        Classification foundClassification = new NullClassification();

        for (Classification c : classifications) {
            if( c.matchClassification(integer) ){
                foundClassification = c;
                break;
            }
        }

        System.out.println( foundClassification.getName() );
    }
}
```

## 2.1 Inclusão de "Classificação Média"

questao1 > src > main > java > questao1 > classifications > MediumClassification.java > Java > MediumClassification

```java
package questao1.classifications;



public class MediumClassification extends Classification {
    private final String NAME = "MÉDIO";

    @Override
    public boolean matchClassification(int number){
        return number == 10;
    }

    @Override
    public String getName(){
        return NAME;
    }
}

```

questao1 > src > main > java > questao1 > Classificator.java > Language Support for Java(TM) by Red Hat > Classificator > classifications

```java
public class Classificator{

    // ⚠ WARNING
    /// Be aware that the order of the list also defines the priority of each classification in case of conflict
    private final List<Classification> classifications = Arrays.asList(
        new RareCaseClassification(),
        new HighClassification(),
        new MediumClassification(),       You, 3 minutes ago • Uncommitted changes
        new LowClassification()
    );

    public void ClassifyInt(int integer) {

        Classification foundClassification = new NullClassification();

        for (Classification c : classifications) {
            if( c.matchClassification(integer) ){
                foundClassification = c;
                break;
            }
        }

        System.out.println( foundClassification.getName() );
    }
}
```

questao1 > src > main > java > questao1 > App.java > ...

```java
You, 1 second ago | 1 author (You)
package questao1;

You, 1 second ago | 1 author (You)
public class App {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {

        var x = new Classificator();
        x.ClassifyInt(integer:10);
    }
}

```

PROBLEMS 18 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

```
--- exec:3.1.0:exec (default-cli) @ questao1 ---
M◆DIO
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.784 s
Finished at: 2025-06-18T18:20:12-03:00
------------------------------------------------------------------------
```

## 2.2 bad smells e melhoria

### Código duplicado

Podemos encontrar o seguinte código duplicado na versão original, dentro da função `y()`:

```
if (z > 10) {
    System.out.println("ALTO");
}
// ...
if (z > 10 && z > 5) {
    System.out.println("ALTO");
}
```

A versão refatorada corrige este problema excluindo a parte duplicada

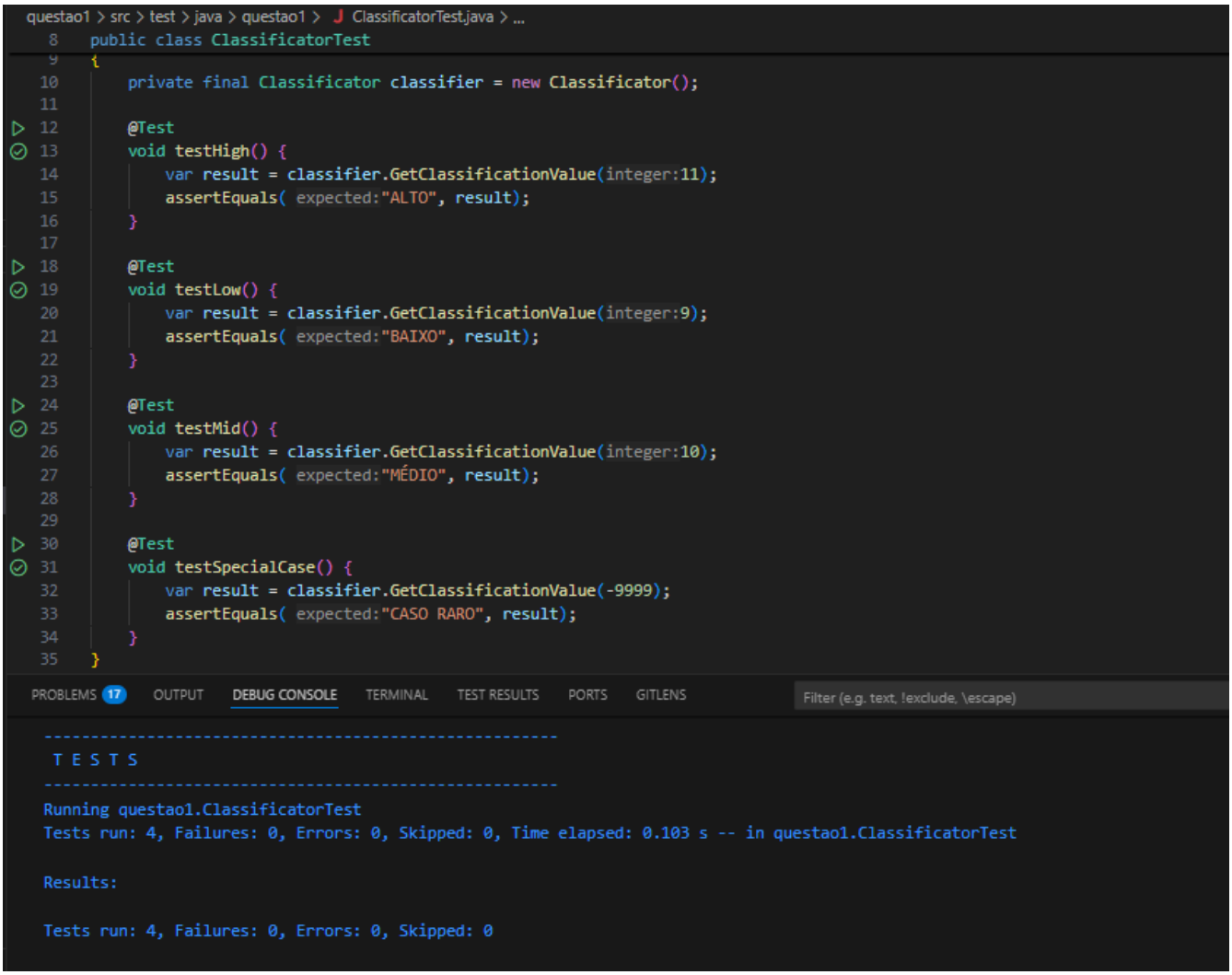
### Numeros Mágicos

O código original possui numeros magicos dentro de cada condicional:

```
if (z > 10)
// ...
else if (z == -9999)
// ...
if (z > 10 && z > 5)
/// ...
```

A versão refatorada corrige este problema isolando cada validação dentro de sua propria classe de classificação, dando contexto a cada numero.

## 3 Casos de Teste




---

# 2 Identificando Bad Smells

Você recebeu o trecho de código abaixo como parte de um sistema fiscal legado. A equipe de desenvolvimento tem enfrentado dificuldades para mantê-lo, e solicita uma revisão técnica que identifique problemas de design e sugestões de melhoria. O código está em produção, mas mudanças simples têm causado erros inesperados.

```
public class Invoice {
    public String clientName;
    public String clientEmail;
    public double amount;
    public int type;

    public void enviarPorEmail(String email, String conteudo) {
        System.out.println("Enviando email para: " + email);
        System.out.println("Conteúdo:\n" + conteudo);
    }

    public void process() {
        if (clientEmail == null && !clientEmail.contains("@")) {
            System.out.println("Email inválido. Falha no envio.");
        }

        if (type == 1) {
```

```
            System.out.println("Nota fiscal simples");
        } else if (type == 2) {
            System.out.println("Nota fiscal com imposto");
        } else if (type == -1) {
            // caso nunca ocorre, mas está presente
            System.out.println("Nota fiscal fantasma");
        } else {
            System.out.println("Tipo desconhecido");
        }
        //imprimir nota
        System.out.println("--- NOTA FISCAL ---");
        System.out.println("Cliente: " + clientName);
        System.out.println("Valor: R$ " + amount);

        if (type == 1) {
            System.out.println("Tipo: Simples");
        } else if (type == 2) {
            System.out.println("Tipo: Com imposto");
        } else {
            System.out.println("Tipo: Desconhecido");
        }
        System.out.println("----------------------");
        // Enviar nota para email
        System.out.println("Enviando nota fiscal para: " + clientEmail);
        String nota = "--- NOTA FISCAL ---\n" +
              "Cliente: " + clientName + "\n" +
              "Valor: R$ " + amount + "\n" +
              "Tipo: " + (type == 1 ? "Simples" : type == 2 ? "Com imposto" : "Desconhecido") +
"\n" +
              "----------------------";
        enviarPorEmail(clientEmail, nota);
    }
}
```

1. Liste os "bad smells" presentes nesse código, associando cada um a um tipo específico do catálogo de code smells. Mostre o máximo que encontrar.
2. Escolha **um** desses problemas e descreva uma refatoracão que poderia eliminá-lo. Use código para ilustrar.

# 1 Bad Smells identificados

### Atributos Publicos

as propriedades `clientName`, `clientEmail`, `ammount` e `type` estão como publicas mesmo isso não sendo nescessário.

### Metodo muito longo

O metodo `process()` é longo demais, apontando que detem muitas responsabilidades diferentes

### Falta de divisão de responsabilidades.

Na maneira como está implementado, a classe `Invoice` fica responsável por:

- Armazenar as informações do cliente ( `cientName` e `clientEmail` )
- Enviar e validar o email do usuario
- Implementar o comportamento de cada tipo de invoice (condicionais que avaliam a propriedade `type` )

### Código inutilizado

O caso com valor `-1` para a propriedade `type` é definido mas nunca é utilizado

### Exesso de valores primitivos

a variavel `type` é a resposável por definir qual string será usada para exibir a informação ao usuario, mas está implementada como sendo um `int` em vez de uma string

### Código repetido

O codigo monta a string da nota fiscal duas vezes. como pode se perceber pela strings duplicadas:

- `--- NOTA FISCAL ---`,
- `"Cliente: " + clientName` ,
- `"Valor: R$ " + amount`,
- `"Tipo: "` + string definida com base na propriedade `type`

## 2 Descrevendo a Refatoração

Levando em conta o Bad Smell **Código repetido**, uma sugestão que pode eliminar este bad smell é declarar uma variavel `invoiceString` inicialmente (em vez de printar ela diretamente na tela) e posteriormente reutilizar esta variavel para as ações da função `process` (Enviar nota para email e imprimir nota), como no exemplo a seguir:

```
public void process() {
    if (clientEmail == null && !clientEmail.contains("@")) {
        System.out.println("Email inválido. Falha no envio.");
    }

    String invoiceString = "--- NOTA FISCAL ---\n" +
          "Cliente: " + clientName + "\n" +
          "Valor: R$ " + amount + "\n" +
          "Tipo: " + (
              type == 1 ? "Simples"
              : type == 2 ? "Com imposto"
              : "Desconhecido"
          ) + "\n"
          + "--------------------";

    //imprimir nota
    System.out.println( invoiceString );

    // Enviar nota para email
    System.out.println("Enviando nota fiscal para: " + clientEmail);
    enviarPorEmail(clientEmail, invoiceString);
}
```

---

---

# 3 Refatorando para legibilidade

Você foi solicitado a revisar parte do código de precificação de uma plataforma de e-commerce. A função abaixo é responsável por calcular o preço final de um produto, considerando o tipo de cliente e se a compra

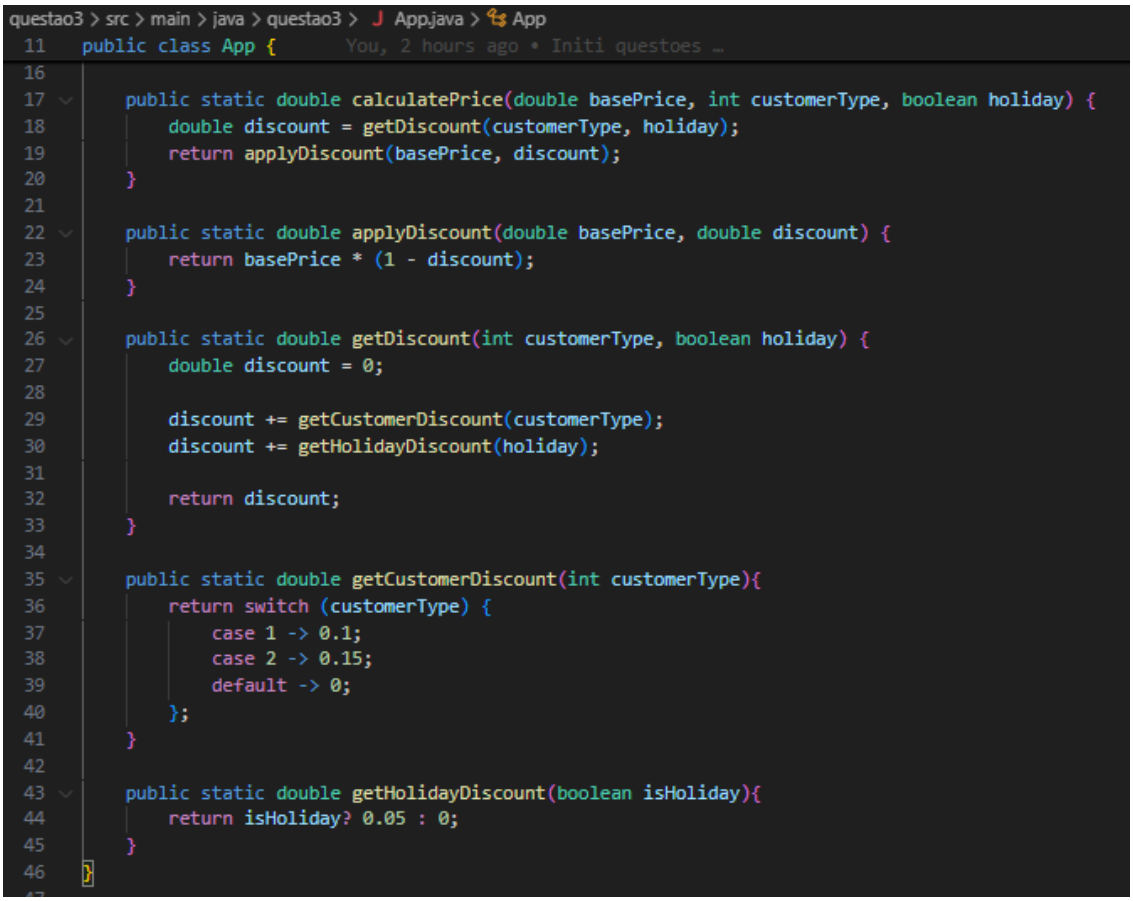
ocorre em um feriado. No entanto, a equipe relatou que a leitura do cálculo tem causado erros frequentes de interpretação e manutenção.

Dado o método abaixo:

```
public double calculatePrice(double basePrice, int customerType, boolean holiday) {
    double discount = 0;
    if (customerType == 1) discount = 0.1;
    else if (customerType == 2) discount = 0.15;
    if (holiday) discount += 0.05;
    return basePrice * (1 - discount);
}
```

Reescreva esse método com foco em **clareza**, **uso de variáveis explicativas** e **métodos auxiliares**, quando apropriado.

## Resposta




---

# 4 Refatorando para modularidade e encapsulamento

Você está revisando a classe User, parte de um sistema de gerenciamento de contas de usuários. Essa classe foi escrita de forma simplificada para acelerar entregas iniciais, mas hoje apresenta sérios problemas de acoplamento, falta de encapsulamento e dificuldade de extensão, especialmente no tratamento de múltiplos endereços associados a um mesmo usuário. Seu objetivo é refatorá-la para torná-la mais robusta, modular e segura para manutenção futura.

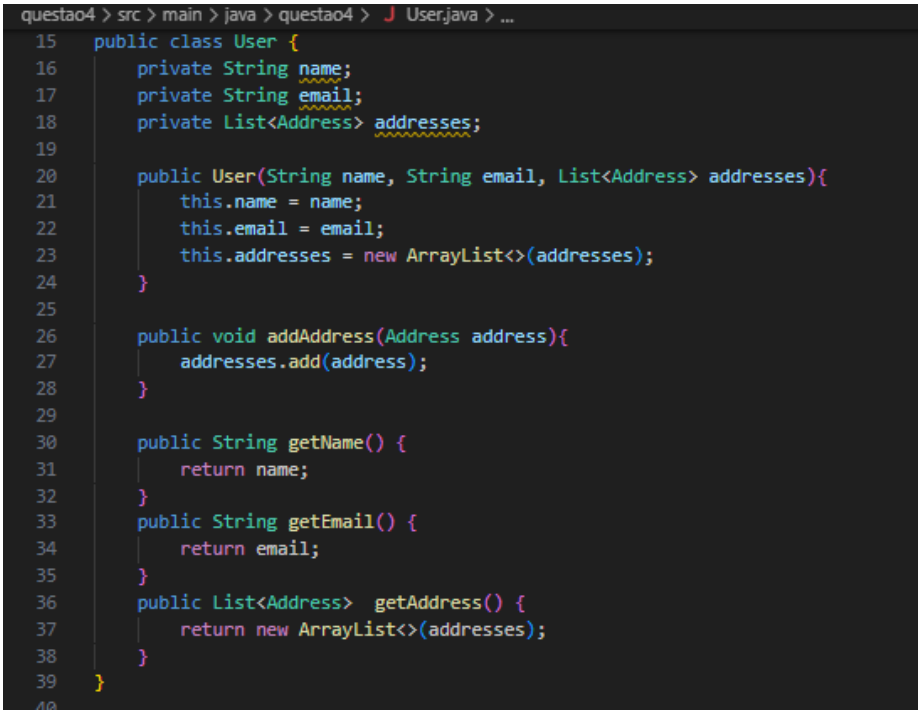
O trecho abaixo representa uma implementação frágil:

```
public class User {
    public String name;
    public String email;
    public List addresses;
}
```

Refatore esse código para:

- Aplicar encapsulamento de campos
- Substituir listas por objetos que representem endereços
- Introduzir um método que oculte o acesso direto à lista, como addAddress()

## Resposta




---

---

# 5 Refatorando condicional complexa com polimorfismo

Dado o seguinte código:

```
public class NotificationService {
    public void notifyUser(String channel, String message) {
        if (channel.equals("EMAIL")) {
            System.out.println("Sending EMAIL: " + message);
        } else if (channel.equals("SMS")) {
            System.out.println("Sending SMS: " + message);
        } else if (channel.equals("PUSH")) {
            System.out.println("Sending PUSH: " + message);
        }
    }
}
```

Você está trabalhando com o módulo de notificações de uma aplicação mobile. Esse módulo cresceu rapidamente e, para dar suporte a diferentes canais, a equipe implementou uma lógica condicional baseada em if-else. Esse modelo está dificultando a adição de novos canais de notificação e aumentando o risco de erros.

Refatore esse código aplicando **polimorfismo** para evitar a estrutura condicional. Use uma hierarquia de classes ou interfaces.

## Resposta

questao5 > src > main > java > questao5 > NotificationService.java > ...

```java
public class NotificationService {

    private List<INotificationChannel> availableChannels = Arrays.asList(
        new EmailChannel(),
        new SmsChannel(),
        new PushChannel()
    );

    public void notifyUser(String channel, String message) {

        for (INotificationChannel c : availableChannels) {
            if(c.getChannelName().equals(channel)){
                c.sendMessage(message);
                return;
            }
        }

        throw new IllegalArgumentException("The channel with the provided name is not available");
    }
}
```

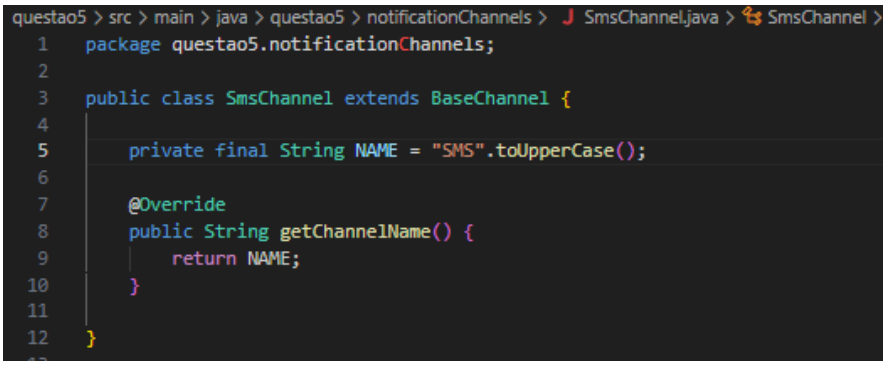
questao5 > src > main > java > questao5 > notificationChannels > INotificationChannel.java > ...

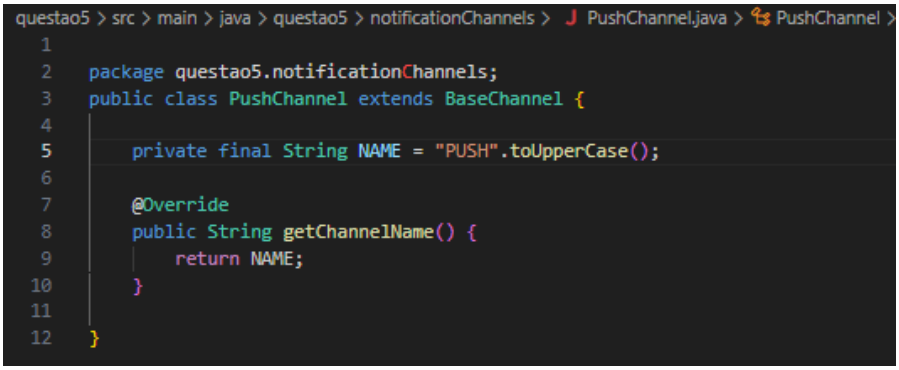
```java
package questao5.notificationChannels;

public interface INotificationChannel {

    public String getChannelName();
    public void sendMessage(String message);
}
```

questao5 > src > main > java > questao5 > notificationChannels > BaseChannel.java > ...

```java
package questao5.notificationChannels;

public abstract class BaseChannel implements INotificationChannel {

    @Override
    public abstract String getChannelName();

    @Override
    public void sendMessage(String message) {
        System.out.println("Sending " + getChannelName() + ": " + message);
    }


```

```java
package questao5.notificationChannels;

public class EmailChannel extends BaseChannel {

    private final String NAME = "EMAIL".toUpperCase();

    @Override
    public String getChannelName() {
        return NAME;
    }

}
```




---

# 6 Substituindo tipos por hierarquias e melhorando coesão

Dado o seguinte código:

```
public class Document {
    public String type;

    public void print() {
        if (type.equals("PDF")) {
            System.out.println("Printing PDF");
        } else if (type.equals("HTML")) {
            System.out.println("Printing HTML");
        } else {
            System.out.println("Unknown format");
        }
    }
}
```

Você recebeu um código legado que representa documentos por meio de uma string de tipo, dificultando a extensão e o reaproveitamento de comportamentos específicos para cada tipo. A equipe deseja transformar esse modelo frágil em uma hierarquia mais expressiva e sustentável.

1. Identifique os problemas causados pela abordagem atual baseada em códigos de tipo.
2. Implemente a refatoração completa substituindo o campo type por uma hierarquia com subclasses PdfDocument, HtmlDocument, etc., aplicando polimorfismo.
3. Implemente uma classe principal com um método main que instancie três documentos diferentes utilizando as novas subclasses (PdfDocument, HtmlDocument, etc.) e invoque o método print() para cada um. O código deve demonstrar o uso correto do polimorfismo.

Mostre o resultado esperado da execução e explique por que essa abordagem melhora a coesão, reduz duplicação e facilita a adição de novos formatos de documento.

# 1 Problemas no código

A implementação mostrada apresenta problemas de **manutenção**, **legibilidade**, **resiliencia a erros** e **divisão de responsábilidades**. Considerando cada um deles individualmente:

### Manutenção

A manutenção é comprometida na implementação mostrada pois para cada novo tipo de Documento é nescessário alterar a sequencia de condicionais, podendo afetar também os demais tipos.

### Legibilidade

A legibilidade do código é comprometida devido a concatenação de *ifs*, que não transmitem claramente a operação que é proposta a ser feita. Para exemplificar, consideremos que o procedimento deste bloco de código (a sequencia de *IFs*) é um ***mapeamento*** entre um comportamento (qualquer coisa dentro do *IF* ) e um valor predefinido.

Para que um leitor compreenda este procedimento, é nescessário que ele confirme que cada condição está construida no modelo `if( type.equals("") )` para poder garantir que o procedimento é, de fato, um ***mapeamento*** entre um comportamento e um valor predefinido.

Isso porque existe a possibilidade de haver uma condicional construida de maneira diferente ( como `if ( len(type) == 2 )` ) que faz com que o procedimento deste bloco não seja um mapeamento.

Em contrapartida, se um switch fosse utilizado, poderiamos garantir que o procedimento daquele bloco de código é um ***mapeamento*** entre um valor predefinido e um comportamento apartir do momento que vissemos a primeira linha ( `switch (type)` )

### Resiliencia a erros

O codigo mostrado possui dois pontos especialmente sucetiveis a erros:

- A variavel `type` é tipada como uma string, o que não garante a outros desenvolvedores valores fixos com os quais se pode comparar. ou seja, um erro de grafia ou capitalização (não apontado pelo compilador ou pela IDE) e o codigo pode não funcionar.
- A concatenação de IF-Elses é propensa a erros já que é extremamente sensível a erros já que uma alteração gera efeitos em cascata ao longo das demais condicionais. Exemplo de um erro que pode acontecer:

```
if (type.type.equals("PDF")) {
    System.out.println("Printing PDF");
} if (type.equals("HTML")) { // Omição do 'else'. erro não apontado pelo compilador.
    System.out.println("Printing HTML");
} else {
    System.out.println("Unknown format");
}
```

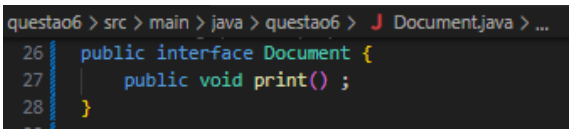
### Separação de responsábilidades

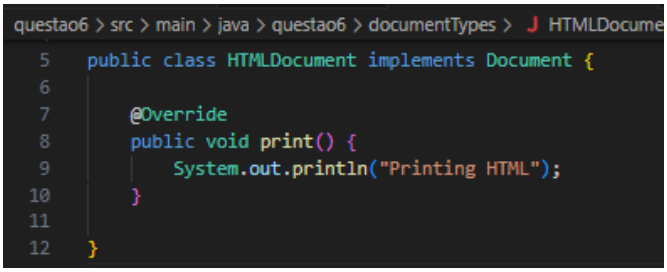
No codigo mostrado, a classe document é a responsavel pela implementação da funcionalidade de impreção de cada um dos diferentes tipos de documento. além disso, o metodo `print` é o responsavel tanto por escolher qual implementação será usada (com base na propriedade `type` ) quanto por implementar as funcionalidades.

# 2 Refatoração

**Document.java**

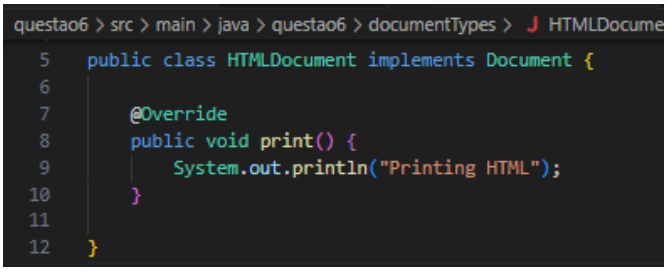
Transformado em interface



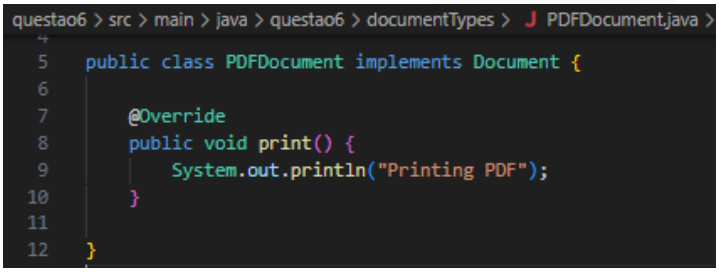

**HTMLDocument.java**

Implementa a interface para o caso `type.equals("HTML")`

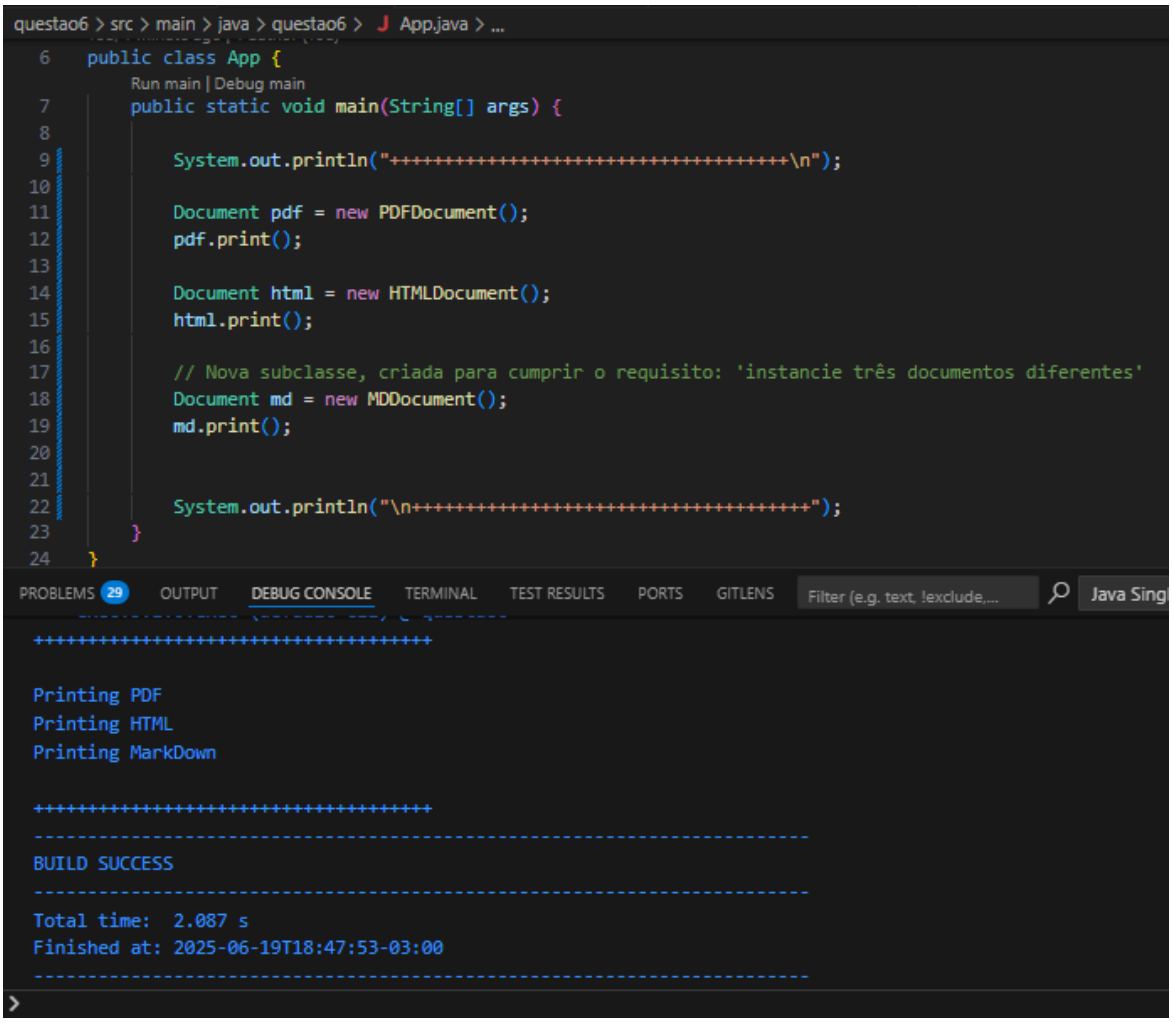



**PDFDocument.java**

Implementa a interface para o caso `type.equals("PDF")`




## 3.1 Execução


questao6 > src > main > java > questao6 > App.java > ...
public class App {
Run main | Debug main
public static void main(String[] args) {
System.out.println("++++++++++++++++++++++++++++++++++++\n");
Document pdf = new PDFDocument();
pdf.print();
Document html = new HTMLDocument();
html.print();
// Nova subclasse, criada para cumprir o requisito: 'instancie três documentos diferentes'
Document md = new MDDocument();
md.print();
System.out.println("\n++++++++++++++++++++++++++++++++++++");
}
}
PROBLEMS 29
OUTPUT
DEBUG CONSOLE
TERMINAL
TEST RESULTS
PORTS
GITLENS
Filter (e.g. text, !exclude,...
Java Sing
++++++++++++++++++++++++++++++++++++
Printing PDF
Printing HTML
Printing MarkDown
++++++++++++++++++++++++++++++++++++
BUILD SUCCESS
Total time: 2.087 s
Finished at: 2025-06-19T18:47:53-03:00