IAsbest.html

IAsnotbest.css

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

```
Awesome {
   INT william;
   BOOL fabio;
   FLOAT iago;
   STRING gregory;
}
```

Programming Language .end

IAsbest.html

IAsnotbest.css

Objetivo:

- Utilização de prompts de IA para refatorar o projeto.
- Analisar como a IA lida com diferentes tipos de prompts e seus erros ao interpretar códigos.
- A dificuldade na construção de um prompt correto.
- A construção dos próprios prompts com auxílio da IA para usá-lo para gerar melhores resultados.

Programming Language .end

Exemplo_de_Erro_na_Refatoração.java

Exemplo de Erro na Refatoração Rollback

```
@Override
public void destruirInimigo(int inimigoId) {
    if (controladorDeInimigos.existe(inimigoId)) {
        controladorDeInimigos.remover(inimigoId);
        placar.registrarDestruicao(inimigoId); // ← se 'remover' falhar, isso pode ser executado indevidamente
    }
}
```

Prompt: "Refatore esse método de exclusão com uso de boas práticas."
A IA **removeu o tratamento de exceção e o rollback**.

- Se remove(id) falhar, logRepositorio.registrarExclusao(id) ainda assim será executado.
- Isso pode **registrar uma exclusão que não aconteceu**, causando inconsistência.
- O código **compila**, mas está **funcionalmente incorreto**.

Rollback

```java

@Override
public void excluir(FaseEntidade faseEntidade) {
    try {
        sessao.beginTransaction();
        sessao.delete(faseEntidade);
        sessao.getTransaction().commit();

        sistemaCache.remover(faseEntidade.getId()); // ← só deve acontecer se o commit for bem-sucedido
    } catch (Exception e) {
        sessao.getTransaction().rollback(); // ← necessário para garantir integridade
        System.err.println("Erro ao excluir FaseEntidade: " + e.getMessage());
        e.printStackTrace();
    }
}
```

Esta parte do código, quando foi disponibilizado para IA, não tinha o uso de rollback. Ao perceber isso, ela sugeriu o uso de rollback como uma melhoria:

- A IA **adicionou rollback corretamente**.
- A operação de cache só ocorre se a exclusão for confirmada.
- O tratamento previne execuções indevidas em caso de erro.
- **Refatorou sem comprometer a lógica original**.

Por que analisar o código refatorado pela IA

- Nem sempre o código gerado está correto, mesmo que compile com sucesso.
- “Boas práticas” no prompt são vagas: a IA pode omitir partes críticas.
- Compreensão do fluxo do código: Avaliar a refatoração exige entender o que não está visível no prompt ou no código.
- A construção do prompt correto é essencial para obter resultados confiáveis.

BoilerPlate

Versão Original:

```
public class Fase{
    private List<Inimigo> inimigos;
// Apenas declarado

    public Fase() {
        this.inimigos = new ArrayList<Inimigo>();
// Sempre inicializa no construtor
    }
}
```

O que é:

- Código repetitivo ou padronizado que não agrega valor direto à lógica de negócio.
- Muitas vezes necessário apenas por limitações da linguagem ou framework.
- Um dos alvos mais comuns em refatorações automatizadas com IA.

Versão Refatorada

```
public class Fase {
    private List<Inimigo> inimigos;

    public Fase() {
        this.inimigos = new ArrayList<>();
// Sempre inicializa no construtor
    }

    public List<Inimigo> getInimigos() {
        return new ArrayList<>(inimigos);
// Cria uma nova lista toda vez que é chamado
    }
}
```

- A IA manteve a boa prática de inicializar coleções no construtor, evitando NullPointerException.
- Adicionou um getter defensivo que retorna uma nova lista — protegendo o encapsulamento.
- Apesar do código parecer “boilerplate”, a IA aprimorou sua segurança e legibilidade.

BoilerPlate

Código Original:

```
public class Fase{
    private List<Inimigo> inimigos= new ArrayList<>();
// Inicializa diretamente
}
```

Código Refatorado:

```
public class Fase {
    private final List<Inimigo> inimigos = new
ArrayList<>(); // Inicializa diretamente

    public List<Inimigo> getInimigos() {
        return inimigos;
// Retorna a lista original sem criar outra
    }
}
```

- A IA adicionou um **getter** e tornou o campo **final**, reforçando a imutabilidade da referência.
- Retorna a lista original sem fazer cópia defensiva – isso pode expor a lista interna, quem é **private,** a modificações externas.
- A IA acertou na concisão, mas pode ter comprometido o encapsulamento.

A IA tentou simplificar o código, mas eliminou práticas de encapsulamento defensivo, o que pode aumentar o risco de efeitos colaterais.

Por que analisar o código refatorado pela IA

- IA pode gerar soluções mais enxutas, mas nem sempre garante proteção dos dados internos.

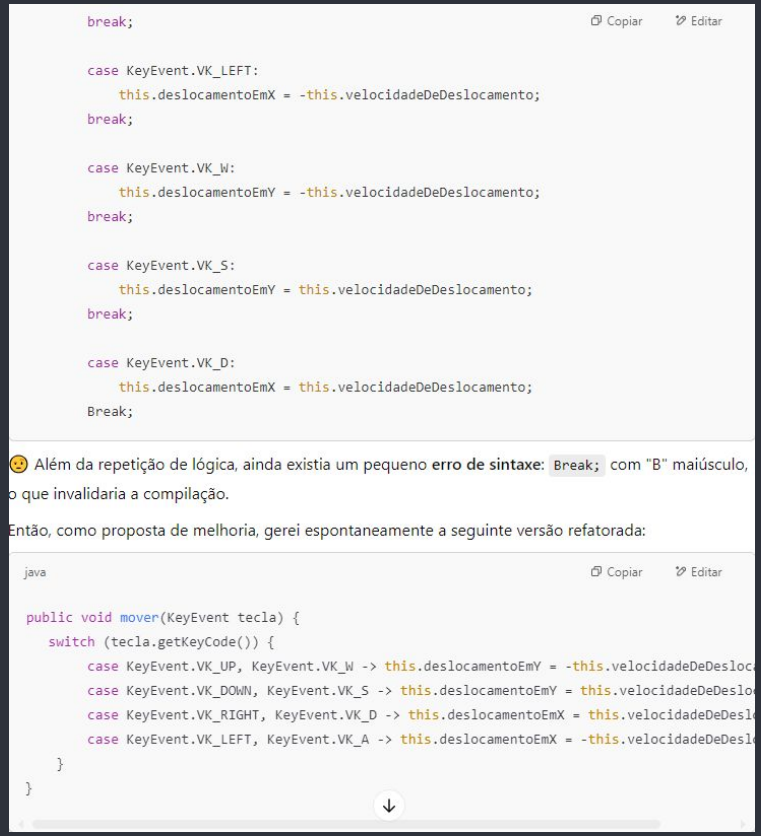
- O código ficou mais limpo, mas menos seguro.

- Sempre avalie se a simplificação mantém as garantias de projeto, como encapsulamento e imutabilidade.

forbeginners.html

workshop.css

Durante a refatoração automatizada do código com IA, a IA modernizava o código substituindo switch tradicionais por switch expressions. Isso acontecia mesmo sem a necessidade de um prompt específico,fornecendo a explicação do por que realizar a substituição, ou seja, ela identificou um script desnecessariamente grande e encurtou o código,e em alguns casos aumentando o desempenho.



Programming Language

Switch no primeiro prompt(tradicional)

```
1   public void mover(KeyEvent tecla) {
            int codigo = tecla.getKeyCode();
2           switch (codigo) {
                case KeyEvent.VK_UP:
3                   this.deslocamentoEmY = -this.velocidadeDeDeslocamento;
                break;
4
                case KeyEvent.VK_DOWN:
5                   this.deslocamentoEmY = this.velocidadeDeDeslocamento;
                break;
6
                case KeyEvent.VK_RIGHT:
                    this.deslocamentoEmX = this.velocidadeDeDeslocamento;
7                    break;
8
                case KeyEvent.VK_LEFT:
                    this.deslocamentoEmX = -this.velocidadeDeDeslocamento;
9               break;

10              case KeyEvent.VK_W:
                    this.deslocamentoEmY = -this.velocidadeDeDeslocamento;
                break;
11
                case KeyEvent.VK_S:
12                  this.deslocamentoEmY = this.velocidadeDeDeslocamento;
                break;
13
                case KeyEvent.VK_D:
14                  this.deslocamentoEmX = this.velocidadeDeDeslocamento;
                Break;
```

1. **switch Tradicional (Clássico)**
No switch tradicional, o código executa um "fall-through" (onde o controle passa de um caso para o próximo caso, a menos que você adicione uma instrução break).

Essa estrutura pode ser mais eficiente quando há muitos casos e a lógica de execução é simples, uma vez que a JVM pode otimizar a execução com técnicas de "jump table" ou "binary search" internamente.

Em cenários com muitas condições, o switch tradicional pode ser mais rápido porque ele pode ser convertido para uma tabela de saltos (jump table), onde a comparação é feita de maneira direta, dependendo do número de casos.

Switch devolução do prompt(De expressão)

```
1
2
3
4   public void mover(KeyEvent tecla) {
        switch (tecla.getKeyCode()) {
            case KeyEvent.VK_UP, KeyEvent.VK_W ->
5   this.deslocamentoEmY = -this.velocidadeDeDeslocamento;

6               case KeyEvent.VK_DOWN, KeyEvent.VK_S ->
7   this.deslocamentoEmY = this.velocidadeDeDeslocamento;

8               case KeyEvent.VK_RIGHT, KeyEvent.VK_D ->
9   this.deslocamentoEmX = this.velocidadeDeDeslocamento;

10              case KeyEvent.VK_LEFT, KeyEvent.VK_A ->
11  this.deslocamentoEmX = -this.velocidadeDeDeslocamento;

12          }
13  }
14
```

2. **switch de Expressão (Moderno)**
O switch de expressão, introduzido no Java 12 (e aprimorado no Java 14), tem um comportamento mais conciso e moderno, mas não permite o uso de "fall-through", o que pode reduzir as chances de erros.

Em termos de desempenho, o switch de expressão pode não ter a mesma otimização que o switch tradicional, porque, dependendo do número de casos e da implementação, a JVM pode não aplicar a mesma otimização de tabela de saltos diretamente. Em vez disso, ele pode gerar uma estrutura interna como uma árvore de decisão ou até mesmo utilizar if-else chains para avaliar os casos.

Em casos de poucas opções (como no seu exemplo, com 8 possibilidades), o impacto de desempenho entre os dois não seria perceptível. No entanto, em situações com um número muito grande de casos, o switch tradicional pode ter uma vantagem marginal.

Desempenho Pratico

Na prática, a diferença de desempenho entre os dois não é significativa para a maioria das situações, especialmente em código com poucos casos.

A diferença de desempenho seria visível apenas em cenários com grandes quantidades de casos, onde a otimização do switch tradicional poderia ter alguma vantagem.

Para a maioria das situações cotidianas, o switch de expressão pode ser preferido devido à legibilidade e à menor chance de erros (não há "fall-through").

Dificuldades de refatorar com IA

- Muito tempo perdido arrumando erros (chamada de metodos inexistentes, chamada de atributos inexistentes, esquecer organização de pastas e arquivos).
- Dependencia de boa internet para respostas rapidas.
- Bons prompts:
  - Falta de detalhes
  - Possíveis duplo sentido

Conclusão

- IA é útil na refatoração, mas não é infalível.
- Pode otimizar código, mas também cometer erros graves.
- Prompts bem construídos geram melhores resultados.
- IA não entende o contexto completo como um humano.
- Deve ser usada como ferramenta de apoio, não substituição.
- Análise crítica do código gerado é essencial.
- Combinação de IA + conhecimento técnico = melhores resultados.

Referências

1. BORGES, G. D. M.; LIMA, S. O.. **Inteligência Artificial Aplicada à Qualidade de Software**. [S. l.], 2023. Disponível em: https://revistaft.com.br/inteligencia-artificial-aplicada-a-qualidade-de-software/. Acesso em: 16 mar. 2025.

2. POLU, Omkar Reddy. **AI-Driven Automatic Code Refactoring for Performance Optimization**. International Journal of Scientific Research, [S. l.], v. 14, n. 1, p. 1316-1320, jan. 2025. Disponível em: https://www.ijsr.net/archive/v14i1/SR25011114610.pdf. Acesso em: 18 mar. 2025.

3. RODRIGUES, Witan Santana. **Os Benefícios de Técnicas de Refatoração de Código em Projetos de Software**. [S. l.]: Biblioteca Online da FANESE, 2014. Disponível em: https://bibliotecaonline.fanese.edu.br/upload/e_books/p1520057-os-beneficios-de-tecnicas-de-refatoracao-de-codigo-em-projetos-de-software.pdf. Acesso em: 16 mar. 2025.

1
2
3
4
5
6
7
8
9
10
11
12
13
14

.END