Tp2

Alunos:

- 190084600 - Arthur José Nascimento de Lima
- 190084731 - Augusto Durães Camargo
- 200056981 - Arthur Ferreira Rodrigue
- 190027355 - Erick Melo Vidal de Oliveira

Code Smells do Projeto e Princípios de Bons Projetos

Simplicidade

Definição: Um código simples é direto, sem elementos ou complexidades desnecessárias. A simplicidade facilita a compreensão e manutenção do código, reduzindo a probabilidade de erros.

Fowler destaca, "qualquer tolo pode escrever código que um computador entenda; bons programadores escrevem código que os humanos possam entender." A simplicidade evita a confusão e reduz a probabilidade de erros, criando um sistema mais coeso.

Segundo Fowler, "Faça a coisa mais simples que poderia possivelmente trabalhar" e "Você não vai precisar" (conhecido como YAGNI), Isso significa que não devemos criar recursos que não são úteis Hoje, Então qualquer código que não está sendo utilizado, fere o princípio da simplicidade.

A segunda razão para um design simples é que um design complexo é mais difícil de entender do que um design simples. Portanto, qualquer modificação do sistema é dificultada pelo aumento da complexidade.

Code Smell no Projeto:

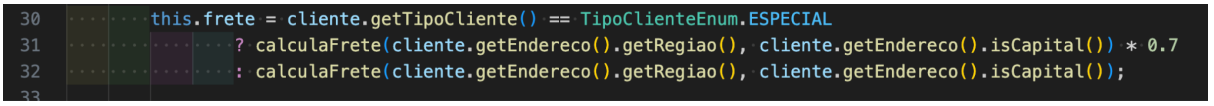
Complexidade Acidental (Complexity): Um código excessivamente complicado, cheio de detalhes desnecessários, torna-se difícil de entender e manter. Refatorações como "Simplify Conditionals" ajudam a reduzir a complexidade, promovendo simplicidade. Código Repetido (Duplicated Code): A presença de código duplicado é um sinal de que a solução pode ser simplificada. Refatorar o código para eliminar duplicações melhora a simplicidade.

No Construtor da classe VendaModel, podemos perceber que há uma quebra dessa regra, criando um design mais complexo para lidar com simples inicialização de atributos da

classe. Como também, as várias condições e cálculos realizados no construtor (`if-else`, cálculos de frete, imposto, desconto cashback) tornam o código difícil de entender e manter.

```
21  public VendaModel(ClienteModel cliente, LocalDateTime dateTime, List<ProdutoModel> produtos) {
22      this.cliente = cliente;
23      this.dateTime = dateTime;
24      this.produtos = produtos;
25
26      this.desconto = calculaDesconto(cliente);
27
28      this.valorTotal = this.calculaValorProdutos() * (1.0 - this.desconto);
29
30      this.frete = cliente.getTipoCliente() == TipoClienteEnum.ESPECIAL
31              ? calculaFrete(cliente.getEndereco().getRegiao(), cliente.getEndereco().isCapital()) * 0.7
32              : calculaFrete(cliente.getEndereco().getRegiao(), cliente.getEndereco().isCapital());
33
34      this.imposto = new ImpostoModel(this.valorTotal + this.frete, this.cliente.getEndereco());
35      this.valorTotal += this.frete + this.imposto.getIcms() + this.imposto.getMunicipal();
36
37      if(this.valorTotal <= this.cliente.getSaldoCashback()){
38          this.descontoCashback = this.valorTotal;
39          this.valorTotal = 0.0;
40      }
41      else{
42          this.descontoCashback = this.cliente.getSaldoCashback();
43          this.valorTotal -= this.cliente.getSaldoCashback();
44      }       ErickMVdO, last month • corrigindo calculo cashback
45      this.cliente.zeraSaldoCashback();
46      this.saldoCashback = Cashback.calcula(this.cliente.getTipoCliente(), valorTotal, cliente.getCartao().isEmpresar
47      cliente.addSaldoCashback(saldoCashback);
48
49      DatabaseModel.getVendas().add(this);
50  }
```

Na linha 30, por exemplo, temos um aumento de complexidade do código ferindo o princípio da simplicidade, onde poderíamos tornar um design mais simples.



```
30          this.frete = cliente.getTipoCliente() == TipoClienteEnum.ESPECIAL
31                  ? calculaFrete(cliente.getEndereco().getRegiao(), cliente.getEndereco().isCapital()) * 0.7
32                  : calculaFrete(cliente.getEndereco().getRegiao(), cliente.getEndereco().isCapital());
33
```

Podemos resolver esse problema com uma refatoração de extrair o método chamando um método getFretePorTipoDeCliente(), passando como parâmetro o cliente. Assim tornamos o código um pouco mais simples para ser compreendido e mais fácil para a manutenção.

Elegância

Definição: Um código elegante é aquele que, além de resolver o problema de maneira eficaz, o faz de uma maneira que é esteticamente agradável, clara e eficiente.

A elegância no código não é apenas uma questão de estilo, mas de efetividade. Um design elegante resolve problemas de maneira direta e clara, com o mínimo de elementos necessários. A elegância muitas vezes emerge da simplicidade, mas também envolve a escolha criteriosa de abstrações que tornam o código mais legível e intuitivo. Como Fowler sugere, o código elegante deve ser um prazer de ler, uma solução que parece "natural".

Code Smell no Projeto:

Código Mal Organizado (Disorganized Code): Código desorganizado ou confuso carece de elegância. Refatorar para melhorar a organização e clareza promove a elegância.

Nomeação Pobre (Poor Naming): Nomes inadequados ou confusos para variáveis, funções ou classes comprometem a elegância do código. Renomear elementos para refletir seu propósito real é uma forma de melhorar a elegância.

Um exemplo, que podemos ver no projeto é que o código abaixo não segue um padrão de idioma, Por exemplo, Na classe CartaoModel, podemos ver que alguns métodos não seguem o padrão do projeto em Português, criando misturas entre portugues e ingles para descrever o código, tornando um pouco deselegante o código.

```
12      private static boolean checkIsEmpresarial(String numero) {
13          if (numero.startsWith(prefix:"4296 13")) {
14              return true;
15          }
16          return false;      ArthurFerreiraRodrigues, last month • #13 – Add Mod
17      }
18
19      public boolean isEmpresarial() {
20          return isEmpresarial;
21      }
22
23      private String encryptNumero() {
24          return numero.substring(beginIndex:0, endIndex:7) + "** **** ****";
25      }
```

No código abaixo podemos também visualizar no método isNumeroValido, o mesmo problema do código acima e também uma falta de preocupacao em ler o código de maneira simples, por exemplo cada string do matches, poderiam ser variáveis descrevendo exatamente o que gostariamos.

```
27      /*
28       * Método que verifica se o número do cartão é válido
29       * e se está no formato correto
30       *
31       * O número do cartão deve ter 19 caracteres e estar no formato:
32       * "XXXX XXXX XXXX XXXX"
33       *
34       * A validação deve ser realizada antes de contruir o objeto
35       */
36      public boolean isNumeroValido() {
37          if (numero.length() == 19
38                  && numero.matches("[0-9]{4}" + " " + "[0-9]{4}" + " " + "[0-9]{4}" + " " + "[0-9]{4}")) {
39              return true;
40          }
41
42          return false;
43      }
44
45      public String getNumero() {
46          return encryptNumero();
47      }
48
49  }
```

Para A primeira imagem podemos simplesmente renomear os metodos para portugues, e manter o padrao geral do projeto

Para a segunda imagem podemos descrever uma string, por exemplo:

padraoQuatroNumeros = “[0-9]{4}”

Assim, além de tornar o código mais legível para aqueles que não conhecem, regex torna o código um pouco mais legível e elegante.

Modularidade

Definição: A modularidade é a divisão do sistema em módulos distintos, cada um com uma responsabilidade específica. Isso facilita a manutenção e permite que diferentes partes do sistema evoluam de forma independente.

Code Smell no Projeto:

Classes Muito Grandes (Large Class): Classes que fazem muitas coisas ao mesmo tempo são um sinal de baixa modularidade. Refatorações como "Extract Class" ajudam a dividir essas classes em módulos menores e mais coesos.
Métodos Longos (Long Method): Métodos longos e complexos devem ser divididos em métodos menores e mais modulares, facilitando a compreensão e reutilização do código.

Boas Interfaces

Definição: Boas interfaces definem contratos claros para a interação entre diferentes partes do sistema, permitindo que módulos e componentes se comuniquem de maneira consistente e previsível.

Code Smell no Projeto:

Interface Excessivamente Complexa (Excessive Interface Complexity): Interfaces que são complicadas demais para serem entendidas ou usadas corretamente. Simplificar e esclarecer as interfaces torna o código mais acessível e fácil de manter.
Inconsistência na Interface (Inconsistent Naming/Usage): A falta de consistência em como as interfaces são definidas e usadas pode causar confusão. Refatorar para garantir consistência ajuda a manter interfaces claras e robustas.

Extensibilidade

Definição: Um código extensível é projetado de forma que novas funcionalidades possam ser adicionadas com o mínimo de impacto nas partes existentes do sistema.

Code Smell no Projeto:

Código Rígido (Rigidity): Código que é difícil de modificar ou estender sem causar problemas em outras partes do sistema. Aplicar o princípio "Open/Closed" pode ajudar a promover a extensibilidade.
Dependências Enredadas (Tangled Dependencies): Quando diferentes partes do código são fortemente acopladas, torna-se difícil fazer mudanças ou adições. Refatorar para reduzir o acoplamento e aumentar a coesão promove a extensibilidade.

Evitar duplicação

Definição: Duplicação de código é quando trechos de código semelhantes ou idênticos aparecem em vários lugares. Evitar duplicação ajuda a manter o código mais fácil de manter e menos propenso a erros.

Code Smell no Projeto:

Código Repetido (Duplicated Code): Quando o mesmo código aparece em múltiplos lugares, torna-se mais difícil de manter e propenso a erros. Refatorações como "Extract Method" ou "Extract Class" são formas de eliminar duplicação.
Viés de Copiar e Colar (Copy-Paste Programming): Reutilizar código por meio de copiar e colar em vez de abstrair a lógica repetida em um único local aumenta a complexidade e o risco de inconsistências. Refatorar para remover essa duplicação melhora a qualidade do código.

Portabilidade

Definição: Portabilidade refere-se à capacidade do código de ser executado em diferentes ambientes ou plataformas com pouca ou nenhuma modificação.

Code Smell no Projeto:

Dependência de Plataforma (Platform Dependency): Código que depende fortemente de características específicas de uma plataforma limita sua portabilidade. Refatorar para abstrair dependências específicas pode aumentar a portabilidade do código.
Configuração de Caminho Absoluto (Hard-Coded Paths): Caminhos ou valores específicos de uma plataforma que estão codificados no sistema comprometem a portabilidade. Refatorar para usar caminhos relativos ou configuráveis melhora a portabilidade.

Código deve ser idiomático e bem documentado

Definição: Código idiomático segue as convenções e melhores práticas da linguagem de programação em uso. Documentação clara e concisa é crucial para garantir que outros desenvolvedores possam entender e manter o código.

Code Smell no Projeto:

Código Esotérico (Obscure Code): Código que é difícil de entender devido à sua complexidade ou falta de clareza. Refatorar para seguir práticas idiomáticas e adicionar documentação pode ajudar a tornar o código mais acessível.
Comentário Supérfluo (Redundant Comment): Comentários que explicam o óbvio ou que se tornam obsoletos podem ser um sinal de que o código precisa ser simplificado ou melhor nomeado. Comentários devem ser informativos e necessários, não substituindo a clareza do código em si.

Referências

- DTI Digital. Manifesto Ágil: Pilares Básicos. DTI Digital, 10 jul. 2020. Disponível em: <https://www.dtidigital.com.br/blog/manifesto-agil-pilares-basicos>. Acesso em: 12 ago. 2024.
- FOWLER, Martin. Is Design Dead?. Martin Fowler, 04 mar. 2001. Disponível em: <https://www.martinfowler.com/articles/designDead.html#TheValueOfSimplicity>. Acesso em: 12 ago. 2024.
- FOWLER, Martin. Beck Design Rules. Martin Fowler, 15 jan. 2003. Disponível em: <https://www.martinfowler.com/bliki/BeckDesignRules.html >. Acesso em: 12 ago. 2024.
- LUZKAN. Code Smells: Definitions and Examples. Luzkan, 05 nov. 2019. Disponível em: <https://luzkan.github.io/smells/>. Acesso em: 12 ago. 2024.