Universidade de Brasília
Faculdade UnB Gama

Técnicas de Programação para Plataformas Emergentes

Trabalho prático 3

Guilherme Leal	15/0128312
João Pedro Soares	15/0132344
Lucas Alexandre	15/0136862
Matheus de Cristo	15/0141220
Moacir Mascarenha	17/0080366

Para esse trabalho o grupo deverá escolher 5 características dentre as 9 características de um bom projeto de software apresentadas acima e, para cada uma delas, apresentar:

uma descrição da característica, mostrando claramente quais são os seus efeitos no código (em termo de estrutura, claridade, coesão, acoplamento dentre outros efeitos aplicáveis);
Uma relação da característica com os maus-cheiros de código definidos por Fowler. Uma descrição dos maus cheiros está disponível nos slides sobre o conteúdo de refatoração;
pelo menos uma operação de refatoração capaz de levar o projeto de código a ter a característica em análise.

Características selecionadas:

- **Simplicidade**
- **Modularidade (baixo acoplamento e alta coesão)**
- **Extensibilidade**
- **Ausência de duplicidades**
- **Boa documentação.**

Modularidade

A modularidade é um fator essencial no desenvolvimento de software, pois com a decomposição do código em módulos menores, garantimos que ocorra diminuição da complexidade em cada módulo, facilitando *debbuging*, testes causando aumento da reutilização e legibilidade. Outras duas qualidades essesnciais que a modularização sao: Alta Coesão e Baixo Acoplamento.

Alta coesão: Onde cada módulo possui apenas funcionalidades que são de sua responsabilidade e mensura o quão bem os modulos sao executados como um todo.

Baixo Acoplamento: O Acoplamento de codigo é uma medida capaz de dizer o quanto os módulos estão interligados entre si, baixo acoplamento significa que os modulos foram bem decompostos e as funciolidades bem divididas.

Exemplo de refatoração:

```
class PlayerDatabase
{
        public void connectDatabase();
        public void printAllPlayersInfo();
        public void printSinglePlayerInfo();
        public void printRankings();
        public void printEvents();
        public void closeDatabase();
}
```

A clase PlayerDatabase possui muitas tarefas diferentes que podem comprometer a manutenabilidade e futuras atualizações.

Refatoração Possivel: Decompor a classe PlayerDatabase em módulos menores que são responsáveis por executar tarefas especificas.

```
class PlayerDatabase
{
        ConnectDatabase connectD= new connectDatabase();
        PrintAllPlayersInfo allPlayer= new PrintAllPlayersInfo();
        PrintRankings rankings = new PrintRankings();
        CloseDatabase closeD= new CloseDatabase();
        PrintSinglePlayerInfo singlePlayer = PrintSinglePlayerInfo();
}

class ConnectDatabase {}

class CloseDatabase {}

class PrintRankings {}

class PrintAllPlayersInfo {}

class PrintSinglePlayerInfo {}
```

Extensibilidade

Um projeto de software bem realizado, permite que funcionalidades novas possam ser inseridas ao código quando há necessidade. Isso significa que é possível adicionar modificações ao código existente, sendo assim o código pode ser estendido.

As extensões podem ser acomodadas em andaimes de software, como plugins que podem ser carregados , hierarquia entre classes, funções de retorno e até mesmo estrutura.

A extensibilidade de acordo com, Stringfixer, é um princípio da engenharia de software e design de sistemas onde prevê o crescimento futuramente, sendo a medida da capacidade de estender um sistema e do nível de esforço necessário para sua implementação.

Projetos extensíveis, devem conter uma estrutura leve capaz de permitir alterações, seguindo princípios de separações dos elementos em unidades de trabalho compreensiveis.Com isso é pode-se evitar problemas como a baixa coesão e alto grau de acoplamento. A aplicação adequada destes princípios podem diminuir os maus cheiros como a duplicação de código, método longo, classe grande, mudanças divergentes, cirurgia com rifle, etc.

Uma **operação de refatoração que pode ser aplicada**, no contexto do Trabalho 1(Refatorado) realizado pelo grupo, é a utilização do padrão de projeto factory para a criação de deduções, possibilitando futuramente adicionar novos tipos de deduções ao projeto sem grandes mudanças na arquitetura.

Passos para a operação:

1- Padronizar estrutura das classes Dedução por meio de classe abstrata ou interface:

```
interface Deducao {
    public void setDeducao();
    public Float getValor();
    public String getDescricao();
}
```

2- As deduções devem implementar os métodos definidos pela interface ou classe abstrata:

```
class DeducaoDependente implements Deducao {
    private final String descricao = "Dependente";
    private final Float valor = 189.59f;
    private String nomeDependente;
    private String dataNascimentoDependente = "";

  public void setDeducao(String nomeDependente, String dataNascimento) {
        this.nomeDependente = nomeDependente;
        this.dataNascimento = dataNascimento;
   }
  public Float getValor(){
    return this.valor;
  }
  public String getDescricao(){
    String info = "Descricao: " + descricao + "\n" + "Nome: " +
        nomeDependente + "\n" + "Data Nascimento: "
        + dataNascimentoDependente + "\n" + "Valor: " +
        Float.toString(valor) + "\n";

    return info;
  }
}
```

3 - Implementar factory para a criação de deduções:

```
class FabricaDeducao {
  public Deducao create() {
    return new DeducaoDependente();
  }
}
```

Simplicidade

Simplicidade é descrito como uma estrutura que deve ser óbvia para não confundir o fluxo normal de execução de um código, apresentando uma ordenação consistente. Além disso, deve-se evitar criar muitos nós, os quais levam a códigos complexos e que necessitam de explicações.
Algumas práticas devem ser executadas, como é de conhecimento que cada função deve executar uma tarefa com um resultado, escolha de nomes significativos para variáveis, arquivos e funções. Essas práticas evitam comentários desnecessários ou análise do que está no código.
Outras características como: decomposição de funções atômicas, tipos descritivos de constantes, ênfase em código importante para o funcionamento, fornecimento de cabeçalho de arquivo, tratamento de erros, e comentários significativos.

Há no livro uma “key concept” a qual descreve o seguinte:“*Simplicity is a virtue. Never make code more complex than necessary*”. De forma geral, o intuito é tornar o objetivo do software simples e produtivo, removendo então as camadas de abstrações desnecessárias e a complexidade do sistema. A simplicidade e a suficiência das atividades, tendem a garantir uma boa produtividade e qualidade no desenvolvimento.

Uma relação direta com os maus cheiros, é a necessidade de refatoração onde deve ocorrer divisão de códigos em trechos menores ou substituição de código para algo de melhor qualidade.

Basicamente segundo Fowler: “I*mplement simplification to actually modify the code to remove the code smell*”. Um exemplo do Trabalho prático pode ser visto na extração de constante, a qual remove a utilização de variável temporária para cálculo de expressão, modificação de variável temporária. Consequentemente a aplicação de refatoração tornou o código menos repetido e executando a mesma proposta de forma simples e organizada. (Exemplo do código pode ver visto no repositório no arquivo “CalculoAliquota.java”).

Boa documentação

Criar um bom código significa criar um código bem documentado. O motivo para isso acontecer é que a comunicação não é apenas para o computador, mas também para que outras pessoas possam corrigir ou estender esses códigos. Código no mundo real nunca é escrito e depois esquecido.

Ele será modificado, estendido e mantido durante a vida útil do software produtos. Para fazer isso, precisamos de instruções e uma boa documentação.

O único documento que descreve seu código de forma completa e correta é o próprio código. Isso não significa automaticamente que é a melhor descrição

possível, mas na maioria das vezes, é a única documentação que você terá acessível. Você deve, portanto, fazer todo o possível para torná-la uma boa documentação. Por necessidade, o código é algo que mais pessoas, além do autor, devem ser capazes de entender.

As linguagens de programação são nosso meio de comunicação. A comunicação clara é vital. Com clareza, seu código ganha qualidade porque é menos provável que você cometa erros e é mais barato manter o código.

O código auto documentado é um código facilmente legível. É compreensível sem depender de documentação externa. Podemos melhorar a clareza do nosso código de muitas maneiras.

Como por exemplo:

- Escrever código simples com boa apresentação;
- Escolher nomes significativos;
- Decompor em funções atômicas;
- Escolher tipos descritivos;
- Enfatizar o código importante;
- Fornecer um cabeçalho de arquivo;
- Lidar com erros adequadamente;
- Escrever comentários significativos.

Ausência de duplicidades

Um código bem projetado não deve conter duplicações, pois códigos que não contém duplicações são códigos mais elegantes e simples. Remover códigos desnecessários é uma refatoração simples e que acrescenta muito ao código deixando o código menos frágil, essa refatoração pode ser a partir de duas partes semelhantes de código que diferem apenas em pequenos detalhes, fazendo uma função com parâmetros apropriados, que não seja necessária a repetição do código.

Com a remoção de duplicidades no código, torna-se mais fácil encontrar e corrigir bugs, pois o bug se encontra em um lugar específico sendo assim mais fácil de aplicar uma correção, além de diminuir a possibilidade de bugs duplicados, comprometendo claramente a segurança do código.

A maioria das duplicações ocorre por meio de programação copiar e colar, causando vários problemas a manutenibilidade do código. Códigos duplicados são desnecessários e devem ser retirados para um ganho expressivo no desenvolvimento.

Exemplo: No commit https://github.com/MoacirMSJ/TPPE-Trab1-TDD/commit/5d4064c4486c716f87bc10558b5baa8f42e245c0 feito no trabalho 2, as istancias de descricao e valor estavam

se reptindo na classe Deducao.java, para remover os codigos duplicados foi criado o metodo setInformacoesPadroesDeducao.