Relatório do Trabalho 1 da disciplina de Reuso de Software

1 - Introdução

O sistema escolhido como objeto de estudo para este trabalho foi o uaifood. O objetivo do trabalho consistia em analisarmos o sistema como um todo, utilizando os conhecimentos adquiridos na disciplina até o momento, para que assim conseguíssemos identificar code smells e más práticas e, a partir daí, realizamos um processo de refatoração do código desse sistema.

2 - Sobre o sistema

O programa que refatoramos tem o intuito de gerenciar restaurantes, semelhante ao que o Ifood faz, tanto que a interface do sistema é inspirada no tema de coloração desse.

2.1 Tecnologias utilizadas

- **IDE utilizada:** NetBeans;
- **Linguagem utilizada:** Java (JDK 16.0.2);
- **Framework de GUI:** Java Swing;

2.2 Problemas Identificados

Ao ser feita a análise do código, foram identificados quatro problemas principais:
1 - A classe ManipuladorArquivo.java tinha uma baixa coesão, pois nela estavam presentes várias operações que fugiam do escopo principal dela, tais como: mapeamento de dados e o CRUD completo do cliente, motoboy e do produto. Isso iria deixar o sistema muito suscetível a quebras no caso de inserir alguma nova funcionalidade, como por exemplo, um CRUD para restaurantes, que necessitaria alterar essa classe para ser possível.
2 - No mesmo arquivo citado no item anterior, foi identificado também duplicação de código, nos métodos dos CRUD de clientes, produtos e motoboy, pois a lógica seguida era praticamente a mesma para todos.
3 - As classes de domínio do sistema eram as responsáveis pela sua persistência. Ou seja, dentro delas tinha um método responsável por salvá-las no banco de dados. Isso é ruim porque se a persistência mudar de txt para SQL, por exemplo, seria necessário alterar cada uma. Detalhe, as classes de domínio que foram citadas, são: Cliente, Motoboy, Produto e Usuario.

4 - A presença de uma regra de negócio rígida. No caso, foi a regra para descontos que estava fixa dentro da classe Produto. Essa regra em questão setava o valor de desconto fixo em 5%, fazendo com que quando fosse necessário mudar o valor de desconto, o programador deveria modificar diretamente o código da classe, podendo assim gerar problemas em demais partes do código.

3 - Justificativa da escolha dos padrões de projeto

Os padrões escolhidos para serem implementados foram os seguintes: Strategy, Facade, Template Method e Factory Method. A seguir estão as justificativas para a escolha de cada um:
Strategy: Esse foi utilizado para resolver o problema da rigidez da lógica de negócios que a aplicação dos descontos apresentava. Para isso, o cálculo do desconto foi encapsulado, para que a classe Produto não conheça mais a lógica do desconto e passasse apenas a utilizar o desconto como uma ferramenta, não mais como parte do produto.
Facade: A estratégia do Facade foi utilizada para resolver o problema da média coesão da camada de persistência, mais especificamente da classe AbstractRepository. Essa classe estava gerenciando a lógica do template e realizando operações de I/O de baixo nível. Então, com o Facade, foi criado a classe FileHandler para esconder os detalhes mais baixos de entrada e saída e fornecer uma interface mais simples. Assim, as responsabilidades são separadas e a coesão melhorada.
Template Method: O template Method foi escolhido para resolver o problema da duplicação de código. A lógica de editar e excluir era um algoritmo com uma estrutura fixa, mas com passos variáveis . O Template Method permite definir o "esqueleto" do algoritmo uma única vez em uma classe base e deixar as subclasses implementarem os detalhes.
Factory Method: Este foi escolhido para resolver o problema da Violação da SRP(Princípio da responsabilidade única, que foi citado em aulas anteriores da disciplina) que estava presente nas classes de domínio da aplicação. Ao invés de competir ações mais complexas, como: gerar id, consultar o banco de dados e salvar o objeto; a essas classes, essas modificações foram movidas para as classes factory dedicadas. Consequentemente, desacoplando a lógica de domínio das lógicas de criação e persistência de dados.

4 - Descrição das refatorações realizadas

4.1. Padrão 1: Template Method (Método Modelo)

- **Objetivo:** Eliminar a **duplicação de código** nos métodos `editar` e `excluir` da classe `ManipuladorArquivo`.
- **Implementação:**
    1. Foi criado um novo pacote `classes.repository` para abrigar a nova camada de persistência.

2. Foi criada a classe abstrata `src/classes/repository/AbstractRepository.java`. Esta classe define o "esqueleto" (template) dos algoritmos `editar` e `excluir`, que consistiam em "ler tudo → limpar arquivo → reescrever tudo".
3. Os passos variáveis desses algoritmos (como o nome do arquivo, o ID da entidade e a conversão de/para string) foram definidos como métodos abstratos nesta classe:
    - `protected abstract String getEntityFileName();`
    - `protected abstract String entityToString(T entity);`
    - `protected abstract T stringToEntity(String line);`
    - `protected abstract int getEntityId(T entity);`
4. Foram criadas três classes concretas que herdam de `AbstractRepository`:
    - `src/classes/repository/ClienteRepository.java`
    - `src/classes/repository/MotoboyRepository.java`
    - `src/classes/repository/ProdutoRepository.java`
5. Cada classe concreta implementou os quatro métodos abstratos, fornecendo os detalhes específicos de sua entidade (ex: `getEntityId()` em `ClienteRepository` retorna `entity.getCodCliente()`).
6. A classe `src/classes/ManipuladorArquivo.java` foi **esvaziada** de sua lógica de CRUD e mapeamento. Ela foi mantida como uma fachada estática para não quebrar o resto da aplicação, e seus métodos agora apenas delegam as chamadas para os repositórios correspondentes (ex: `public static void editar(Cliente c) { clienteRepo.editar(c); }`).

4.2. Padrão 2: Facade (Fachada)

- **Objetivo:** Corrigir a **baixa coesão** da classe `AbstractRepository`, que estava misturando lógica de template com lógica de I/O (Input/Output) de baixo nível.
- **Implementação:**
    1. Foi criado um novo pacote `classes.util`.
    2. Foi criada a classe `src/classes/util/FileHandler.java`. Esta classe atua como uma "Fachada", simplificando as complexas operações de `BufferedReader` e `BufferedWriter` em métodos simples.
    3. Os métodos `leitor`, `escritor` e `limparArquivo` foram movidos do `AbstractRepository` para dentro do `FileHandler` e renomeados para `readAllLines`, `writeAllLines` (sobrescrevendo o arquivo) e `appendLine` (adicionando ao final do arquivo).
    4. A classe `src/classes/repository/AbstractRepository.java` foi refatorada para **usar** a `FileHandler`.

5. Os métodos `editar` e `excluir` foram otimizados: em vez de lerem e escreverem no arquivo em cada iteração do loop, eles agora usam `FileHandler.readAllLines` uma vez, modificam a lista de strings na memória e, ao final, usam `FileHandler.writeAllLines` para salvar o resultado de uma só vez.

4.3. Padrão 3: Factory Method (Fábrica)

- **Objetivo:** Corrigir a **Violação do Princípio da Responsabilidade Única (SRP)**, removendo das classes de domínio (`Cliente`, `Motoboy`, `Produto`) a responsabilidade de saber como gerar seus próprios IDs e como se salvar no banco de dados.
- **Implementação:**
    1. Foi criado um novo pacote `classes.factory`.
    2. Foram criadas três classes Factory:
        - `src/classes/factory/ClienteFactory.java`
        - `src/classes/factory/MotoboyFactory.java`
        - `src/classes/factory/ProdutoFactory.java`
    3. A lógica de geração de ID (ler todos os dados, pegar o último ID e somar 1) e a chamada de persistência (`ManipuladorArquivo.armazenar`) foram **movidas** dos métodos `init()` das classes de domínio para os métodos `create...()` das suas respectivas Factories.
    4. Os métodos `init()` foram completamente **removidos** das classes `Cliente.java`, `Motoboy.java` e `Produto.java`.
    5. O método abstrato `public abstract void init()` foi **removido** da superclasse `Usuario.java`, quebrando a dependência.
    6. As classes de interface gráfica (`CadastroCliente.java`, `CadastrarMotoboy.java` e `CadastroProduto.java`) foram modificadas. As chamadas `new Cliente(...)` seguidas de `cliente.init()` foram substituídas por uma única chamada à Factory: `ClienteFactory.createCliente(...)`.

4.4. Padrão 4: Strategy (Estratégia)

- **Objetivo:** Corrigir a **Lógica de Negócio Rígida** (hardcoded) do cálculo de descontos na classe `Produto`.
- **Implementação:**
    1. Foi criado um novo pacote `classes.strategy`.
    2. Foi criada a interface `src/classes/strategy/IDescontoStrategy.java`, definindo o contrato `float calcularPreco(float precoOriginal)`.
    3. Foram criadas duas implementações concretas desta interface:
        - `src/classes/strategy/SemDesconto.java` (retorna o preço original).

    - `src/classes/strategy/DescontoPromocional.java` (retorna `precoOriginal * 0.95f`).
4. A classe `src/classes/Produto.java` foi modificada:
    - Foi adicionado o atributo `private IDescontoStrategy descontoStrategy`.
    - O construtor foi ajustado para definir `this.descontoStrategy = new SemDesconto();` como o valor padrão.
    - Foi adicionado um novo método `setDescontoStrategy(IDescontoStrategy strategy)`.
    - O método `getValorAtual()` foi refatorado para uma única linha: `return this.descontoStrategy.calcularPreco(this.valor);`, delegando o cálculo para o objeto de estratégia.
5. As classes `src/classes/factory/ProdutoFactory.java` e `src/classes/repository/ProdutoRepository.java` foram atualizadas. Agora, ao criar ou carregar um `Produto`, elas verificam a data atual e usam `produto.setDescontoStrategy(...)` para injetar a estratégia correta (`DescontoPromocional` ou `SemDesconto`).

5 - Ganhos de reutilização e manutenibilidade

As refatorações realizadas permitiram que o código que antes apresentava os seguintes problemas: baixa coesão, duplicação de código e alto acoplamento, tenha recebido as modificações necessárias para a resolução desses. Abaixo, serão discorridos sobre algumas vantagens obtidas a partir dessas modificações.
Ganho em manutenibilidade: Aumento da coesão das classes, onde agora as responsabilidades estão claramente separadas. Isso é notado na lógica de domínio, que não se mistura mais com a persistência dos dados(como em ProdutoRepository). Além disso, vale destacar a diminuição do acoplamento, onde mudar o sistema de persistência, para sql por exemplo, se tornou bem mais simples, bastando criar novas implementações dos Repositórios sem precisar alterar nenhuma outra classe.
Quanto a facilitação da reutilização: O AbstractRepository agora é reutilizável para qualquer outra entidade que precise de CRUD. Além disso, o FileHandle é reutilizável em outros projetos que se tenha a necessidade de ler/escrever arquivos de texto. Vale destacar também que as Strategy desenvolvidas para resolver o problema do desconto citado anteriormente, podem ser reutilizadas e aplicadas a outras entidades, caso seja necessário.
Por fim, pode-se concluir que o sistema teve uma grande melhora em relação a componentização, diminuição de duplicação de código e redução considerável de acoplamento, além de uma melhora geral na coesão geral das classes do sistema.