NOME: BRUNO FELICIANO MARTINS

PADRÕES DE PROJETO E MULTICAMADA

TRABALHO FINAL

- Projeto escolhido: https://github.com/Brunofell/Joguinho-em-Java.git

- Pontos de melhoria:

1. Métodos complexos

Os métodos da classe jogo são muito extensos e verbosos, uma solução para este problema seria a repartição desses métodos em métodos menores, a partir dessas funções novas, poderemos chamá-las por recursão funções para evitar esse problema.

Um padrão de projeto para essa melhoria seria o Composite, esse padrão permite estruturar um método complexo dividindo-o em operações menores e reutilizáveis, que podem ser compostas recursivamente.

Um protótipo para melhorar esse tópico seria a alteração desses métodos para que fiquem mais simples, veja no exemplo abaixo:

```
private void inicio() {
    Jogavel guerreiro = new Jogavel("Kayn", "Guerreiro", 400, 150);
    Jogavel mago = new Jogavel("Veigar", "Mago", 200, 150);
    Jogavel atirador = new Jogavel("Caitlyn", "Atirador", 250, 200);
    Jogavel assassino = new Jogavel("Yone", "Assassino", 400, 5000);
    Jogavel alquimista = new Jogavel("Teemo", "Alquimista", 10, 120);
    this.campeoes.add(guerreiro);
    this.campeoes.add(mago);
    this.campeoes.add(atirador);
    this.campeoes.add(assassino);
    this.campeoes.add(alquimista);
    this.appendToOutput( text: "Escolha um personagem:");
    this.appendToOutput( text: "");
    this.appendToOutput(guerreiro.getNome());
    guerreiro.mostraPersonagem(guerreiro.getNome(), guerreiro.getVida(), guerreiro.getDano(), this, guerreiro.getClasse());
    this.appendToOutput( text: "");
    this.appendToOutput(mago.getNome());
    mago.mostraPersonagem(mago.getNome(), mago.getVida(), mago.getDano(), this, mago.getClasse());
    this.appendToOutput( text: "");
    this.appendToOutput(atirador.getNome());
    atirador.mostraPersonagem(atirador.getNome(), atirador.getVida(), atirador.getDano(), this, atirador.getClasse());
    this.appendToOutput( text: "");
    this.appendToOutput(assassino.getNome());
    assassino.mostraPersonagem(assassino.getNome(), assassino.getVida(), assassino.getDano(), this, assassino.getClasse());
    this.appendToOutput( text: "");
    this.appendToOutput(alquimista.getNome());
    alquimista.mostraPersonagem(alquimista.getNome(), alquimista.getVida(), alquimista.getDano(), this, alquimista.getClasse());
    this.appendToOutput( text: "---------------------------------");
}
```

Para otimizar esse método, podemos usar uma lista para definir os personagens e iterar sobre ela, adicionando cada um à lista de campeões e

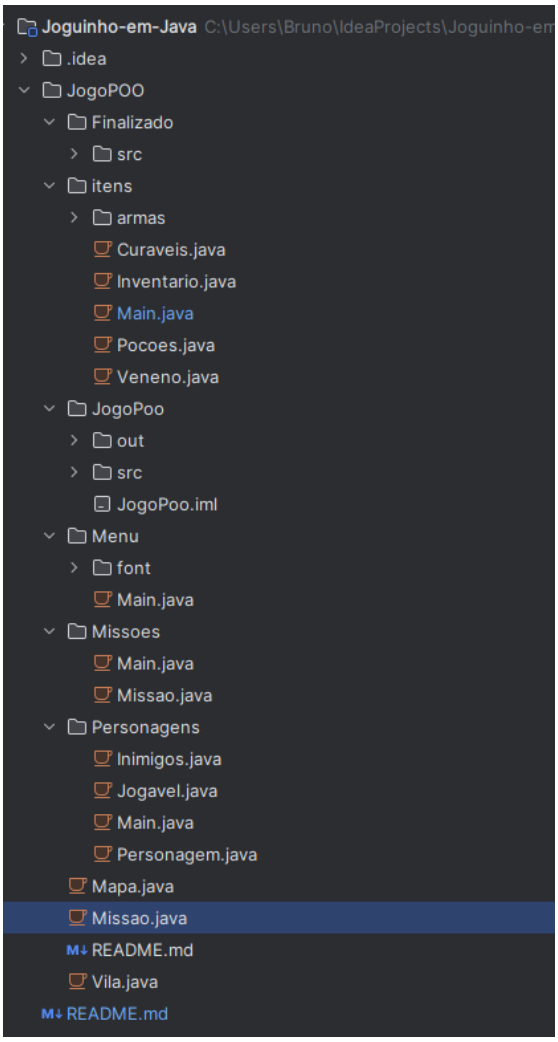
exibindo seus detalhes. Isso torna o código mais fácil de ler, reduzir duplicações e facilita a manutenção.

```
private void inicio() {
    List<Jogavel> personagens = Arrays.asList(
            new Jogavel("Kayn", "Guerreiro", 400, 150),
            new Jogavel("Veigar", "Mago", 200, 150),
            new Jogavel("Caitlyn", "Atirador", 250, 200),
            new Jogavel("Yone", "Assassino", 400, 5000),
            new Jogavel("Teemo", "Alquimista", 10, 120)
    );

    this.appendToOutput("Escolha um personagem:");
    this.appendToOutput("");

    for (Jogavel personagem : personagens) {
        this.campeoes.add(personagem);
        this.appendToOutput(personagem.getNome());
        personagem.mostraPersonagem(personagem.getNome(), personagem.getVida(), personagem.getDano(), this, personagem.getClasse());
        this.appendToOutput("");
    }

    this.appendToOutput("----------------------------------");
}
```

Para evitar métodos longos e repetitivos, três padrões de projeto podem ser aplicados, factory Method, builder e strategy. O factory Method centraliza a criação de objetos, permitindo que uma fábrica crie personagens com base em parâmetros, o que reduz a repetição de instâncias no método principal. O builder facilita a construção de objetos complexos, organizando as propriedades dos personagens de forma mais clara e legível. Já o strategy define comportamentos intercambiáveis, como a exibição de detalhes dos personagens, tornando o código mais modular e fácil de expandir.

2. Má organização de arquivos

Nesse projeto, existe um grande problema que é a má organização de pacotes, existem pastas vazias, pastas com arquivos que não estão sendo utilizados e pastas que estão fora de ordem e pessimamente organizadas.



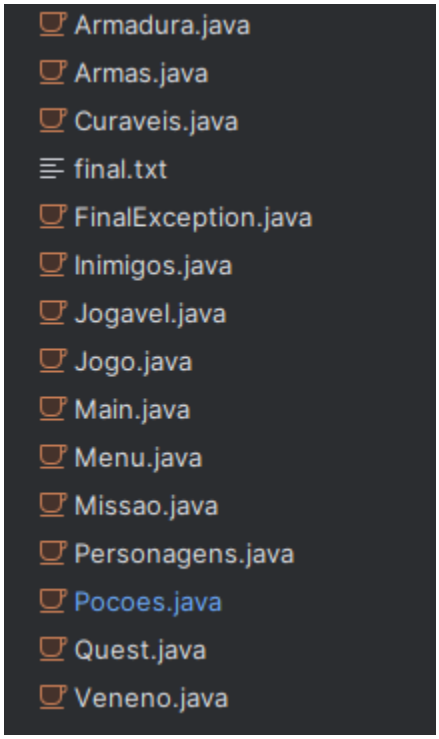
Um padrão de projeto que pode auxiliar nisso é o padrão de projeto arquitetural em camadas (Layered Architecture). Esse padrão sugere separar as responsabilidades em camadas bem definidas, como apresentação, negócio, serviço e dados, cada uma contendo apenas os arquivos e classes que correspondem à sua função. Essa estrutura garante que cada camada tenha uma finalidade clara, facilitando a organização e o acesso ao código necessário.

A solução para esse problema seria investir tempo na organização do projeto, sempre ter em mente a estrutura que quer seguir, remover arquivos desnecessários e revisar a estrutura atual do projeto.

Para resolver problemas de organização de pacotes em um projeto, três padrões de projeto são úteis. O Padrão MVC (Model-View-Controller) separa a lógica de apresentação, de negócios e de dados, facilitando a manutenção e a localização de arquivos. O Padrão de Camadas divide o sistema em camadas distintas, como apresentação e negócios, o que ajuda a entender as responsabilidades de cada parte. Por fim, o Padrão de Microserviços organiza a aplicação em serviços independentes, permitindo que cada um tenha sua própria estrutura de pacotes, promovendo assim uma melhor colaboração e escalabilidade. Esses padrões ajudam a manter um projeto mais organizado e gerenciável.

3. Classes que não estão sendo utilizadas

No projeto existe classes e metodos que não estão sendo utilizadas. Uma solução para isso seria a exclusão dessas classes para melhorar a organização do código e deixá-lo mais simples.



Um princípio que podemos utilizar seria o YAGNI (You Aren’t Gonna Need It), que sugere que classes e metodos redundantes nã o devem fazer parte do projeto, apenas estruturas necessarias devem permanecer.

Um protótipo para esse ponto de melhoria seria apenas deletar essas classes redundantes, mas antes deve-se certificar que elas não então sendo usadas para não causar nenhum problema.

Para evitar classes sem uso, redundantes ou inúteis, três padrões de projeto recomendados são: singleton, facade e observer.

O singleton garante que uma classe tenha apenas uma instância, evitando a criação desnecessária de múltiplos objetos para o mesmo propósito. O facade simplifica o código ao fornecer uma interface unificada para sistemas complexos, reduzindo a necessidade de classes auxiliares repetitivas. Já o observer permite que objetos reajam a mudanças sem precisar de classes intermediárias, removendo dependências desnecessárias.

Esses padrões ajudam a manter o projeto enxuto e eficiente, eliminando redundâncias.

4. If-else muito extenso:

If-else muito extenso geralmente indica uma falta de padrões de projeto, pois existem várias abordagens que podem tornar o código mais modular e organizado, especialmente quando o If-else lida com diferentes condições de maneira repetitiva.

Um padrão de projeto eficaz para evitar If-else extensos é o strategy. Esse padrão permite que você encapsule diferentes comportamentos (estratégias) em classes separadas, que podem ser selecionadas e aplicadas dinamicamente conforme necessário.

Temos esse método comprarItem() na classe Jogo que possui um If-else imenso:

```
private void comprarItem(String escolha) {
    if (Objects.equals(escolha, b: "Espada")) {
        if (this.ouro < 50.0F) {
            this.appendToOutput( text: "Você está pobre! Dinheiro insuficiente! ouro: " + this.ouro);
        } else {
            this.ouro -= 50.0F;
            this.danoPersonagem += 50;
            this.espada.Espada(50, this);
            this.appendToOutput( text: "Ouro atual: " + this.ouro);
            this.appendToOutput( text: "Seu dano atual é de: " + this.danoPersonagem);
        }
    } else if (Objects.equals(escolha, b: "Poção de vida")) {
        if (this.ouro < 40.0F) {
            this.appendToOutput( text: "Você está pobre! Dinheiro insuficiente! ouro: " + this.ouro);
        } else {
            this.ouro -= 40.0F;
            this.vidaPersonagem += 20;
            this.appendToOutput( text: "Você comprou a poção de vida!! ouro: " + this.ouro);
            this.potion.PotDefesa(this, this.potion.getPotDefesa(), this.vidaPersonagem);
            this.appendToOutput( text: "Sua vida atual é de: " + this.vidaPersonagem);
        }
    } else if (Objects.equals(escolha, b: "Poção de força")) {
        if (this.ouro < 60.0F) {
            this.appendToOutput( text: "Você está pobre! Dinheiro insuficiente! ouro: " + this.ouro);
        } else {
            this.ouro -= 60.0F;
            this.danoPersonagem += 60;
            this.appendToOutput( text: "Você comprou a poção de força!! ouro: " + this.ouro);
            this.potion.PotForca(this, this.potion.getPotForca(), this.danoPersonagem);
            this.appendToOutput( text: "");
```

Para consertar isso podemos criar classes individuais para a compra dos itens, e um interface comum para ser implementada por essas classes. Dessa forma, cada item tem sua própria lógica de verificação de recursos e atualização de status, eliminando a necessidade de uma estrutura de decisão complexa.

```
public interface ItemCompravel {  2 usages  new *
    void comprar(Jogador jogador); // 'Jogador' seria a classe que contém 'ouro', 'danoPersonagem', etc.  no usages  new *
}
```

```java
public class Espada implements ItemCompravel {
    @Override
    public void comprar(Jogador jogador) {
        if (jogador.getOuro() < 50.0F) {
            jogador.appendToOutput("Você está pobre! Dinheiro insuficiente! ouro: " + jogador.getOuro());
        } else {
            jogador.decrementarOuro(50.0F);
            jogador.incrementarDano(50);
            jogador.appendToOutput("Ouro atual: " + jogador.getOuro());
            jogador.appendToOutput("Seu dano atual é de: " + jogador.getDanoPersonagem());
        }
    }
}

public class PocaoDeVida implements ItemCompravel {
    @Override
    public void comprar(Jogador jogador) {
        if (jogador.getOuro() < 40.0F) {
            jogador.appendToOutput("Você está pobre! Dinheiro insuficiente! ouro: " + jogador.getOuro());
        } else {
            jogador.decrementarOuro(40.0F);
            jogador.incrementarVida(20);
            jogador.appendToOutput("Você comprou a poção de vida!! ouro: " + jogador.getOuro());
            jogador.appendToOutput("Sua vida atual é de: " + jogador.getVidaPersonagem());
        }
    }
}
```

Alguns padrões de projeto são eficazes para reduzir o excesso de estruturas if-else. O strategy organiza comportamentos em classes distintas, permitindo que cada ação seja tratada individualmente, sem verificar muitas condições. O factory method é útil para criar instâncias de objetos diferentes conforme parâmetros, eliminando a necessidade de vários if-else. Já o chain of responsibility permite passar a responsabilidade de decisão ao longo de uma cadeia de classes, onde cada uma verifica apenas a condição que lhe cabe. Esses padrões tornam o código mais modular, legível e fácil de manter.

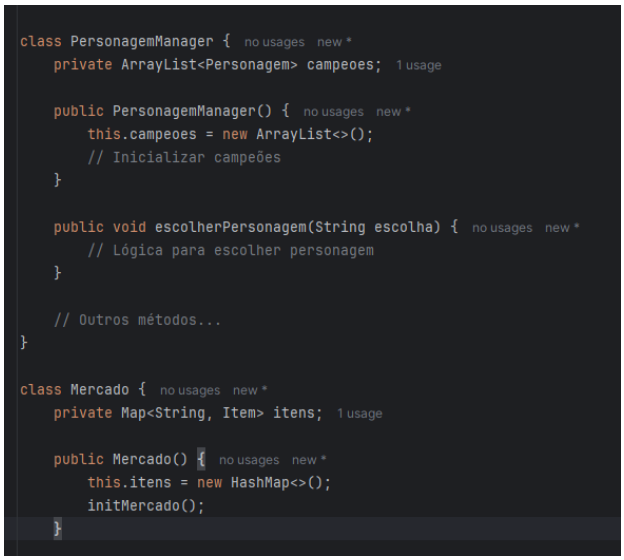
5. Classe Jogo assume muitas responsabilidades

```java
public class Jogo extends JFrame {
    private JTextArea outputArea;
    private JTextField inputField;
```

A classe intitulada “Jogo”, possui muitos métodos e funcionalidades, ter uma classe com muitas responsabilidades é um antipadrão em programação orientada a objetos e pode levar a vários problemas sérios que afetam a qualidade, a manutenção e a extensibilidade do código.

Um padrão que podemos utilizar é o Single Responsibility Principle (SRP), O SRP é um dos princípios fundamentais da programação orientada a objetos e sugere que cada classe deve ter uma única responsabilidade ou tarefa. Isso facilita a manutenção e a evolução do sistema, além de tornar o código mais claro e modular.

Um protótipo para esse caso seria segmentar a classe Jogo em outras classes menores, e depois só chamar essas classes novas dentro da principal para executar as funcionalidades



```
class PersonagemManager {  no usages  new *
    private ArrayList<Personagem> campeoes;  1 usage

    public PersonagemManager() {  no usages  new *
        this.campeoes = new ArrayList<>();
        // Inicializar campeões
    }

    public void escolherPersonagem(String escolha) {  no usages  new *
        // Lógica para escolher personagem
    }

    // Outros métodos...
}

class Mercado {  no usages  new *
    private Map<String, Item> itens;  1 usage

    public Mercado() {  no usages  new *
        this.itens = new HashMap<>();
        initMercado();
    }
```

Para melhorar classes imensas, o padrão Singleton pode ser utilizado para garantir que uma classe tenha apenas uma instância, facilitando o controle de estados. O padrão Strategy permite encapsular algoritmos em classes separadas, promovendo a coesão e a troca de implementações, enquanto o padrão Facade simplifica a interação com subsistemas complexos, oferecendo uma interface única que oculta a complexidade. Esses padrões ajudam a tornar o código mais legível e fácil de manter

Referências:

https://refactoring.guru/pt-br/design-patterns/composite

https://awari.com.br/solid-kiss-dry-yagni

https://refactoring.guru/pt-br/design-patterns/builder

https://medium.com/@tbaragao/solid-s-r-p-single-responsibility-principle-2760ff4a7edc

https://refactoring.guru/pt-br/design-patterns