Relatório - Batalha Naval

Programação Orientada a Objetos 2

Autor: Gabriel Rodrigues Barbosa[⋆]

Profª Dra. Fabíola Souza Fernandes Pereira

[⋆] Matrícula: 31711BSI035, Email: gabriel.barbosa@ufu.br

Resumo

O projeto consiste no desenvolvimento de um jogo de batalha naval em Java, aplicando Padrões de Projetos e Princípios de Projetos, onde dois jogadores competem para afundar os navios do oponente. O jogo é implementado em um ambiente de console, onde os jogadores posicionam estrategicamente seus navios em um tabuleiro e tentam adivinhar as posições dos navios do oponente para realizar ataques precisos.

Sumário

1 Introdução

O jogo Batalha Naval é um clássico jogo de tabuleiro em que dois jogadores competem para afundar os navios do oponente. Cada jogador possui seu próprio tabuleiro onde posiciona seus navios e tenta adivinhar a posição dos navios do oponente para atacá-los.

Este relatório documentará a implementação do jogo Batalha Naval em Java, destacando as classes principais, bem como os padrões de projeto utilizados para organizar e estruturar o código.

1.1 Objetivo

O objetivo do projeto foi criar uma versão digital do clássico jogo de tabuleiro Batalha Naval, permitindo que duas pessoas joguem entre si em um ambiente gráfico intuitivo. Algumas metas específicas incluíram:

- Implementar a Lógica do Jogo: Desenvolver a lógica do jogo Batalha Naval, incluindo a colocação aleatória de navios nos tabuleiros dos jogadores, a execução de ataques alternados e a verificação das condições de vitória.
- Criar uma Interface Gráfica Amigável: Desenvolver uma interface gráfica (GUI) intuitiva e fácil de usar, que permita aos jogadores interagir com os tabuleiros, realizar ataques e acompanhar o progresso do jogo de forma clara e visual.
- Garantir a Interação Multiplayer: Implementar a lógica necessária para permitir que dois jogadores interajam com o jogo simultaneamente, alternando entre suas respectivas jogadas e mantendo a integridade das informações do jogo entre os dois tabuleiros.
- Adicionar Aleatoriedade e Desafio: Introduzir elementos de aleatoriedade na colocação dos navios nos tabuleiros, garantindo que cada partida seja única e desafiadora, proporcionando aos jogadores uma experiência dinâmica e empolgante.
- Facilitar a Identificação dos Elementos do Jogo: Garantir que os elementos do jogo, como os navios, os ataques realizados e os resultados, sejam claramente identificados na interface gráfica, proporcionando uma experiência visualmente agradável e compreensível.
-

Em suma, o objetivo do projeto foi criar uma versão digital cativante e envolvente do jogo Batalha Naval, oferecendo uma experiência de jogo divertida e desafiadora para dois jogadores.

2 Projeto

2.1 Classes Principais

2.2 TF_BatalhaNaval

Esta classe é responsável por criar a interface gráfica do jogo Batalha Naval. Ela estende a classe JFrame e implementa a interface ActionListener para lidar com eventos de botão. Esta classe representa a visualização do jogo e interage com a classe Game para atualizar e exibir o estado do jogo na interface gráfica.

O código implementa a lógica básica de um jogo de Batalha Naval em uma interface gráfica usando Java Swing. Veja suas funcionalidades:

- **Pacotes e imports:** O código começa importando as classes necessárias dos pacotes *javax.swing.** e *java.awt.**. Esses pacotes são usados para criar a interface gráfica do jogo.
- **Classe 'TF_BatalhaNaval':** Esta classe estende *JFrame* e implementa *ActionListener*, o que significa que ela é uma janela de aplicativo e pode ouvir eventos de ação, como cliques de botão. Ela possui membros para armazenar os botões dos jogadores 1 e 2, uma instância do jogo (Game) e o número do jogador atual. O construtor recebe uma instância de Game como argumento e inicializa a interface gráfica do jogo. A lógica principal do jogo está contida neste construtor, onde os tabuleiros dos jogadores são inicializados, os navios são adicionados aleatoriamente e os botões são configurados para responder aos cliques do jogador. Dois métodos principais, *updatePlayer1Board* e *updatePlayer2Board*, são chamados para atualizar os tabuleiros dos jogadores após cada jogada. Eles também verificam se houve uma vitória após cada jogada e exibem uma mensagem de vitória, se necessário. O método *adicionarNaviosAleatorios* é usado para adicionar navios aleatórios aos tabuleiros dos jogadores.
- **Método main:** Este método cria uma instância do jogo (Game) e uma instância de *AtaquePosicao* para simular um ataque de um jogador. Em seguida, ele inicia a interface gráfica do jogo na thread de despacho de eventos Swing.

Esse código fornece uma estrutura básica para implementar um jogo de Batalha Naval em uma interface gráfica usando Java Swing. Ele pode ser expandido e aprimorado com recursos adicionais, como funcionalidades de multiplayer, efeitos visuais e animações.

2.3 Game

A classe Game é responsável por gerenciar a lógica do jogo Batalha Naval. Ela mantém o estado atual do jogo, incluindo os tabuleiros dos jogadores, controla os ataques e verifica as condições de vitória. Além disso, ela recebe atualizações do estado do jogo por meio de um objeto *GameObserver*, Veja suas funcionalidades.

- **Instância Única do Jogo:** A classe *Game* implementa o padrão *Singleton*, garantindo que haja apenas uma instância do jogo em execução.
- **Inicialização do Tabuleiro:** Ao instanciar um novo jogo, os tabuleiros dos dois jogadores são inicializados com células vazias.
- **Adição de Navios Aleatórios:** O método *adicionarNaviosAleatorios(int jogador)* adiciona navios aleatórios aos tabuleiros dos jogadores. Cinco navios são adicionados para cada jogador.
- **Ataque em uma Posição:** O método *atacar(int jogador, int row, int col)* permite que um jogador ataque uma posição no tabuleiro do oponente. A lógica de ataque é simulada neste método.
- *Verificação de Vitória:* O método *verificarVitoria()* verifica se um dos jogadores venceu o jogo. Ele verifica se todos os navios de um jogador foram destruídos no tabuleiro do oponente.
- **Notificação de Observadores:** A classe Game mantém uma instância de GameObserver para notificar os observadores sobre eventos importantes no jogo, como ataques e mudanças de estado.

- **Fim de Jogo e Jogador Vencedor:** A variável *jogoAcabou* é definida como verdadeira quando um jogador vence o jogo. O método *getJogadorVencedor()* retorna o número do jogador vencedor.

Essas funcionalidades são essenciais para o funcionamento básico do jogo de Batalha Naval. Elas fornecem a estrutura necessária para gerenciar os tabuleiros, realizar ataques, verificar a vitória e determinar o jogador vencedor.

2.4 Navio

A classe Navio representa um navio no jogo Batalha Naval. Ela contém informações sobre o tipo e a posição do navio no tabuleiro.

2.5 NavioFactory

A interface NavioFactory define um contrato para a criação de navios no jogo Batalha Naval. Ela possui um método criarNavioAleatorio() para criar navios de forma aleatória.

2.6 NavioFactoryImpl

A classe NavioFactoryImpl implementa a interface NavioFactory e fornece uma implementação concreta do método criarNavioAleatorio(). Ela é responsável por criar navios aleatórios no jogo Batalha Naval.

3 Padrões de Projeto

3.1 Padrão Singleton

O padrão Singleton é utilizado na classe *Game* para garantir que apenas uma instância do jogo seja criada. Isso é útil em situações onde precisamos garantir que apenas uma instância de uma classe esteja disponível em todo o sistema. Isso simplifica o gerenciamento de recursos e evita inconsistências.

3.2 Padrão Observer

O padrão Observer é utilizado para permitir que objetos interessados (observadores) sejam notificados e atualizados quando ocorrem mudanças no estado de um objeto (sujeito). No projeto de Batalha Naval, esse padrão é aplicado na comunicação entre a classe Game e as classes que observam o jogo, como a classe 'TF_BatalhaNaval'. Quando há uma mudança no estado do jogo, como um ataque realizado, a classe 'Game' notifica seus observadores, como a interface gráfica, para que possam ser atualizados e refletir o novo estado do jogo na tela.

3.3 Padrão Factory Method

O padrão Factory Method é utilizado para encapsular a criação de objetos, permitindo que uma classe delegue a responsabilidade de criar instâncias de subclasses para classes filhas. No projeto de Batalha Naval, esse padrão pode ser aplicado na criação de navios. A interface 'NavioFactory' define um método de fábrica 'criarNavio', enquanto a classe 'NavioFactoryImpl' implementa esse método para criar instâncias específicas de navios, como submarinos, porta-aviões, etc. Isso permite que a criação de novos tipos de navios seja facilmente estendida sem modificar o código existente.

3.4 Padrão Strategy

O padrão Strategy é utilizado para definir uma família de algoritmos, encapsulá-los e torná-los intercambiáveis. No projeto de Batalha Naval, poderíamos aplicar esse padrão para representar diferentes estratégias de ataque. Por exemplo, poderíamos ter diferentes estratégias de ataque para diferentes tipos de navios, como submarinos, porta-aviões, etc. Cada estratégia implementaria a lógica específica de ataque para o tipo de navio correspondente, e a classe 'Game' poderia trocar entre essas estratégias dinamicamente com base nas circunstâncias do jogo. No entanto, essa implementação específica não está presente no código fornecido.

4 Princípios de Projeto

4.1 Princípio da Responsabilidade Única (SRP)

Esse princípio ajuda a manter o código organizado e coeso, garantindo que cada classe tenha uma única responsabilidade bem definida. No projeto de batalha naval, a classe 'Game' é responsável por gerenciar a lógica do jogo, como realizar ataques, verificar vitórias e adicionar navios aos tabuleiros. Por outro lado, a classe 'TF_BatalhaNaval' cuida exclusivamente da interface gráfica do usuário, como exibir os tabuleiros e os botões de ataque. Essa separação clara de responsabilidades facilita a compreensão do código e torna mais fácil fazer alterações específicas em cada parte do sistema sem afetar as outras.

4.2 Princípio da Abertura/Fechamento (OCP)

Embora não tenha sido explicitamente aplicado no código fornecido, o OCP incentiva a criação de código que possa ser estendido sem precisar ser modificado. No contexto do jogo de batalha naval, seguindo esse princípio, poderíamos projetar o sistema de forma que novos recursos, modos de jogo ou tipos de navio pudessem ser adicionados posteriormente sem a necessidade de alterar o código existente. Isso contribuiria para a manutenibilidade do código, pois reduziria a necessidade de modificar classes existentes sempre que novos requisitos surgissem.

4.3 Princípio da Substituição de Liskov (LSP)

Esse princípio garante que as classes derivadas possam ser substituídas pelas classes base sem afetar o comportamento esperado do programa. No projeto de batalha naval, isso poderia ser aplicado garantindo que cada classe de navio derivada da classe base Navio mantenha o mesmo contrato de comportamento. Por exemplo, independentemente do tipo de navio (submarino, porta-aviões, etc.), todos devem ser capazes de serem colocados nos tabuleiros, atacados e representados de forma consistente. Ao seguir o LSP, o código se torna mais coeso e consistente, facilitando a manutenção e a extensão do sistema.

4.4 Princípio da Inversão de Dependência (Dependency Inversion Principle - DIP)

A classe Game depende de uma interface GameObserver em vez de implementações concretas. Isso permite que diferentes tipos de observadores sejam facilmente injetados na classe Game sem modificar seu código interno, seguindo o princípio da abstração e reduzindo o acoplamento.

5 Conclusão

O projeto de Batalha Naval implementado em Java apresenta uma estrutura sólida e bem organizada, demonstrando o uso de padrões de projeto e princípios de design importantes. Através da aplicação de padrões como *Singleton*, *Observer* e o respeito a princípios como SRP (*Single Responsibility Principle*), OCP (*Open/Closed Principle*) e DIP (D*ependency Inversion Principle*), o código se torna modular, extensível e de fácil manutenção. Veja a interface do jogo, com dois tabuleiros para se jogar em dupla:

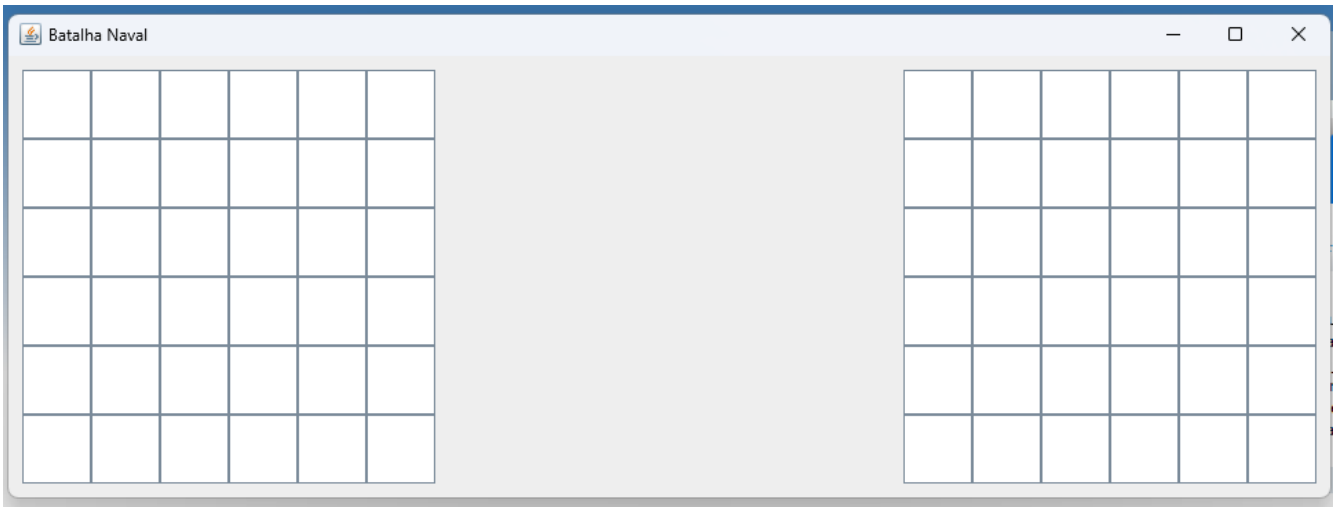

Figura 1: Interface do Jogo

Através da utilização do padrão *Singleton*, garantimos que apenas uma instância do jogo seja criada, evitando inconsistências e simplificando o gerenciamento de recursos. O padrão *Observer* possibilita a comunicação eficaz entre o jogo e outros componentes do sistema, mantendo uma baixa dependência entre eles. Além disso, o código respeita o princípio da responsabilidade única, onde cada classe tem uma responsabilidade bem definida, facilitando a compreensão e a manutenção do sistema.

O projeto também implementa a lógica do jogo de Batalha Naval de forma clara e eficiente, permitindo aos jogadores atacarem posições no tabuleiro e verificarem a ocorrência de vitória. A adição de funcionalidades como o controle do turno dos jogadores e a exibição de mensagens de vitória tornam a experiência do jogo mais completa e interativa.

Em resumo, o projeto de Batalha Naval em Java representa não apenas a implementação de um jogo clássico, mas também a aplicação de conceitos fundamentais de design de software, tornando-o uma excelente base para aprendizado e prática de desenvolvimento orientado a objetos.