Implementação do Tabuleiro Hnefatafl utilizando padrões de projeto

Jéssica Bernardi Petersen[1]

[1]Departamento de Engenharia de Software – Universidade do Estado de Santa Catarina (UDESC)

Ibirama – SC– Brasil

{jessica041ster@gmail.com}

Abstract. *Design Patterns describe problems that happen very often and can be used several times without rework. The Singleton pattern ensures that only one object is instantiated for a given class. It was used to instantiate the movement object. The Abstract Factory pattern creates a family of related objects through an interface and was used to create the board and the moves. The MVC pattern separates the model, view and controller layers to improve flexibility and reuse. The Observer pattern allows the communication and automation of data between the controller and view layers. The Command pattern registers the client's requests, allowing to undo such operations through the encapsulation of the method calls. In this work, this pattern was applied to make a log and register all piece moves in the game. Finally, the Proxy pattern controls the access to a specific object using a substitute or location marker and was used to control the player moves.*

Resumo. *Os Padrões de projeto descrevem problemas que sempre ocorrem, podendo serem utilizados repetidamente sem que se faça duas vezes iguais. O Padrão Singleton garante que apenas um objeto seja instanciado para uma devida classe e foi utilizado na instanciação do objeto movimento. O Abstract Factory cria uma família de objetos relacionados através de uma interface e foi utilizado para criar o tabuleiro e os movimentos. O MVC separa os objetos modelo, visão e controlador em pacotes separados para aumentar a flexibilidade e a reutilização e o Observer permite a comunicação e a automatização dos dados da camada de controller com a visão. O Command enfileira ou registra as solicitações dos clientes permitindo desfazer tais operações através do encapsulamento da chamada do método, sendo utilizado no trabalho para registrar os logs de movimento das peças. O Proxy permite controlar o acesso a um determinado objeto por meio de um substituto ou marcador da localização e foi utilizado para controlar as jogadas de acordo com o jogador.*

1. Introdução

Alexander Christopher observou muitas construções, cidades, ruas e qualquer espaço habitacional que os seres humanos moram e descobriu que as boas construções possuíam algo em comum. Um exemplo é o dos pórticos, que são estruturalmente

distintos, porém, ambos são considerados de alta qualidade, onde um pode ser uma transição da calçada para a porta da entrada e o outro um lugar de sombra para um dia quente. Os dois pórticos podem solucionar um problema em comum de maneiras diferentes, assim, ele descobriu que poderia ter similaridade entre projetos de alta qualidade, ao qual ele deu o nome de *padrões* (SHALLOWAY; TROTT, 2004).

Segundo Shalloway e Trott (2004), cada padrão descreve um problema que sempre ocorre e podemos utilizá-lo repetidas vezes de modos diferentes . De acordo com Gamma et al (2000), normalmente um padrão possui quatro elementos essenciais, que são: o nome para identificador do padrão, o problema, que explica em quais situações podem ser aplicados, a solução, que cita os elementos que o compõe, seus relacionamentos, responsabilidades e colaborações, e a consequência,a que cita as vantagens e desvantagens e o resultado de sua aplicação.

Os padrões são classificados por dois critérios (Tabela 1): finalidade, sendo o que o padrão faz podendo ser de criação, estrutural ou comportamental e o escopo, que especifica se o padrão se aplica a classes ou a objetos (GAMMA; et al, 2000).

Tabela 1: O espaço dos padrões de projeto. Fonte: Gama et al (2000).

		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**classe**	Factory Method	Adapter (class)	Interpreter Template Method
	objeto	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

2. Padrões de Criação

Esses padrões ajudam o sistema a ser mais independente em relação a como seus objetos são criados, compostos e representados. Um padrão de criação de classe “usa a herança variar a classe que é instanciada”, já um padrão de criação de objetos “delegará a instanciação para outro objeto” (GAMMA; et al, 2000).

2.1. Singleton

O singleton cria objetos únicos e só há uma instância garantido que somente um objeto seja instanciado para uma dada classe (FREEMAN; FREEMAN, 2009) e fornece um ponto global para acessá-lo (GAMMA; et al, 2000).

```java
public class MovimentoOutros extends Movimento {

    private static MovimentoOutros instance;

    public synchronized static MovimentoOutros getInstance() {
        if (instance == null) {
            instance = new MovimentoOutros();
        }
        return instance;
    }
```

Figura 1: Padrão Singleton na classe MovimentoOutros

O padrão Singleton foi utilizado nas seguintes classes de movimento: MovimentoOutros (Figura 1) e MovimentoTablut (Figura 2). Como as regras de movimentação dos mercenários é diferente no tabuleiro Tablut dos outros dois, foi feito uma classe abstrata Movimento ao qual implementa os movimentos genéricos, que são iguais nos três tabuleiros e nas classes herdeiras (MovimentoOutros e MovimentoTablut), foram criados os métodos que diferenciam os movimentos de um tabuleiro do outro, como o tratamento para os mercenários que no Tablut são permitidos irem para o refúgio.

```java
public class MovimentoTablut extends Movimento {

    private static MovimentoTablut instance;

    public synchronized static MovimentoTablut getInstance() {
        if (instance == null) {
            instance = new MovimentoTablut();
        }
        return instance;
    }
```

Figura 2: Padrão Singleton na classe MovimentoTablut

Dessa forma, para garantir que apenas um tipo de movimento vai ser criado, foi utilizado o padrão singleton. O Padrão singleton será instanciado nas classes concretas da Factory de cada tabuleiro (Figura 3).

```java
public class FactoryBrandubh extends Factory{

    private Tabuleiro Brandubh;
    private Movimento movimento;

    @Override
    public Tabuleiro createTabuleiro() {
        if(Brandubh == null)
            Brandubh = new TabuleiroBrandubh();
        return Brandubh;
    }

    @Override
    public Movimento createMovimento() {
        if(movimento == null)
            movimento = getInstance();
        return movimento;
    }
}
```

Figura 3: Instanciação do Padrão Singleton em uma das classes Singleton

2.2. Abstract Factory

O Abstract Factory vai fornecer uma interface para criar famílias de objetos relacionados/dependentes sem especificar suas classes concretas.

Utilizou-se o Abstract Factory (Figuraa 4) para criar o tabuleiro e os movimentos do, de acordo com a opção que o jogador escolher. Foi desenvolvida uma classe abstrata Movimento e duas classes concretas que *extends* essa classe abstrata, uma classe abstrata Tabuleiro e três classes concretas que *extends* essa classe abstrata e uma classe abstract Factory e três factorys concretas que *extends* essa classe abstrata aos quais serão responsável pela criação dos objetos concretos de movimento e tabuleiro.

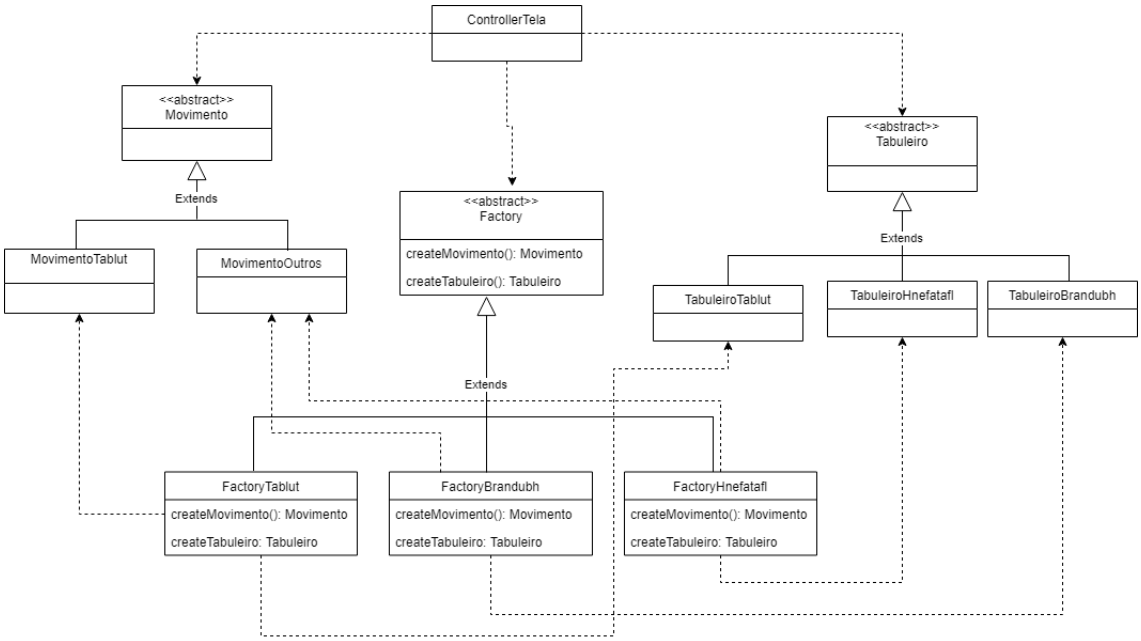
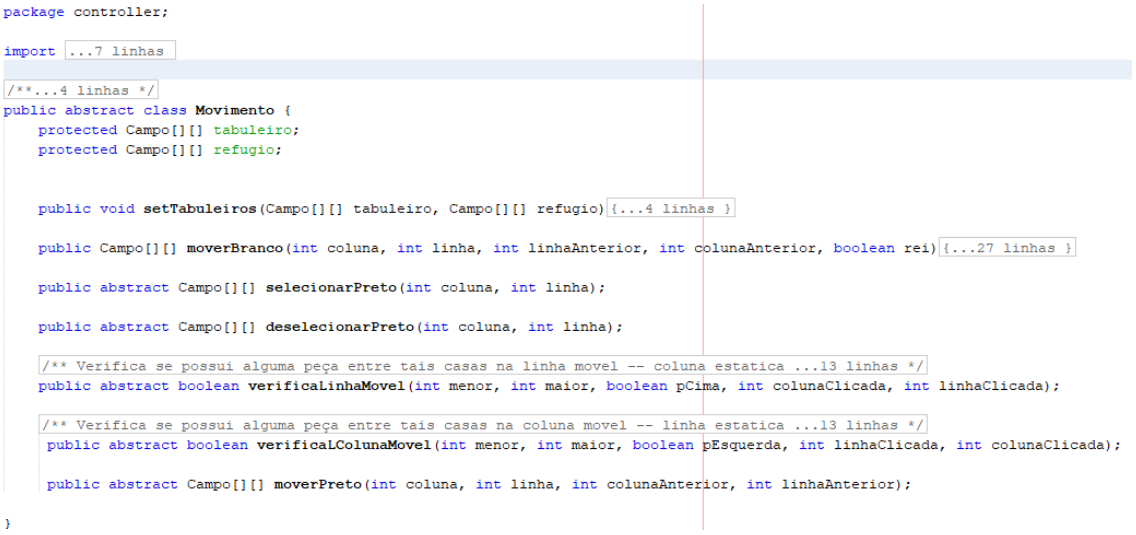


Figura 4: Abstract Factory

Utilizou-se esse padrão na criação dos movimentos, pois os mercenários podem ir nos refúgios se o tabuleiro for o Tablut, tendo assim um comportamento diferente dos outros dois tabuleiros. Já para a criação do tabuleiro em si, por possuírem tamanhos diferentes, foi feito uma classe concentra para cada.

Na classe abstrata Movimento (Figura 5) foram criados métodos que são abstratos, aos quais serão implementados nas classes filhas para realizarem os movimentos específicos de cada tabuleiro, como selecionarPreto() e deselecionarPreto() e os métodos genéricos que serão os mesmos para todos os tabuleiros, como o moverBranco().



```
package controller;

import ...7 linhas

/**...4 linhas */
public abstract class Movimento {
    protected Campo[][] tabuleiro;
    protected Campo[][] refugio;


    public void setTabuleiros(Campo[][] tabuleiro, Campo[][] refugio){...4 linhas }

    public Campo[][] moverBranco(int coluna, int linha, int linhaAnterior, int colunaAnterior, boolean rei){...27 linhas }

    public abstract Campo[][] selecionarPreto(int coluna, int linha);

    public abstract Campo[][] deselecionarPreto(int coluna, int linha);

    /** Verifica se possui alguma peça entre tais casas na linha movel -- coluna estatica ...13 linhas */
    public abstract boolean verificaLinhaMovel(int menor, int maior, boolean pCima, int colunaClicada, int linhaClicada);

    /** Verifica se possui alguma peça entre tais casas na coluna movel -- linha estatica ...13 linhas */
     public abstract boolean verificaLColunaMovel(int menor, int maior, boolean pEsquerda, int linhaClicada, int colunaClicada);

     public abstract Campo[][] moverPreto(int coluna, int linha, int colunaAnterior, int linhaAnterior);

}
```

Figura 5: Classe abstrata Movimento

Na classe MovimentoOutros (Figura 6) e MovimentoTablut foram implementados os métodos abstratos da classe Movimento para realizar as validações da classe em específico de acordo com o tabuleiro escolhido.

```
package controller;

import ...4 linhas

/**...4 linhas */
public class MovimentoOutros extends Movimento {

    private static MovimentoOutros instance;

    public synchronized static MovimentoOutros getInstance() {...6 linhas }

    @Override
    public Campo[][] selecionarPreto(int coluna, int linha) {...4 linhas }

    @Override
    public Campo[][] deselecionarPreto(int coluna, int linha) {...4 linhas }


    @Override
    public Campo[][] moverPreto(int coluna, int linha, int colunaAnterior, int linhaAnterior) {...5 linhas }

    @Override
    public boolean verificaLinhaMovel(int menor, int maior, boolean pCima, int colunaClicada, int linhaClicada) {...19 linhas }

    @Override
    public boolean verificaLColunaMovel(int menor, int maior, boolean pEsquerda, int linhaClicada, int colunaClicada) {...19 linhas }

}
```

Figura 6: Classe MovimentoOutros

A classe abstrata Tabuleiro (Figura 7) possui apenas métodos abstratos que devem serem implementados em suas classes filhas. Esses métodos são responsáveis por dimensionar o tamanho do tabuleiro e distribuir cada tipo de campo e peça.

```java
package controller;

import model.Campo;

/**...4 linhas */
public abstract class Tabuleiro  {

    /** Distribui os Campos na tela (CampoNormal ou Refugio) de acordo com o tabuleiro ...7 linhas */
    public abstract Campo[][] distribuirCampos(int normal, int rei);

    /** Distribui as peças (Branco - Preto - Rei) nas casas ...5 linhas */
    public abstract Campo[][] distribuiPecas();

    public abstract Campo[][] getRefugio();

}
```

Figura 7: Classe abstrata Tabuleiro

Cada classe filha concreta de Tabuleiro irá implementar os métodos abstratos da classe pai, no exemplo da figura 8 temos a classe filha TabuleiroBrandubh, que cria uma matriz tabuleiro e uma matriz refugio do tamanho 7, e para cada posição vai atribuindo um objeto do tipo campo, que pode ser um Refugio, Trono ou CampoNormal.

```java
package controller;

import ...8 linhas

/**...4 linhas */
public class TabuleiroBrandubh extends Tabuleiro {
    private Campo[][] tabuleiro;
    private Campo[][] refugio;

    @Override
    public Campo[][] distribuirCampos(int normal, int rei) {
        this.tabuleiro = new Campo[7][7];
        this.refugio = new Campo[7][7];
        for (int i = 0; i < 7; i++) {
            for (int j = 0; j < 7; j++) {
                if ((i == j && (i == 0 || i == tabuleiro.length - 1))
                        || (i == 0 && j == tabuleiro.length - 1)
                        || (j == 0 && i == tabuleiro.length - 1)) {
                    this.tabuleiro[i][j] = new Refugio();
                    this.refugio[i][j] = new Refugio();
                } else {
                    if(i == j && i == tabuleiro.length / 2){
                        this.tabuleiro[i][j] = new Trono();
                        this.refugio[i][j] = new Trono();
                    }else{
                        this.tabuleiro[i][j] = new CampoNormal();
                        this.refugio[i][j] = new CampoNormal();
                    }
                }
            }
        }
        return this.tabuleiro;
    }

    @Override
    public Campo[][] distribuiPecas() {
        for (int i = 0; i < 7; i++) {
            for (int j = 0; j < 7; j++) {
                if ((j == 3 && (i == 0 || i == 1 || i == 5 || i == 6))
                        || (i == 3 && (j == 0 || j == 1 || j == 5 || j == 6))) {
                    tabuleiro[i][j] = new Preto();
                } else {
                    if (j == 3 && i == 3) {
                        this.tabuleiro[i][j] = new Rei();
                        this.tabuleiro[i][j].setImagem(new ImageIcon("img/reibrancoregufio.png"));
                    } else {
                        if ((i == 3 && (j == 2 || j == 4)) || (j == 3 && (i == 2 || i == 4))) {
                            tabuleiro[i][j] = new Branco();
                        }
                    }
                }
            }
        }
        return tabuleiro;
    }

    @Override
    public Campo[][] getRefugio() {
        return refugio;
    }
}
```

Figura 8: Classe TabuleiroBrandubh

Ainda na mesma classe temos um método que irá distribuir as peças pelo tabuleiro, as peças podem ser Rei, Branco ou Preto.

```
package controller;

/**...4 linhas */
public abstract class Factory {

    public abstract Tabuleiro createTabuleiro();

    public abstract Movimento createMovimento();

}
```

Figura 9: Classe abstrata Factory

A classe abstrata Factory (Figura 9) possui métodos abstratos para a criação do tabuleiro e para a criação do movimento, aos quais suas filhas devem implementa-los. Na figura 10, temos uma classe filha, a classe FactoryBrandubh, que implementa os métodos abstratos do pai e retorna os objetos concretos.

```
package controller;

import static controller.MovimentoOutros.getInstance;

/**...4 linhas */
public class FactoryBrandubh extends Factory{

    private Tabuleiro Brandubh;
    private Movimento movimento;

    @Override
    public Tabuleiro createTabuleiro() {
        if(Brandubh == null)
            Brandubh = new TabuleiroBrandubh();
        return Brandubh;
    }

    @Override
    public Movimento createMovimento() {
        if(movimento == null)
            movimento = getInstance();
        return movimento;
    }
}
```

Figura 10: classe FactoryBrandubh

O método getFactory() na classe ControllerTela (Figura 11) é o responsável por verificar qual Factory deve ser chamado para criar os objetos concretos, movimento e tabuleiro.

```java
public ControllerTela() {
}

@Override
public void addObservador(Observador obs) {
    observadores.add(obs);
}

public void getFactory() {
    Factory fac = null;
    switch (tamanho) {
        case 11:
            fac = new FactoryHnefatafl();
            break;
        case 7:
            fac = new FactoryBrandubh();
            break;
        case 9:
            fac = new FactoryTablut();
            break;
    }
    this.distribuir = fac.createTabuleiro();
    this.movimento = fac.createMovimento();
    this.tabuleiro = this.distribuir.distribuirCampos(pecaMove, reiMove);
    this.refugio = distribuir.getRefugio();
}
```

Figura 11: Método getFactory() da classe ControllerTela

3. Padrões Comportamentais

Os padrões comportamentais descrevem padrões de objetos ou classes e os padrões de comunicações entre eles, se preocupando com o algoritmo e a atribuição de responsabilidade entre os objetos (GAMMA; el al, 2000).

3.1. MVC/Observer

O MVC é composto por: Modelo, corresponde ao objeto de aplicação, a Visão, a tela e o Controller, é o que define a maneira como a interface do usuário reage às entradas do mesmo. O MVC separa as três camadas para aumentar a flexibilidade e a reutilização. O Observer é utilizado para quando um objeto muda de estado, todos os dependentes são notificados e atualizados automaticamente (GAMMA; el al, 2000).

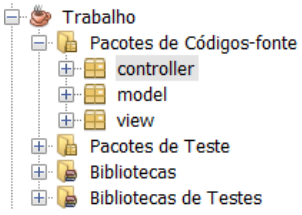


Figura 12: Padrão MVC aplicado no trabalho

O Projeto do tabuleiro foi separado em pacotes de acordo com o padrão MVC (Figura 12) e para realizar a comunicação do controller com a view, foi implementado o padrão observer, responsável por atualizar automaticamente o tabuleiro na tela de acordo com os cliques do usuário sobre a mesma (Figura 13).

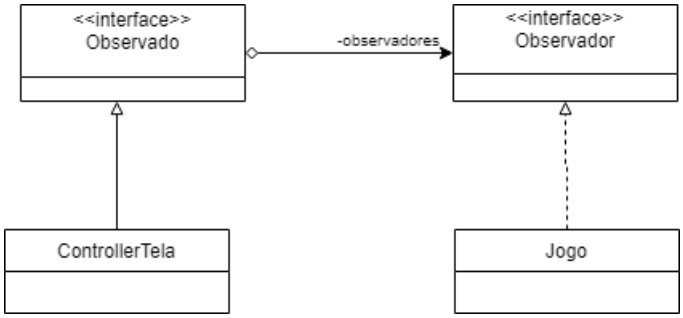


Figura 13: Padrão Observer

A interface Observado (Figura 14) possui métodos aos quais a classe ControllerTela terá que implementar, entre eles o addObservador, que será o responsável por adicionar todas as outras classes que ficaram observando o controller, neste caso é apenas a classe Jogo.

```
package controller;

import javax.swing.Icon;

/**...4 linhas */
public interface Observado {

    void addObservador(Observador obs);

    void inicializar();

    Icon getPeca(int col, int row);

    int getTamanho();

    void onMouseClicado(int linha, int coluna);

    void escolhaJogo(int tabuleiro, int movimentoNormal, int movimentoRei);

    void distribuiPecas();

    void novoJogo();

    String logMovimento();

}
```

Figura 14: Interface Observado

A interface Observador (Figura 15) possui métodos que a classe Jogo irá implementar para permitir que o controller se comunique com ele, para realizar a atualização automáticas dos dados.

```
package controller;

/**...4 linhas */
public interface Observador {

    void tamanhoTabela(int tamanho);

    void mudouTabuleiro();

    void mudouJogador(String frase);
}
```

Figura 15: Interface Observador

A classe Jogo (Figura 16) implementa os métodos da interface Observador que será responsável por mudar automaticamente os dados na tela do jogo. A classe ControllerTela (Figura 17) implementa os métodos da interface Observado que permitirá a view chama-los.

```
public class Jogo extends JFrame implements Observador {

    private int coluna;
    private int linha;
    private static final long serialVersionUID = 1L;
    private Observado controller;
    private JTable tabuleiro;

    public Jogo(int tamanho, int casaNormal, int casaRei) {...13 linhas }

    public Jogo() { }

    @Override
    public void tamanhoTabela(int tamanho) {
        this.coluna = tamanho;
        this.linha = tamanho;
    }

    @Override
    public void mudouJogador(String frase) {
        jogadorTitutlo.setText(frase);
    }

    @Override
    public void mudouTabuleiro() {
        tabuleiro.repaint();
    }

    class TableModel extends AbstractTableModel {...26 linhas }
```

Figura 16: Classe Jogo com os métodos da interface implementados

```
public class ControllerTela implements Observado {

    @Override
    public void addObservador(Observador obs) {
        observadores.add(obs);
    }

    @Override
    public void inicializar() {
        log = new Log();
        ink = new Invoker();
        jogador1 = true;
        linhaClicada = -1;
        colunaClicada = -1;
        this.tabuleiro = new Campo[tamanho][tamanho];
        this.refugio = new Campo[tamanho][tamanho];
        getFactory();
    }

    @Override
    public void distribuiPecas() {
        tabuleiro = distribuir.distribuiPecas();
        movimento.setTabuleiros(tabuleiro, refugio);
        notificarMudancaTabuleiro();
    }
    @Override
    public Icon getPeca(int col, int row) {
        return (this.tabuleiro[col][row] == null ? null : this.tabuleiro[col][row].getImagem());
    }

    @Override
    public int getTamanho() {
        return tamanho;
    }
```

Figura 17: Classe ControllerTela com alguns métodos implementados da interface Observado

3.2. Command

O padrão Command, de acordo com Gamma et al (2000), permite enfileirar ou fazer registros das solicitações de clientes bem como desfazer tais operações através do encapsulamento dessas solitações como um objeto. Dessa forma o objeto encapsulado não necessita saber como os métodos funcionam, apenas chamá-los.

Esse Padrão foi utilizado apenas para registrar os Logs de movimento das peças de cada usuário (Figura 18). Ao qual informará apenas o número da jogada, qual jogador que fez o movimento e onde a peça estava e para onde ela foi.

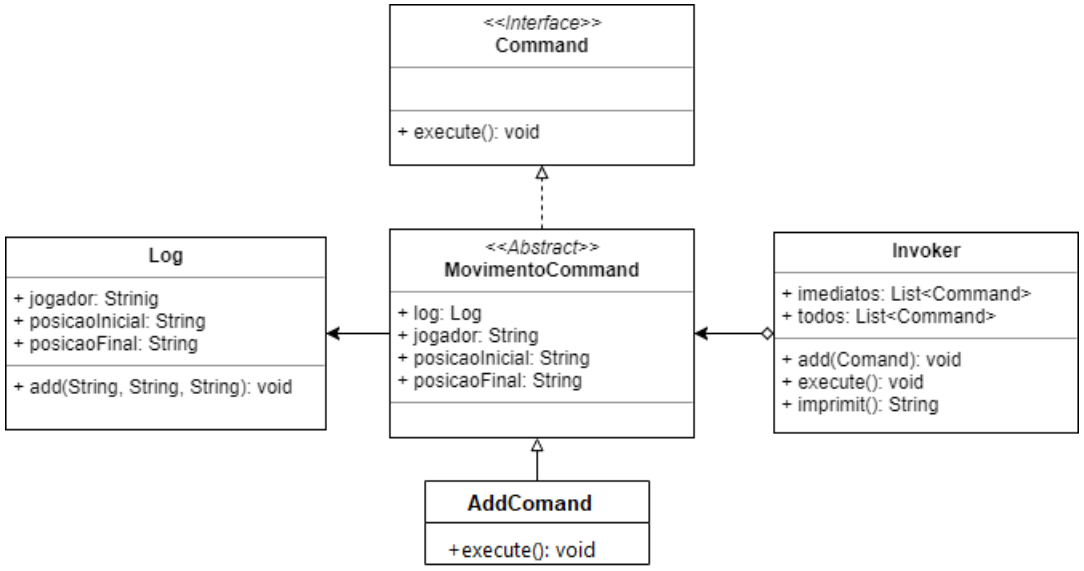


Figura 18: Padrão Command

A implementação da interface Command pode ser vista na figura 19, onde possui apenas o método executar() a ser implementado.

```
package controller;

/**...4 linhas */
public interface Command {

    void execute();
}
```

Figura 19: Interface Command

A classe abstrata MovimentoComand (Figura 20) implementa a interface Command e possui os atributos jogador, posicaoInicial e posiçãoFinal como String e um objeto log da classe Log (Figura 21). A solicitação do sistema para registrar o Log foi encapsulada através da classe AddComand (Figura 23).

```java
package controller;

import model.Log;

/**...4 linhas */
public abstract class MovimentoCommand implements Command {

    protected Log log;
    protected String jogador;
    protected String posicaoInicial;
    protected String posicaoFinal;

    public MovimentoCommand(Log log, String jogador, String posicaoInicial, String posicaoFInal){
        this.log = log;
        this.jogador = jogador;
        this.posicaoFinal = posicaoFInal;
        this.posicaoInicial = posicaoInicial;
    }
}
```

Figura 20: Classe abstrata MovimentoCommand

```java
package model;

/**...4 linhas */
public class Log  {
    private String jogador;
    private String posicaoInicial;
    private String posicaoFinal;

    public void add(String jogador, String posicaoInicial, String posicaoFinal){
        this.jogador = jogador;
        this.posicaoFinal = posicaoFinal;
        this.posicaoInicial = posicaoInicial;
    }

    public String getJogador() {...3 linhas }

    public void setJogador(String jogador) {...3 linhas }

    public String getPosicaoInicial() {...3 linhas }

    public void setPosicaoInicial(String posicaoInicial) {...3 linhas }

    public String getPosicaoFinal() {...3 linhas }

    public void setPosicaoFinal(String posicaoFinal) {...3 linhas }
    @Override
    public String toString(){
        return "Jogador: "+jogador+"\nposição inicial: "+ posicaoInicial+"\n posição final:"+posicaoFinal+"\n\n";
    }
}
```

Figura 21: Classe Log

```java
package controller;

import model.Log;

/**...4 linhas */
public class AddCommand extends MovimentoCommand{

    public AddCommand(Log log, String jogador, String posicaoInicial, String posicaoFInal){
        super(log, jogador, posicaoInicial, posicaoFInal);
    }

    @Override
    public void execute() {
        log.add(jogador, posicaoInicial, posicaoFinal);
    }

    @Override
    public String toString(){
        return "Jogador: "+jogador+"\nposição inicial: "+ posicaoInicial+"\n posição final:"+posicaoFinal+"\n\n";
    }

}
```

Figura 22: Classe AddComand

A classe invoker (Figura 23) tem duas listas com objetos Command, que será responsável por armazenar todos os logs do jogo.

```java
package controller;

import ...2 linhas

/**...4 linhas */
public class Invoker {

    private List<Command> imediatos = new ArrayList<>();
    private List<Command> todos = new ArrayList<>();

    public void add(Command comm) {
        imediatos.add(comm);
    }

    public void execute() {

        for (Command comm : imediatos) {
            comm.execute();
            todos.add(comm);
        }
        imediatos.clear();
    }

    public String imprimir() {
        String log = "Log: \n";
        int jogada = 1;
        for (Command comm : todos) {
            log += "\n Jogada número: "+jogada+"\n"+comm;
            jogada++;
        }
        return log;
    }
}
```

Figura 23: Classe Invoker

4. Padrões Estruturais

Os padrões estruturais cuidam como os objetos e as classes são compostos para formar uma estrutura maior. É utilizado herança nos padrões de classe para compor interfaces/implementações. Os padrões de objeto descrevem maneiras de construir objetos para adquirir novas funcionalidades (GAMMA; et al, 2000).

4.1. Proxy

Há quatro maneiras de aplicar o proxy segundo Gamma et al(2000), o remote proxy proporciona um local que representa um objeto num espaço de endereçamento distinto, o virtual proxy responsável pela criação de objetos caros sob demanda, o protection proxy que visa controlar a conexão de indeterminado objeto e o smart referente que substitui um simples pointer executando ações complementares no momento em que o objeto for acessado.

O Proxy foi utilizado para a aplicação de proteção, onde ele só vai deixar acontecer a jogada caso seja a vez daquele jogador (Figura 24).

Figura 24: Padrão Proxy

A classe Jogador (Figura 25) só possui o atributo jogador. A classe JogadorProxy (Figura 26) possui um construtor, que ao inicializar já define que o objeto Jogador terá

seu atributo jogador como sendo a vez do jogador 1, um método para inverter o atributo do objeto Jogador e um método para verificar se é a vez do jogador 1.

```java
package model;

/**...4 linhas */
public class Jogador {
    private String jogador;

    public String getJogador() {
        return jogador;
    }

    public void setJogador(String jogador) {
        this.jogador = jogador;
    }

}
```

Figura 25: Classe Jogador

O ControllerTela chamará o método verificarJogador1(), da classe JogadorProxy, para verificar se é a vez dos defensores (jogador 1), retornando true caso seja verdadeiro e false caso seja a vez dos mercenários (jogador 2) para assim fazer as demais verificações. Após a jogada ser feita, o método inverter() é chamado para realizar a mudança de jogador.

```java
package controller;

import model.Jogador;

/**...4 linhas */
public class JogadorProxy extends Jogador {

    public JogadorProxy() {
        super.setJogador("jogador1");
    }
    public void inverter(){
        if(super.getJogador() == "jogador1"){
            super.setJogador("jogador2");
        }else{
            super.setJogador("jogador1");
        }
    }

    public boolean verificarJogador1(){
        if(super.getJogador() == "jogador1")
            return true;
        return false;
    }
}
```

Figura 26: Classe JogadorProxy

References

GAMMA, E. et all. **Padrões de Projeto:** soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.

FREEMAN, E; FREEMAN, E. **Use a cabeça:** Padrões de Projetos. 2. ed. Rio de Janeiro: Alta Books, 2009.

SHALLOWAY, Alan; TROTT, James R.. **Explicando Padrões de Projeto:** Uma Nova Perspectiva em Projeto Orientado a Objeto. Porto Alegre: Bookman, 2004. 328 p.