Implementation of the Hnefatafl Board using Design Patterns

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. Introduction

Alexander Christopher observed many buildings, cities, streets, and any residential spaces inhabited by humans and discovered that well-constructed structures shared something in common. An example is that of porticos, which are structurally distinct but both considered high-quality. One portico might serve as a transition from the sidewalk to the entrance door, while another provides shade on a hot day. These two porticos can address a common problem in different ways. Thus, he identified that there could be similarities among high-quality designs, to which he gave the name *patterns* (Shalloway & Trott, 2004).

According to Shalloway and Trott (2004), each pattern describes a recurring problem that can be used repeatedly in different ways. As Gamma et al. (2000) state, a pattern typically contains four essential elements: the *name*, which identifies the pattern; the *problem*, explaining the situations in which it may be applied; the *solution*, which outlines the elements that comprise it, including their relationships, responsibilities, and collaborations; and the *consequence*, which mentions the advantages, disadvantages, and results of its application.

Table 1: The Design Pattern Space. Source: Gamma et al. (2000).

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter
				Template Method
	Object	Abstract Factory	Adapter (object)	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flywight	Observer
			Proxy	State
				Strategy
				Visitor

Patterns are classified according to two criteria (Table 1): *purpose*, which defines what the pattern accomplishes—whether it is creational, structural, or behavioral—and *scope*, specifying whether the pattern applies to classes or objects (Gamma et al., 2000).

2. Creational Patterns

These patterns help the system become more independent of how its objects are created, composed, and represented. A class creational pattern "uses inheritance to vary the class that is instantiated," while an object creational pattern "delegates the instantiation to another object" (Gamma et al., 2000).

2.1. Singleton

The singleton creates unique objects, ensuring that only one instance is instantiated for a given class (Freeman; Freeman, 2009) and provides a global point of access to it (Gamma et al., 2000).

```
public class MovimentoOutros extends Movimento {

    private static MovimentoOutros instance;

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

Figure 1: Singleton pattern - class MovimentoOutros

The Singleton pattern was used in the following movement classes: MovimentoOutros (Figure 1) and MovimentoTablut (Figure 2). Since the movement rules for the mercenaries differ on the Tablut board compared to the other two, an abstract class Movimento was created to implement the generic movements, which are the same across all three boards. In the derived classes (MovimentoOutros and MovimentoTablut), methods were created to differentiate the movements on each board, such as the handling for mercenaries, who in Tablut are allowed to move to the refuge.

```
public class MovimentoTablut extends Movimento {

    private static MovimentoTablut instance;

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

Figure 2: Singleton Pattern - class MovimentoTablut

In this way, to ensure that only one type of movement is created, the Singleton pattern was used. The Singleton pattern will be instantiated in the concrete classes of the Factory for each board (Figure 3).

```
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;
    }
}
```

Figure 3: Instantiation of the Singleton Pattern in one of the Singleton classes

2.2. Abstract Factory

The Abstract Factory provides an interface for creating families of related/dependent objects without specifying their concrete classes.

The Abstract Factory (Figure 4) was used to create the board and the movements based on the option the player selects. An abstract class Movimento was developed, along with two concrete classes that extend this abstract class, an abstract class Tabuleiro, and three concrete classes that extend this abstract class. Additionally, an abstract Factory class and three concrete factories that extend this abstract class were created, and these factories are responsible for creating the concrete objects for movement and the board.

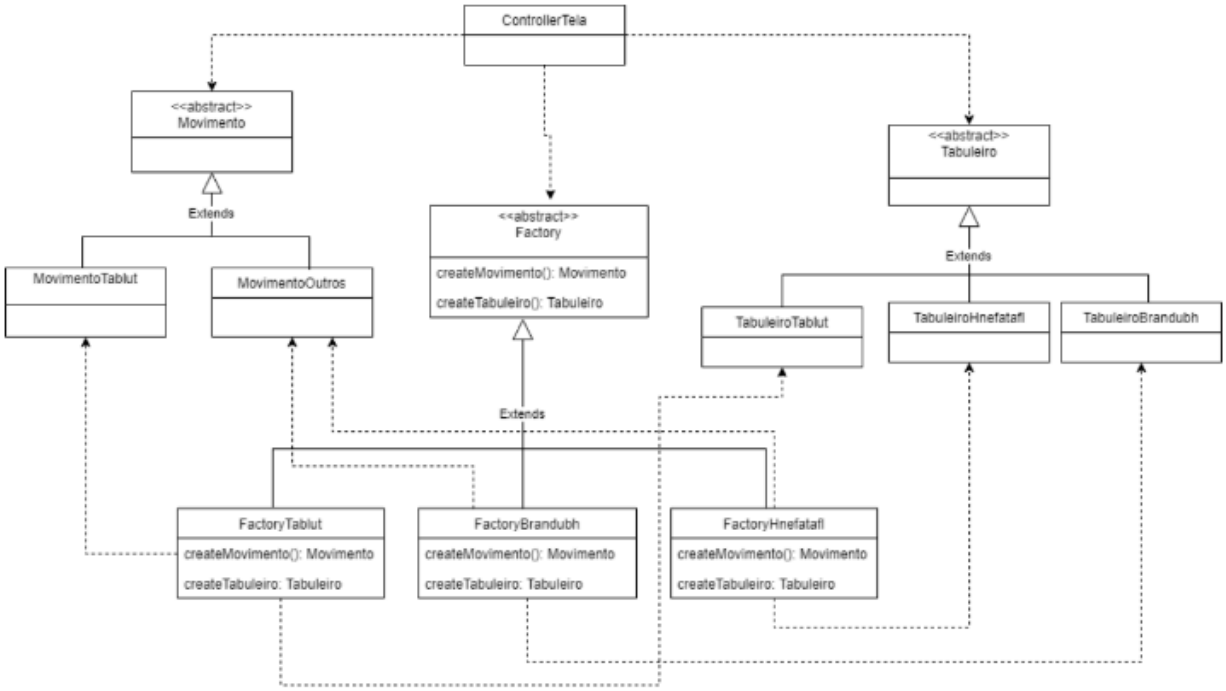

Figure 4: Abstract Factory

This pattern was used in the creation of the movements because the mercenaries can move to the refuges if the board is Tablut, thus having a behavior different from the other two boards. As for the creation of the board itself, due to their different sizes, a specific class was created for each one.

In the abstract class Movimento (Figure 5), abstract methods were created, which will be implemented in the child classes to perform the specific movements for each board, such as selecionarPreto() and deselecionarPreto(), as well as generic methods that will be the same for all boards, such as 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);

}
```

Figure 5: abstract class Movimento

In the MovimentoOutros class (Figure 6) and MovimentoTablut, the abstract methods from the Movimento class were implemented to perform the specific validations according to the chosen board.

Figure 6: class MovimentoOutros

The abstract class Tabuleiro (Figure 7) contains only abstract methods that must be implemented in its child classes. These methods are responsible for defining the size of the board and distributing each type of field and piece.

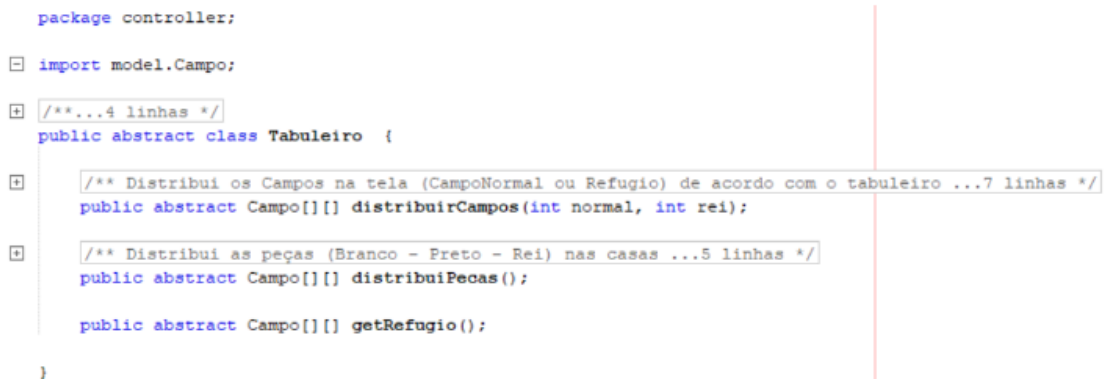

Figure 7: Abstract class Tabuleiro

Each concrete child class of Tabuleiro will implement the abstract methods from the parent class. In the example in Figure 8, we have the child class TabuleiroBrandubh, which creates a board matrix and a refuge matrix of size 7. For each position, it assigns an object of type Campo, which can be a Refugio, Trono, or 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;
    }
}
```

Figure 8: class TabuleiroBrandubh

Still in the same class, we have a method that will distribute the pieces across the board. The pieces can be Rei (King), Branco (White), or Preto (Black).

```java
package controller;

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

    public abstract Tabuleiro createTabuleiro();

    public abstract Movimento createMovimento();

}
```

Figure 9: abstract class Factory

The abstract class Factory (Figure 9) contains abstract methods for creating the board and the movement, which its child classes must implement. In Figure 10, we have a child class, FactoryBrandubh, which implements the abstract methods from the parent class and returns the concrete objects.

```
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;
    }
}
```

Figure 10: class FactoryBrandubh

The getFactory() method in the ControllerTela class (Figure 11) is responsible for determining which Factory should be called to create the concrete objects, including the movement and the board.

```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();
}
```

Figure 11: getFactory() method of the ControllerTela class

3. Behavioral Patterns

Behavioral patterns describe patterns of objects or classes and the communication patterns between them, focusing on the algorithm and the assignment of responsibilities between objects (Gamma et al., 2000).

3.1. MVC/Observer

MVC consists of: Model, which corresponds to the application object; View, which represents the interface; and Controller, which defines how the user interface responds to user input. MVC separates these three layers to increase flexibility and reusability. The Observer pattern is used when an object changes its state, and all dependent objects are notified and automatically updated (Gamma et al., 2000).

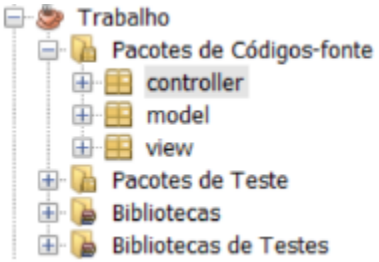

Figure 12: MVC pattern

The board design was separated into packages according to the MVC pattern (Figure 12). To enable communication between the controller and the view, the Observer pattern was implemented, which is responsible for automatically updating the board on the screen based on the user's clicks (Figure 13).

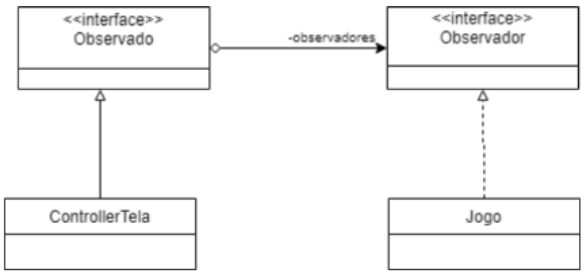

Figure 13: Observer pattern

The Observado interface (Figure 14) contains methods that the ControllerTela class must implement. Among these methods is addObservador, which is responsible for adding all other classes that will observe the controller. In this case, the only class observing is Jogo.

```java
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();

}
```

Figure 14: Observado interface

The Observador interface (Figure 15) contains methods that the Jogo class must implement to allow the controller to communicate with it, enabling the automatic updating of data.

```java
package controller;

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

    void tamanhoTabela(int tamanho);

    void mudouTabuleiro();

    void mudouJogador(String frase);
}
```

Figure 15: Observador interface

The Jogo class (Figure 16) implements the methods of the Observador interface, which are responsible for automatically updating the game data on the screen. The ControllerTela class (Figure 17) implements the methods of the Observado interface, allowing the view to call them.

```java
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 }
```

Figure 16: Jogo Class with Implemented Interface Methods

```java
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;
    }
```

Figure 17: ControllerTela Class with Implemented Methods from the Observado Interface

3.2. Command

According to Gamma et al. (2000), the Command pattern allows for queuing or recording client requests, as well as undoing operations by encapsulating these requests as objects. This way, the encapsulated object does not need to know how the methods work; it only needs to invoke them.

This pattern was used solely to record the move logs of each player's pieces (Figure 18). It records the move number, which player made the move, the starting position of the piece, and its destination.

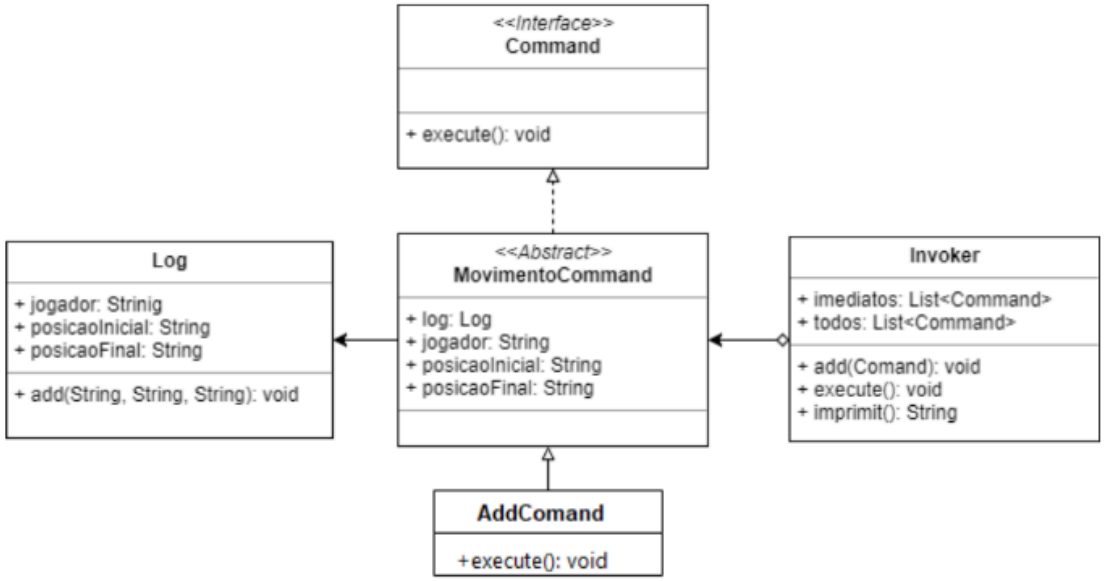

Figure 18: Command pattern

The implementation of the Command interface can be seen in Figure 19, where it contains only the executar() method to be implemented. This method encapsulates the action or command that needs to be executed, following the principles of the Command pattern.

```
package controller;

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

    void execute();
}
```

Figure 19: Command interface

The abstract class **MovimentoComand** (Figure 20) implements the **Command** interface and has attributes such as **jogador** (player), **posicaoInicial** (initial position), and **posicaoFinal** (final position) as Strings, along with a **log** object of the **Log** class (Figure 21). The system's request to record the log has been encapsulated through the **AddCommand** class (Figure 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;
    }
}
```

Figure 20: MovimentoCommand abstract class

The **MovimentoCommand** abstract class (Figure 20) serves as a foundation for encapsulating the details of a player's move, including the initial and final positions of the piece, as well as the player who made the move. By implementing the **Command** interface, this class defines the **executar()** method that will be responsible for executing the movement logic. It also holds a **log** object, which records the move, following the principles of the **Command** pattern.

This class provides a structure for concrete classes to inherit from, where specific move actions can be detailed, while ensuring consistency in how moves are recorded and executed.

```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";
    }
}
```

Figure 21: Log class

```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";
    }

}
```

Figure 22: addComand class

The **Invoker** class (Figure 23) contains two lists of **Command** objects, which are responsible for storing all the game logs.

These lists act as a queue or a collection of commands, where each movement or action performed by the players is captured as a **Command** object. The **Invoker** class then manages the execution of these commands, ensuring that they are processed and stored properly. By maintaining this separation, it adheres to the **Command** pattern, allowing for actions to be encapsulated, tracked, and potentially undone or replayed later if needed.

```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;
    }
}
```

Figure 23: Invoker class

4. Structural Patterns

Structural patterns deal with how objects and classes are composed to form a larger structure. In class patterns, inheritance is used to compose interfaces/implementations. Object patterns describe ways to build objects to acquire new functionalities (GAMMA et al., 2000).

4.1. Proxy

There are four ways to apply the proxy according to Gamma et al. (2000): the remote proxy provides a location that represents an object in a different address space; the virtual proxy is responsible for creating costly objects on demand; the protection proxy aims to control the access to an indefinite object; and the smart reference proxy replaces a simple pointer by performing complementary actions when the object is accessed.

The Proxy pattern was used for protection purposes, where it will only allow the move to happen if it is the turn of the player (Figure 24).

Figure 24: Proxy Pattern

The **Jogador** class (Figure 25) only has the **jogador** attribute. The **JogadorProxy** class (Figure 26) has a constructor, which, when initialized, sets the **Jogador** object's **jogador** attribute to be player 1's turn. It also has a method to invert the object's **jogador** attribute and a method to check if it is player 1's turn.

```
package model;

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

    public String getJogador() {
        return jogador;
    }

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

}
```

Figure 25: Jogador class

The **ControllerTela** will call the **verificarJogador1()** method from the **JogadorProxy** class to check if it is the defenders' turn (player 1). It will return true if it is, and false if it is the

mercenaries' turn (player 2), so the other checks can be made. After the move is made, the **inverter()** method is called to switch the player.

```
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;
    }
}
```

Figure 26: JogadorProxy class

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.