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 Visitor pattern add new operations to existent object structures and it was used to create small reports about the match. The Strategy pattern enables the selection of an algorithm at runtime as to which a family of algorithms to use and was used to do the board pieces movements chosen by the user. The Decorator pattern functionality allow to be added to an object and was used to add functionality to the board pieces. The State pattern allow an object to alter its behavior when its internal state changes and was used to change the state of the game to White when the defenders win, Black when the mercenaries wino r game over when one player win.*

Resumo. *Os Padrões de projeto descrevem problemas que sempre ocorrem, podendo serem utilizados repetidamente sem que se faça duas vezes iguais. O Visitor realiza operações em estruturas de objetos e foi utilizado para elaborar pequenos relatórios sobre a partida. O Strategy defini uma família de algoritmo tornando-os intercambiáveis, sendo utilizado para realizar a movimentação das peças de acordo com a opções escolhida pelo usuário. O Decorator é utilizado para acrescentar funcionalidade a mais no objeto sendo utilizado para acrescentar funcionalidades nas peças. O State permite a um objeto mudar de comportamento quando seu estado interno muda e foi utilizado para mudar o estado do tabuleiro para branco, quando é a vez dos defensores, preto, quando é a vez dos mercenários e game over para quando um dos jogadores vence.*

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 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).

2.1. Visitor

O visitor, de acordo com Gamma et al (2000), realiza uma operação nos elementos de uma estrutura de objetos sem mudas as classes desses elementos.

No jogo, esse padrão foi utilizado para construir relatórios sobre a partida (Figura 1).

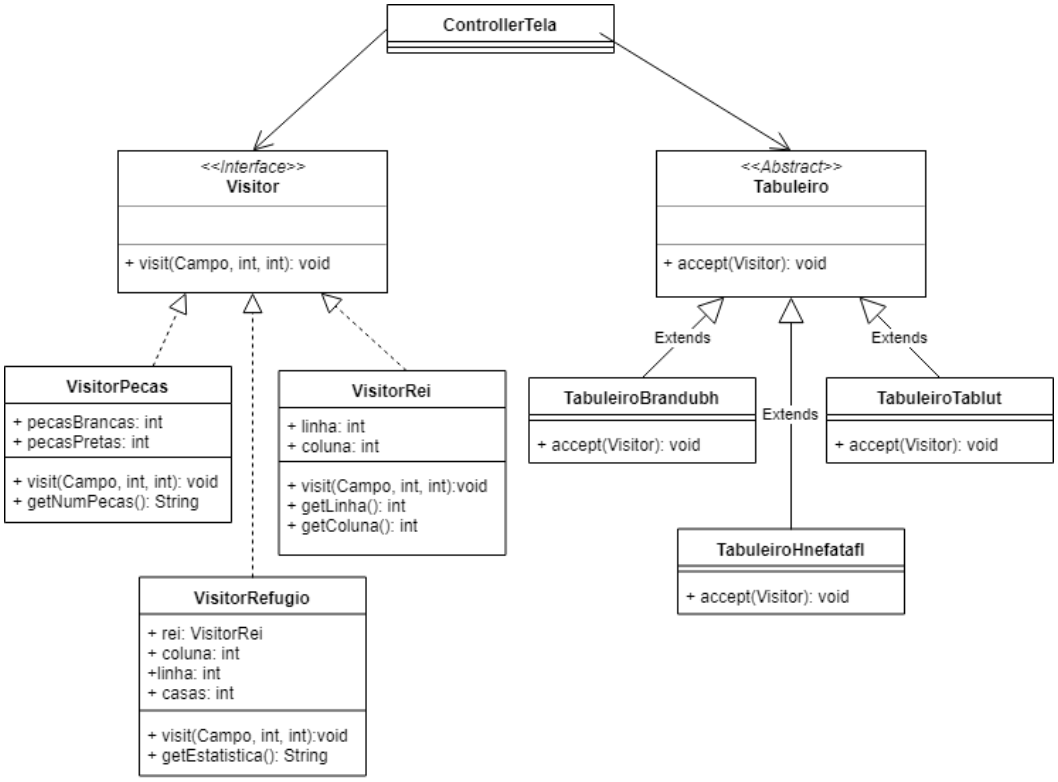

Figura 1: Diagrama do padrão Visitor utilizado no jogo

A interface Visitor (Figura 2) possui um método visit ao qual deve ser passado como parâmetro o Campo, a coluna e a linha ao qual esse campo se encontra no tabuleiro. As classes Concretas VisitorPecas (Figura 3), VisitorRei (Figura 4) e VisitorRefugio (Figura 5) implementas a interface Visitor.

```
package controller;

import model.Campo;

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

    void visit(Campo campo, int coluna, int linha);
}
```

Figura 2: Interface Visitor

A classe VisitorPecas possui dois atributos inteiros: pecasBrancas e pecasPretas, que serão a contagem das peças de cada jogador para serem utilizados na elaboração do relatório que dirá quantas peças cada jogador possui.

```java
package controller;

import model.Campo;

/**...4 linhas */
public class VisitorPecas implements Visitor {

    public int pecasBrancas = 0;
    public int pecasPretas = 0;

    @Override
    public void visit(Campo campo, int coluna, int linha) {
        if( campo.verificaBranc() || campo.verificarRei()){
            pecasBrancas++;
        }else{
            if(campo.verificarPret()){
                pecasPretas++;
            }
        }
    }

    public String getNumPecas(){
        return " Defensores possuem "+ pecasBrancas+" peças\n Mercenários possuem "+ pecasPretas+" peças";
    }

}
```

Figura 3: Classe VisitorPecas

A classe VisitorRei vai verificar qual campo está o rei e guardará nas variáveis inteiras: linha e coluna. Esse visitor será chamado no VisitorRefugio para encontrar qual o refúgio que está mais perto do rei, em caso de empate, salvará apenas o primeiro campo que passar pela classe.

```java
package controller;

import model.Campo;

/**...4 linhas */
public class VisitorRei implements Visitor{

    private int linha;
    private int coluna;

    @Override
    public void visit(Campo campo, int coluna, int linha) {
        if(campo.verificarRei()){
            this.linha = linha;
            this.coluna = coluna;
        }
    }

    public int getLinha() {
        return linha;
    }

    public int getColuna() {
        return coluna;
    }
}
```

Figura 4: VisitorRei

```java
package controller;

import model.Campo;

/**...4 linhas */
public class VisitorRefugio implements Visitor{

    private VisitorRei rei;
    private int coluna;
    private int linha ;
    private int casas = 999;

    public VisitorRefugio(VisitorRei rei){
        this.rei = rei;
    }


    @Override
    public void visit(Campo campo, int coluna, int linha) {
        if(campo.verificaRefugio()){
            int casaColuna =  Math.abs(rei.getColuna() - coluna);
            int casaLinha = Math.abs(rei.getLinha() - linha);
            if((casaColuna+ casaLinha) < casas){
                this.coluna = coluna;
                this.linha = linha;
                this.casas = (casaColuna + casaLinha);
            }
        }
    }

    public String getEstatistica(){
        return "O Refúgio mais perto está na linha "+ (linha+1)+" e coluna "+(coluna+1)+ " há "+casas+" casa(s) de distância do rei.";
    }

}
```

Figura 5: Classe VisitorRefugio

Ao clicar nos botões 'nº peças' e 'Rei e Refúgio', serão chamados os métodos 'numPecas()' e 'estatisticaRei()' respectivamente (Figura 6), ao qual o método é criado e passado como parâmetro para o método accept do tabuleiro e é retornado o método desejado.

```java
@Override
public String numPecas() {
    VisitorPecas visitor = new VisitorPecas();
    this.distribuir.accept(visitor);
    return visitor.getNumPecas();
}

@Override
public String estatisticaRei() {
    VisitorRei visitor = new VisitorRei();
    this.distribuir.accept(visitor);
    VisitorRefugio visitorR = new VisitorRefugio(visitor);
    this.distribuir.accept(visitorR);
    return visitorR.getEstatistica();
}
```

Figura 6: Métodos numPecas() e estatisticaRei() dentro da classe ControllerTela.

2.2. Strategy

Segundo Gamma et al (2000), o Strategy irá descrever e encapsular uma família de algoritmos, tornando-as intercambiáveis e que permita que o algoritmo varie independentemente do cliente.

Esse padrão foi utilizado para realizar a movimentação das peças de acordo com a opção escolhida do usuário, onde o Rei pode mover 1, 4 ou quantas casas quiser e as peças normais (brancas e pretas) podem mover 1 ou quantas casas quiserem (Figura 7).

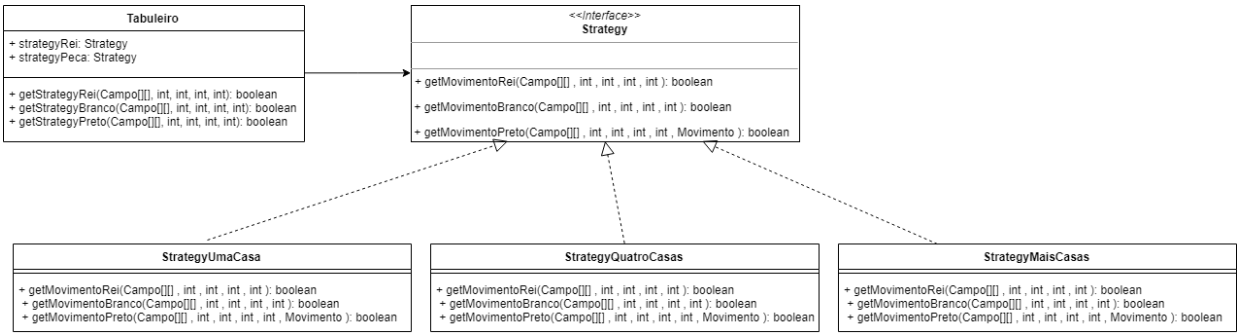

Figura 7: Diagrama do padrão Strategy utilizado no jogo.

A interface Strategy (Figura 8) possui os métodos ‘getMovimentoRei’, ‘getMovimentoBranco’ e ‘getMovimentoPreto’ ao qual são passados o tabuleiro, a coluna e linha que foram clicados e a linha e a coluna da peça clicada anteriormente.

```
package controller;

import model.Campo;

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

    boolean getMovimentoRei(Campo[][] tabuleiro, int coluna, int linha, int colunaClicada, int linhaClicada);

    boolean getMovimentoBranco(Campo[][] tabuleiro, int coluna, int linha, int colunaClicada, int linhaClicada);

    boolean getMovimentoPreto(Campo[][] tabuleiro, int coluna, int linha, int colunaClicada, int linhaClicada, Movimento movimento);
}
```

Figura 8: Interface Strategy

As classes strategyMaisCasas (Figura 9), strategyQuatroCasas (Figura 10) e strategyUmaCasa (Figura 11) implementam a interface Strategy e sobrescrevem os métodos citados anteriormente ao qual retornaram true se a peça pode ser movimentada de acordo com as regras ou falses caso tenha algo que impeça a movimentação.

```
package controller;

import model.Campo;

/**...4 linhas */
public class StrategyMaisCasas implements Strategy {

    @Override
    public boolean getMovimentoBranco(Campo[][] tabuleiro, int coluna, int linha, int colunaClicada, int linhaClicada) {...32 linhas }

    @Override
    public boolean getMovimentoRei(Campo[][] tabuleiro, int coluna, int linha, int colunaClicada, int linhaClicada) {...34 linhas }


    @Override
    public boolean getMovimentoPreto(Campo[][] tabuleiro, int coluna, int linha, int colunaAnterior, int linhaAnterior, Movimento movimento) {...28 linhas }

    /** Verifica se moveu certo - em linha ou em coluna ...9 linhas */
    public boolean moveuEmLinhaOuColuna(int coluna, int linha, int colunaClicada, int linhaClicada) {...7 linhas }

    /** Verifica se moveu em linha ...6 linhas */
    public boolean moveuEmLinha(int linha, int linhaClicada) {
        if (linha == linhaClicada) {
            return true;
        }
        return false;
    }

    /** Verifica se possui alguma peça entre tais casas na coluna movel -- linha ...11 linhas */
    public boolean verificaColunaMovel(int menor, int maior, boolean pEsquerda, int linhaClicada, boolean rei, Campo[][] tabuleiro) {...33 linhas }

    /** Verifica se possui alguma peça entre tais casas na linha movel -- coluna ...11 linhas */
    public boolean verificaLinhaMovel(int menor, int maior, boolean pCima, int colunaClicada, boolean rei, Campo[][] tabuleiro) {...33 linhas }

}
```

Figura 9: Classe StrategyMaisCasas

Figura 10: Classe StrategyQuatroCasas

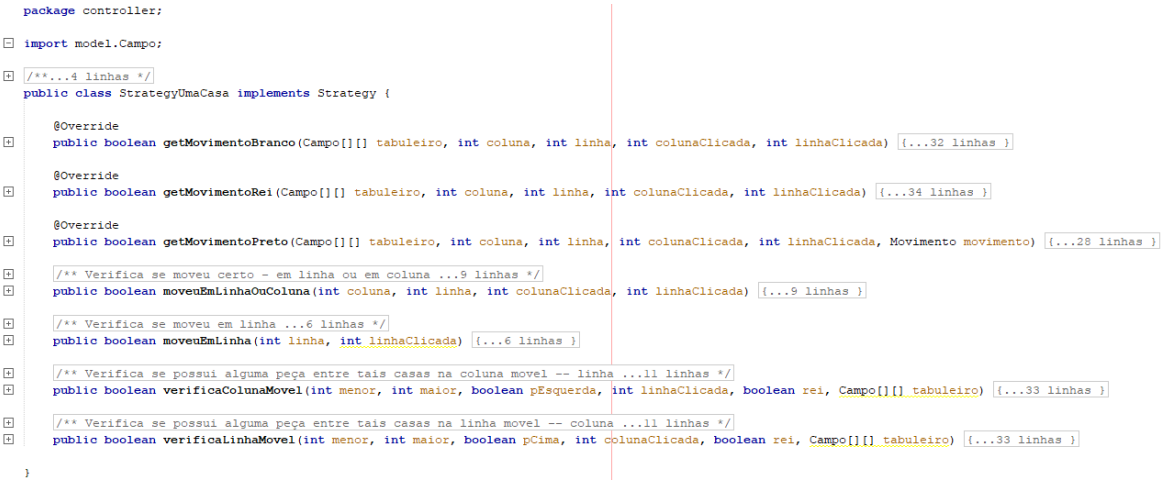

Figura 11: Classe StrategyUmaCasa

No construtor da classe abstrata tabuleiro (Figura 12) tem como parâmetros duas interfaces Strategy que serão os movimentos escolhidos para o rei e para as peças.

Figura 12: Classe abstrata Tabuleiro

Ao jogador escolher a opção desejada de movimento, serão salvas nas variáveis ‘reiMove’ e ‘pecaMove’ que então serão utilizadas para instanciar a Strategy concreta dentro da interface Strategy para então ser passados como parâmetros na criação do tabuleiro “fac.createTabuleiro(rei, peca)” (Figura 13).

```
}
Strategy rei = null;
switch (reiMove) {
    case 1:
        rei = new StrategyUmaCasa();
        break;
    case 2:
        rei = new StrategyMaisCasas();
        break;
    case 4:
        rei = new StrategyQuatroCasas();
        break;
}

Strategy peca = null;
switch (pecaMove) {
    case 1:
        peca = new StrategyUmaCasa();
        break;
    case 2:
        peca = new StrategyMaisCasas();
        break;
}
this.distribuir = fac.createTabuleiro(rei, peca);
this.movimento = fac.createMovimento();
director = new Director(distribuir);
director.construir();
```

Figura 13: Parte do método ‘getFactory()’ da classe ControllerTela

2.2. State

O State irá permitir a um objeto, quando seu estado interno muda, alterar seu comportamento (GAMMA; et al, 2000), sendo utilizado no trabalho para mudar o estado do jogo para ‘EstadoBranco’, quando é a vez do jogador ‘defensor’ jogar, ‘EstadoPreto’ quando é a vez dos ‘mercenários’ e ‘EstadoGameOver’ quando um dos jogadores vencer (Figura 14).

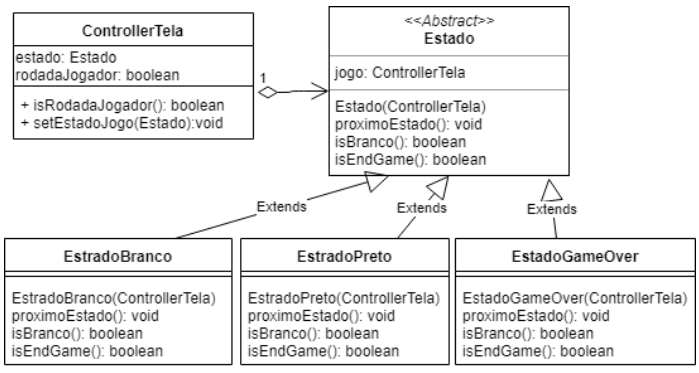

Figura 14: Diagrama do padrão State aplicado no jogo.

A classe abstrata Estado (Figura 15) possui um atributo ControllerTela que é passado no construtor da classe. Possui os métodos abstratos que serão implementados pelas classes filhas.

```java
package controller;

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

    protected ControllerTela jogo;

    public Estado(ControllerTela jogo){
        this.jogo = jogo;
    }

    abstract void proximoEstado();

    abstract boolean isBranco();

    abstract boolean isEndGame();

}
```

Figura 15: Classe abstrata Estado

A classe EstadoBranco (Figura 16) implementa os métodos abstratos onde o método ‘proximoEstado’ pode retornar o estado ‘EstadoPreto’ ou ‘EstadoGameOver’, dependendo se é a vez do jogador ou se o jogador defensor venceu.

```java
package controller;

/**...4 linhas */
public class EstadoBranco extends Estado {

    public EstadoBranco(ControllerTela jogo) {
        super(jogo);
    }

    @Override
    void proximoEstado() {
        if (this.jogo.isRodadaJogador()) {
            this.jogo.setEstadoJogo(new EstadoPreto(jogo));
        } else {
            this.jogo.setEstadoJogo(new EstadoGameOver(jogo));
        }
    }

    @Override
    boolean isBranco() {
        return true;
    }

    @Override
    boolean isEndGame() {
        return false;
    }

}
```

Figura 16: Classe EstadoBranco

```java
package controller;

/**...4 linhas */
public class EstadoPreto extends Estado {

    public EstadoPreto(ControllerTela jogo) {
        super(jogo);
    }

    @Override
    void proximoEstado() {
        if (this.jogo.isRodadaJogador()) {
            this.jogo.setEstadoJogo(new EstadoBranco(jogo));
        } else {
            this.jogo.setEstadoJogo(new EstadoGameOver(jogo));
        }
    }

    @Override
    boolean isBranco() {
        return false;
    }

    @Override
    boolean isEndGame() {
        return false;
    }

}
```

Figura 17: Classe EstadoPreto

A Classe EstadoPreto (Figura 17) poderá mudar de estado indo para o estado 'EstadoBranco' ou para 'EstadoGameOver', caso seja a vez do próximo jogador ou se os mercenários venceram o jogo, respectivamente. A classe EstadoGameOver (Figura 18) só poderá mudar de estado para 'EstadoBranco', ao iniciar um novo jogo.

```java
package controller;

/**...4 linhas */
public class EstadoGameOver  extends Estado {

    public EstadoGameOver(ControllerTela jogo) {
        super(jogo);
    }

    @Override
    void proximoEstado() {
        if(this.jogo.isRodadaJogador()){
            this.jogo.setEstadoJogo(new EstadoBranco(jogo));
        }
    }

    @Override
    boolean isBranco() {
        return false;
    }

    @Override
    boolean isEndGame() {
        return true;
    }
}
```

Figura 18: Classe EstadoGameOver

```java
public class ControllerTela implements Observado {

    private List<Observador> observadores = new ArrayList<>();
    private Campo[][] tabuleiro;
    private Campo[][] refugio;
    private int tamanho;
    private int reiMove;
    private int pecaMove;
    private int colunaClicada;
    private int linhaClicada;
    private boolean mover;
    private Tabuleiro distribuir;
    private Movimento movimento;
    private Log log;
    private Invoker ink;
    private JogadorProxy jogador;
    private static ControllerTela instance;
    Director director;
    Decorator imagem;
    private int decorou;
    private Estado estadoJogo;
    private boolean rodadaJogador;

    public ControllerTela(){
        estadoJogo = new EstadoBranco(this);
    }
    public boolean isRodadaJogador() {
        return rodadaJogador;
    }

     public void setEstadoJogo(Estado estadoJogo) {
        this.estadoJogo = estadoJogo;
    }
```

Figura 19: Classe ControllerTela com os atributos e métodos para a implementação do padrão de projetos State

A classe ControllerTela (Figura 19) possui os atributos Estado ‘estadoJogo’ e um boolean ‘rodadaJogador’ com os métodos ‘isRodadaJogador()’ que é utilizado nas classes de estado para verificar qual o próximo estado e o ‘setEstadoJogo()’ para atribuir o novo estado a variável. No construtor, a variável estado é inicializada no ‘EstadoBranco’.

```java
            if (mover && !this.estadoJogo.isEndGame()) {
                this.estadoJogo.proximoEstado();
                if (this.estadoJogo.isBranco()) {
                    notificarLabelJogador("É a vez dos Defensores");
                } else {
                    notificarLabelJogador("É a vez dos Mercenários");
                }
                mover = false;
                this.distribuir.setTabuleiro(this.tabuleiro);
            }
            notificarMudancaTabuleiro();
        }
    }
```

Figura 20: Chamada do método proximoEstado() quando um jogador move uma peça

```
public void verificarVenceuRei(int coluna, int linha) {
    if (verificarRaichiDuplo(linha, coluna)) {
        for (Observador obs : observadores) {
            obs.falarTela("Tuichi!");
            obs.desabilitarBotoes();
        }
    } else {
        if (this.refugio[coluna][linha].verificaRefugio()) {
            setRodadaJogador(false);
            this.estadoJogo.proximoEstado();
            for (Observador obs : observadores) {
                obs.falarTela("Defensor Venceu");
                obs.desabilitarBotoes();
            }
        }
    }
}

public void venceuPreto() {
    setRodadaJogador(false);
    this.estadoJogo.proximoEstado();
    for (Observador obs : observadores) {
        obs.falarTela("Mercenário Venceu");
        obs.desabilitarBotoes();
    }
}

public void olhaRei() {
    for (Observador obs : observadores) {
        obs.falarTela("Olha o Rei");
    }
}
```

Figura 21: Chamadas do método 'proximoEstado()' quando o defensor ou o mercenário vencem o jogo.

O método 'proximoEstado()' é chamado sempre que o estado muda. O estado irá mudar quando uma peça mover e for a vez do próximo jogador (Figura 20), quando um dos dois jogadores vencer (Figura 21) ou quando iniciar um novo jogo (Figura 22).

```
@Override
public void novoJogo() {
    this.tabuleiro = null;
    inicializar();
    distribuiPecas();
    notificarMudancaTabuleiro();
    notificarLabelJogador("É a vez dos Defensores");
    mover = false;
    if(!isRodadaJogador() || !this.estadoJogo.isBranco()){
        this.estadoJogo.proximoEstado();
    }
}
```

Figura 22: Chamada do método 'proximoEstado()' ao iniciar um novo jogo.

3. 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).

3.1. Decorator

O Decorator, de acordo com Gamma et al (2000), acrescenta funcionalidades a mais ao objeto, fornecendo uma alternativa ao uso de subclasses que estendem funcionalidades.

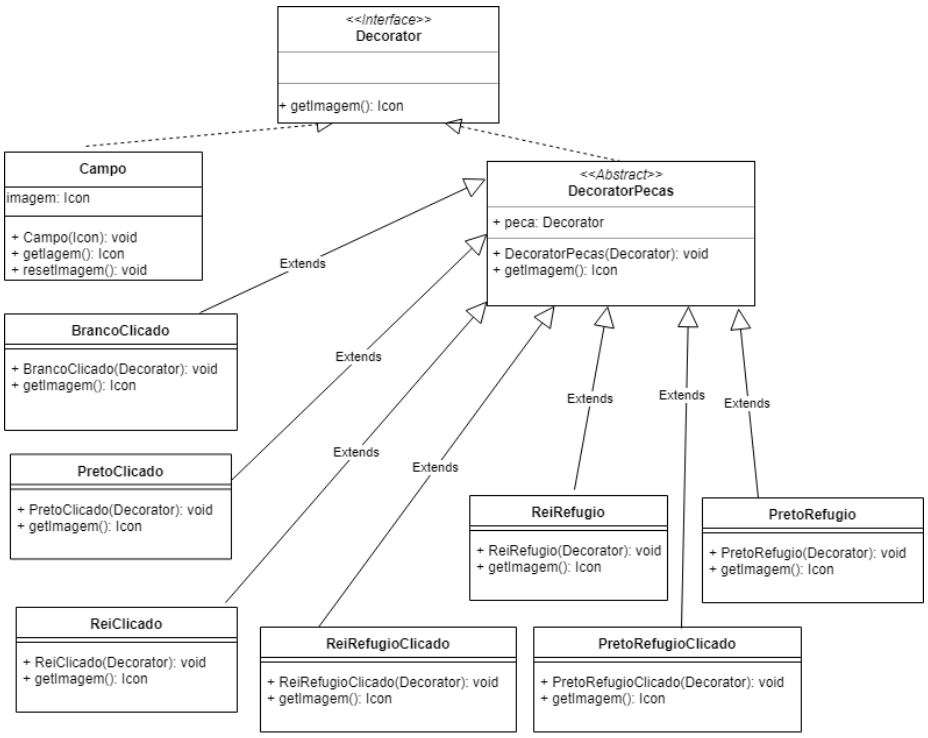

Figura 23: Diagrama do padrão Decorator utilizado no jogo

Esse padrão foi utilizado adicionar funcionalidades nas peças do tabuleiro (Figura 23). Quando uma peça branca é clicada, por exemplo, chamará a classe BrancoClicado (Figura 24) que permitirá mover a peça de lugar. Todas as classes que estendem a classe abstrata 'DecoratorPecas' possuem um método construtor onde é passado como parâmetro o Decorator e um método 'getImagem()' onde retorna uma imagem.

```
package model;

import javax.swing.Icon;
import javax.swing.ImageIcon;

/**...4 linhas */
public class BrancoClicado extends DecoratorPecas{

    public BrancoClicado(Decorator peca) {
        super(peca);
    }

    @Override
    public Icon getImagem() {
        super.setImagem(new ImageIcon("img/brancoclicado.png"));
        return  super.getImagem();
    }

}
```

Figura 24: Classe BrancoClicado.

O decorator irá ser chamado quando o usuário clicar em uma peça, o que mudará a sua imagem, mostrando que a peça pode ser movida. Ao ser movida, ela voltará para seu estado original, indicando que não foi clicado nela. Na figura 25, é mostrado uma parte do código onde acontece uma dessas mudanças.

```
    if (this.tabuleiro[coluna][linha].clicado(this, linha, coluna)) {
        Decorator clicado;
        if(this.tabuleiro[coluna][linha].verificaBranc()){
            clicado = new BrancoClicado(this.tabuleiro[coluna][linha]);
        }else{
            if(this.refugio[coluna][linha].verificarX()){
                clicado = new ReiRefugioClicado(this.tabuleiro[coluna][linha]);
            }else{
                clicado = new ReiClicado(this.tabuleiro[coluna][linha]);
            }
        }
            this.tabuleiro[coluna][linha].setImagem(clicado.getImagem());
        this.linhaClicada = linha;
        this.colunaClicada = coluna;
    }
} else {
    // deselecionar ou mover
    if (linha == linhaClicada && coluna == colunaClicada) {// deselecionar
        if(this.tabuleiro[coluna][linha].verificarRei()){
            if(this.refugio[coluna][linha].verificarX()){
                ReiRefugio reiRefugio = new ReiRefugio(this.tabuleiro[coluna][linha]);
                this.tabuleiro[coluna][linha].setImagem(reiRefugio.getImagem());
            }else{
                this.tabuleiro[coluna][linha].resetImagem();
            }
        }else{
            this.tabuleiro[coluna][linha].resetImagem();
        }
        resetClique();
```

Figura 25: Código da classe ControllerTela onde é chamado o decorator.

A classe DecoratorPecas (Figura 26) implementa a interface Decorator (Figura 27) que servira como pai para todos os decorator e possui um construtor ao qual recebe como parâmetro a interface Decorator.

```
/**...4 linhas */
public abstract class DecoratorPecas implements Decorator{
    protected Decorator peca;

    public DecoratorPecas(Decorator peca){
        this.peca = peca;
    }

    @Override
    public Icon getImagem() {
       return this.peca.getImagem();
    }

    @Override
    public void setImagem(Icon imagem) {
        this.peca.setImagem(imagem);
    }
}
```

Figura 26: Classe DecoratorPecas

```
/**...4 linhas */
public interface Decorator {

    Icon getImagem();

    void setImagem(Icon imagem);

}
```

Figura 27: Interface Decorator

References

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

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