Aplicação de padrões de projeto

Willeson Thomas da Silva[1]

Departamento de Engenharia de Software – Universidade do Estado de Santa Catarina

UDESC

will.thomassilva@gmail.com

Abstract. *Design standards are the procedures for maintenance of software and software maintenance. In this article, patterns are presented that create a robust and reliable software development environment and are used in work 1 for the development of the "Dou Shou Qi" game.*

Keywords. *Design Patterns, Software, Development.*

Resumo. *Os padrões de projeto são conceitos aplicados para deixar fácil o desenvolvimento e manutenção de softwares. Neste artigo, é apresentado padrões que criam um ambiente robusto e confiável para o desenvolvimento de software e que foram usados no trabalho 2, para o desenvolvimento do jogo "Dou Shou Qi".*

Palavras-chave. *Padrões de Projeto, Software, Desenvolvimento.*

1. Introdução

O estudo dos padrões de projeto é sugerido somente após o domínio básico de programação orientada a objetos, tendo os padrões o objetivo de ajudar a solucionar problemas que ocorrem frequentemente, tornando-se uma poderosa ferramenta para qualquer desenvolvedor de software.

Cada padrão descreve um problema que ocorre repetidamente no mundo real, desta forma, descreve a solução deste problema de um jeito que se possa reutilizar o código.

No jogo "Dou Shou Qi", foi aplicado padrões de projeto, com o intuito de tornar o código robusto e confiável, tendo o presente artigo o objetivo de explicar os motivos da aplicação dos padrões de projeto e como os mesmos foram aplicados.

O jogo "Dou Shou Qi" é dividido em várias casas com os respectivos animais alocados e divididos em times branco e preto. O tabuleiro representa uma selva com seus lago, tocas, armadilhas e grama, e as peças são os animais que se movimentam com regras especificas para cada, além de possuírem valores definidos de suas forças, cujo seu valor influencia em jogadas de ataque.

2. MVC

Para a aplicação correta do MVC no jogo, criou-se a interface *"Observador.java"* que é implementada por *"Tabuleiro.java"*, pois o padrão de projeto observer defini uma

dependência um-para-muitos entre objetos, em que todos os seus dependentes recebam notificações e se atualizem automaticamente quando os objetos mudarem seus estados.

Nesse contexto, criou-se a classe *"ControllerTabuleiro.java"* que é a classe observada, tendo como objetivo notificar *"Tabuleiro.java"* sobre mudanças de estado dos objetos.

```
public interface Observador {

    void novoJogo(DirectorCenario c1, DirectorPeca d1, DirectorPeca d2);

    void carregarJogo(ArrayList<Casa> casa);

    void mover(int posDestino, int posOrigem, Peca peca, String imgPosOrigem);

    void fimJogo(int qtdadeBranco, int qtdadePreto, String msgVitoria, String jogador) throws Exception;

    void montrarDados(String dados);

}
```

Figura 1. Interface "Observador.java".

```
private void start() {
    controller = new ControllerTabuleiro();
    controller.addObservador(this);
```

Figura 2. Método que adiciona "Tabuleiro.java", como observador em "ControllerTabuleiro.java".

3. Singleton

No desenvolvimento do jogo, foi aplicado o padrão de projeto singleton, na classe *"Tabuleiro.java"*, pois esta possui muitos recursos que se faz necessário para o completo funcionamento do jogo, não sendo necessário, portanto criar referências, tendo o singleton por objetivo unificar o acesso a esses recursos.

```
private Tabuleiro() {
    Iniciar();
}

private static Tabuleiro instance;

public synchronized static Tabuleiro getInstance() {
    if (instance == null) {
        instance = new Tabuleiro();
    }

    return instance;
}
```

Figura 3. Padrão de projeto singleton na classe "Tabuleiro,java".

```
public static void main(String args[]) {
    Tabuleiro tabuleiro = Tabuleiro.getInstance();
}
```

Figura 4. Aplicação do padrão de projeto singleton.

4. Abstract Factory e Builder

Foi implementado o padrão abstract factory em conjunto com o builder, para a construção dos elementos do jogo. O padrão de projeto abstract factory foi aplicado para a criação de famílias, tanto de animais quanto dos elementos do cenário, pelo fato de possuírem, nos 2 casos, objetos relacionados. No caso dos animais, tem-se algumas versões do jogo que possuem porco e macaco como peça.

Assim, criou-se as classes concretas *"PecaConcretFactory.java"* e *"CenarioConcretFactory"* que estendem classes abstratas *"PecaAbstractFactory.java"* e *"CenarioAbstractFactory.java"* respectivamente.

A Classe *"PecaAbstractFactory.java"* tem por objetivo fornecer uma interface de criação de famílias relacionadas aos pecas (animais), enquanto "*CenarioAbstractFactory.java*", preocupa-se com a criação de elementos do cenário como tocas, armadilhas, lago e grama, sem especificar, em ambos os casos, suas classes concretas. Já a classe *"PecaConcretFactory.java"* e *"CenarioConcretFactory.java"* são classes concretas que ficam responsáveis pela criação dos objetos de fato.

O padrão de projeto builder foi implementado em duas situações. Primeiramente viu-se a necessidade da criação de dois tipos de peca, ou seja, times diferentes (branco e preto), então usou-se o mesmo processo de construção para criar representações diferentes dessas peças. Já a segunda situação refere-se à criação dos elementos do cenário, pois observou-se assim como para os animais, a necessidade da criação de diferentes representações de tocas, lagos, armadilhas e grama.

Primeiramente foi criado uma classe abstrata *"Peca.java",* a qual é estendida pelas classes pecas (leão, tigre, rato, gato, lobo, leopardo, elefante) e a classe abstrata "Cenario" que é estendida pelas classes do cenário (toca, armadilha, lago e grama).

Criou-se as classes concretas *"Player1ConcretBuilder"* e *"Player2ConcretBuilder"* que estende a classes abstrata *"PecaBuilder.java"* que tem associação com *"DirectorPeca.java".* Já a classe concreta *"CenarioConcretBuilder"* estende apenas da classe abstrata *"CenarioBuilder"* tendo está, uma associação com "DirectorCenario.java".

Assim, a classe *"ControllerTabuleiro.java",* possui o método *"montarTabuleiro"* no qual ocorre a aplicação do padrão de projeto abstract factory em conjunto com o builder.

```
public void novoJogo(ArrayList lista) {
    CenarioConcretFactory cenarioFactory = new CenarioConcretFactory();
    CenarioBuilder c = new CenarioConcretBuilder(cenarioFactory);
    DirectorCenario c1 = new DirectorCenario(c);
    PecaConcretFactory pecaFactory = new PecaConcretFactory();
    PecaBuilder p1 = new Player1ConcretBuilder(pecaFactory);
    PecaBuilder p2 = new Player2ConcretBuilder(pecaFactory);
    DirectorPeca d1 = new DirectorPeca(p1);
    DirectorPeca d2 = new DirectorPeca(p2);

    c1.construir(lista);
    d1.construir(lista.get(5).toString(), lista.get(6).toString(), lista.get(7).toString(), lista.get(8).toString(),
            lista.get(9).toString(), lista.get(10).toString(), lista.get(11).toString(), lista.get(12).toString(),
            lista.get(13).toString());

    d2.construir(lista.get(14).toString(), lista.get(15).toString(), lista.get(16).toString(), lista.get(17).toString(),
            lista.get(18).toString(), lista.get(19).toString(), lista.get(20).toString(), lista.get(21).toString(),
            lista.get(22).toString());

    for (Observador ob : obs) {
        ob.novoJogo(c1, d1, d2);
    }
}
```

Figura 5. Padrão de projeto abstract factory e builder no método "novoJogo" da classe "Tabuleiro.java".

5. Command

O padrão de projeto command foi implementado no jogo para o movimento das peças, pois observou-se a necessidade do encapsulamento de uma solicitação como objeto, criando-se, portanto, um objeto de comando, que encapsula uma solicitação para efetuar a movimentação das peças.

Nesse contexto criou-se uma interface *"Command.java"* que é implementada pela classe concreta *"Movimento.java"*.

```
public interface Command {

    public void executar(int novaPos, Map<Integer, Casa> objetos);
}
```

Figura 6. Interface "Command.java".

```java
public class Movimento implements Command {

    private Peca peca;

    public Movimento(Peca peca) {
        this.peca = peca;
    }

    @Override
    public void executar(int proxPos, Map<Integer, Casa> objetos) {
        peca.movimentar(proxPos, objetos);
    }
}
```

Figura 7. Classe "Movimento.java".

A Classe "Movimento.java" é chamada pelo método "execute" da classe *"CommandInvoker.java"*. No entanto, tendo-se em vista a aplicação do MVC, a implementação do padrão command inicia-se na classe anônima *"EventoAnimal"* que implementa *"ActionListener"*, criada em *"Tabuleiro.java"*.

```java
public class CommandInvoker {

    public void execute(int proxPos, Peca pecaOri, Map<Integer, Casa> objetos) {
        Movimento mov = new Movimento(pecaOri);
        mov.executar(proxPos, objetos);
    }
}
```

Figura 8. Classe "CommandInvoker".

Assim, no método "actionPerformed" da classe anônima "EventoAnimal", chama-se "movimentarPeca" de *"ControllerTabuleiro.java"*.

```java
public void movimentarPeca(int proxPos, Map<Integer, Casa> objetos) {
    if (proxPos != getPosAnimal()) {
        invoker = new CommandInvoker();
        invoker.execute(proxPos, getPeca(), objetos);
        if (getPeca().isVerificaMovimento() == true) {
            for (Observador ob : obs) {
                ob.mover(proxPos, getPeca().getPosicao(), getPeca(), getPeca().getImgPosAnt());
            }
            verificaFimJogo(objetos);
            alterarJogador();
            getPeca().setPosicao(proxPos);
            getPeca().setVerificaMovimento(false);
        }
    }
    setVerifica(false);

}
```

Figura 9. Chamada do método "execute" da classe "CommandInvoker" em "ControllerTabuleiro.java".

Visitor

Trata-se de um padrão de projeto comportamental que visa representar uma operação à ser executado nos elementos de uma estrutura de objetos. Assim, aplicou-se este padrão no jogo para efetuar a cálculo de peças dos jogadores tanto do time branco como do time preto, percorrendo portando as casas do tabuleiro, informando ao final da partida, o número de peças que sobraram dos respectivos times.

Diante deste cenário, foi criado uma classe *"Visitor.java"* que é implementada por *"Estatistica.java",* e a classe *"PecasTime.java"* que possui o método "*accept"*. Já na classe *"ControllerTabuleiro.java"* foi implementado o método *"verificaFimJogo",* para então executar o padrão visitor.

```
public interface Visitor {

    void visit(Casa casa) throws Exception;
}
```

Figura 10. Interface "Visitor.java".

```
public class PecasTime {

    private Map<Integer, Casa> casas;

    public PecasTime(Map<Integer, Casa> casas) {
        this.casas = casas;
    }

    public void accept(Visitor visitor) throws Exception {
        Casa casa = null;
        for (int i = 0; i < casas.size(); i++) {
            casa = casas.get(i);
            visitor.visit(casa);
        }
    }

}
```

Figura 11. Classe "PecasTime.java", que possui o método "accept".

```java
public class Estatistica implements Visitor {

    private int qtdadeBranco;
    private int qtdadePreto;

    @Override
    public void visit(Casa casa) throws Exception {
        if (casa.getPeca() != null) {
            if (casa.verificaImgCasa().substring(0, 4).equalsIgnoreCase("imgB")) {
                qtdadeBranco++;
            } else if (casa.verificaImgCasa().substring(0, 4).equalsIgnoreCase("imgP")) {
                qtdadePreto++;
            }
        }
    }

    public int getQtdadeBranco() {
        return qtdadeBranco;
    }

    public int getQtdadePreto() {
        return qtdadePreto;
    }
```

Figura 12. Classe "Estatistica.java", que implementa "Visitor.java".

```java
public void verificaFimJogo(Map<Integer, Casa> objetos) {
    try {
        PecasTime x = new PecasTime(objetos);
        Estatistica est = new Estatistica();
        x.accept(est);
        if (getPeca().isVerificaFimJogo() == true || est.getQtdadeBranco() == 0 || est.getQtdadePreto() == 0) {
            for (Observador ob : obs) {
                ob.fimJogo(est.getQtdadeBranco(), est.getQtdadePreto(),
                        jogadorAtual.mensagemVitoria(), jogadorAtual.getNome());
            }
        }
    } catch (Exception e) {
        System.err.println("Não foi possivel finalizar o jogo corretamente");
    }
}
```

Figura 13. Classe "ControllerTabuleiro.java", que possui o método "verificaFimJogo".

Strategy

O padrão de projeto strategy tem por objetivo definir uma família de algoritmos, encapsular cada uma delas e torna-las intercambiáveis. Nesse contexto, viu-se a necessidade de criar duas estratégias ao iniciar o jogo. Assim, tem-se como estratégia a permissão de o jogador iniciar um novo jogo ou carregar um jogo salvo (figura 21). Diante disso, criou-se uma interface *"StrategyMenu.java"* que é implementada pelas classes *"StrategyNovoJogo.java"* e *"StrategyCarregarJogo.java"*.

```java
public interface StrategyMenu {

    void run();
}
```

Figura 14. Interface "StrategyMenu.java".

Na classe "Tabuleiro.java", tem-se o método *"menuJogo"*, no qual será solicitado ao jogador escolher qual a estratégia que deseja utilizar para iniciar seu jogo.

```java
public void menuJogo() {
    try {
        List<StrategyMenu> ss = new ArrayList<>();
        ss.add(new StrategyNovoJogo(controller));//0
        ss.add(new StrategyCarregarJogo(controller));//1
        int index = 1;
        String opcoes = "";
        for (StrategyMenu sm : ss) {
            opcoes += "\n" + index + "-" + sm;
            index++;
        }
        int iniciar = Integer.parseInt(JOptionPane.showInputDialog(null, "O que deseja fazer: " + opcoes));
        StrategyMenu sm = ss.get(iniciar - 1);
        sm.run();

        action();
    } catch (Exception e) {
        System.out.println("Jogo encerrado.");
    }
}
```

Figura 15. Classe "Tabuleiro.java".

State

Permite que um objeto altere seu comportamento, quando seu estado interno muda. Com base nessa definição, o padrão foi aplicado na mudança de estado dos jogadores, pois estes devem efetuar jogadas alternadamente, necessitando, portanto, de mudança de estado para ativo ou inativo.

Nesse contexto, foi criado a classe *"Jogador.java"*, que possui o método *"alterarEstado"* que é responsável pela mudança de estado do objeto. Já o método *"verificaEstado"*, tem por objetivo verificar se a peça selecionada no tabuleiro corresponde ao jogador que está ativo para efetuar a jogada com a respectiva peça, pois cada jogador tem um time correspondente.

```java
public void alterarEstado(ControllerTabuleiro c) {
    jogador.verificaEstado(c);
}

public void verificarEstado(ControllerTabuleiro c, int posicao) {
    jogador.verificarPeca(c, posicao);
}
```

Figura 15. Métodos da Classe "Jogador.java".

Para a correta aplicação do padrão de projeto state, ciou-se na classe “Jogador.java” uma associação (figura 16) com a classe abstrata *“EstadoJogador.java”* que é implementada pelas classes concretas *“Ativo.java”* e *“Inativo.java”,* referindo ao estado do jogador.

Conforme as regras do jogo “Dou Shou Qi”, a primeira jogada é efetuada pelo time branco, tendo portando que determinar o estado inicial para os jogadores de acordo com seus respectivos times no construtor da classe *“Jogador.java”* (figura 16).

```
public class Jogador implements Serializable {

    private String nome;
    private String time;
    private EstadoJogador jogador;

    public Jogador(String nome, String time) {
        if (time.equalsIgnoreCase("branco")) {
            this.nome = nome;
            this.time = "imgB";
            this.jogador = new Ativo(this);
        } else if (time.equalsIgnoreCase("preto")) {
            this.nome = nome;
            this.time = "imgP";
            this.jogador = new Inativo(this);
        }

    }
```

Figura 16. Classe “Jogador.java”.

```
public abstract class EstadoJogador implements Serializable {

    protected Jogador jogador;

    public EstadoJogador(Jogador jogador) {
        this.jogador = jogador;
    }

    public abstract void verificaEstado(ControllerTabuleiro c);

    public abstract void verificarPeca(ControllerTabuleiro c,int posicao) ;

}
```

Figura 17. Classe abstrata “EstadoJogador.java”.

A mudança de jogador ocorre a cada movimento de uma peça, ocorrendo, portanto, na classe *“ControllerTabuleiro.java”.*

```
public void alterarJogador() {
    jogador01.alterarEstado(this);
    jogador02.alterarEstado(this);
}
```

Figura 18. Método da Classe "ControllerTabuleiro.java", responsável por alterar o estado dos jogadores 1 e 2.

Composite

Esse padrão de projeto tem por objetivo compor objetos em estruturas de árvore para representar hierarquia partes-todo. Nesse contexto, aplicou-se no jogo a ideia de ranking dos jogadores em que para cada partida vencida é somado 1 ponto e, posteriormente adicionando no ranking (figura 20).

Assim, criou-se uma classe abstrata *"Dados.java"*, que é herdada pelas classes *"DadosPartida.java"* e *"DadosRanking.java"*, em que está última possui em sua estrutura uma lista de objetos do tipo "Dados".

```
@Override
public void fimJogo(int qtdadeBranco, int qtdadePreto, String msgVitoria, String jogador) throws Exception {
    JOptionPane.showMessageDialog(null, msgVitoria
            + "\n_____"
            + "\nEstatistica: "
            + "\nPecas Brancas = " + qtdadeBranco + "\nPecas Pretas = " + qtdadePreto
            + "\n_____");
    controller.gravarDadosRanking(jogador, 1);
    System.exit(0);

}
```

Figura 19. Método da classe "Tabuleiro.java", responsável por finalizar o jogo.

A cada partida finalizada (figura 19), é lido se necessário um arquivo com ranking existente e iniciado um processo de gravação. Com isso, adiciona-se os dados da partida recente em um novo ranking, e adiciona este ao ranking existente, gerando, portanto, um novo arquivo devidamente atualizado, que pode ser visualizado através do menu do jogo (figura 21).

```
public void gravarDadosRanking(String jogador, int pontos) throws Exception {
    Dados partida = new DadosPartida(jogador, pontos);
    Dados ranking = new DadosRanking();
    ranking.addDados(partida);
    ranking.gerarRanking();
}
```

Figura 20. Método da classe "ControllerTabuleiro.java", responsável por aplicar o padrão de projeto composite.

Figura 21. Menu do jogo Dou Shou Qi.

Referências

SHALLOWAY. Explicando Padrões de Projeto: Uma nova perspectiva em projeto orientado a objetos. Porto Alegre (2004).

GAMMA, E. et al. Padrões de projeto: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.