UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

RELATÓRIO TRABALHO FINAL - BANCO IMOBILIÁRIO DA DISCIPLINA DE PROJETO DETALHADO DE SOFTWARE

Rafael Gonçalves Lima, 399509
Victor Lucas da Silva Monteiro, 397307

Quixadá, 05 de dezembro de 2018

1. Introdução

O trabalho de implementação do Banco Imobiliário, proposto pela professora Paulyne Matthews Jucá, fora realizado conforme os requisitos elicitados, atendendo todas as exigências presentes no documento fornecido no dia 22 (vinte e dois) de novembro de 2018. Para correta execução do projeto, houve necessidade de adotar parte dos Padrões de Projeto GoF, uma porção destes apresentados ou comentados em sala. As respectivas explicações e motivações de escolha para cada padrão, presente no diagrama de classes, estão presentes nos tópicos abaixo.

2. Padrões de projeto

2.1 Singleton

Para garantir a existência de apenas uma instância das classes **ControlBancoImobiliario**, **ControlSorteOuReves**, **FactoryCampoEspecial** e **FactoryCartas**, mantendo um ponto de acesso global e único ao objeto, fora usado o padrão de projeto Singleton. A implementação deste padrão fora realizado da seguinte forma:

ControlBancoImobiliario

```
private static ControlBancoImobiliario controlBancoImobiliario = null;

public static synchronized ControlBancoImobiliario getInstance(){
    if(controlBancoImobiliario == null) controlBancoImobiliario = new ControlBancoImobiliario();
    return controlBancoImobiliario;
}
```

ControlSorteOuReves

```
private static ControlSorteOuReves controlSorteOuReves = null;

public static synchronized ControlSorteOuReves getInstance() {
    if(controlSorteOuReves == null) controlSorteOuReves =  new ControlSorteOuReves();
    return controlSorteOuReves;
}
```

FactoryCartas

```
private static FactoryCartas factoryCartas = new FactoryCartas();

private FactoryCartas() {}

public static synchronized FactoryCartas getInstance() {
    if(factoryCartas == null) factoryCartas =  new FactoryCartas();
    return factoryCartas;
}
```

FactoryCampoEspecial

```
private static FactoryCampoEspecial factoryCampoEspecial = new FactoryCampoEspecial();

private FactoryCampoEspecial() {}

public static synchronized FactoryCampoEspecial getInstance() {
    if(factoryCampoEspecial == null) factoryCampoEspecial = new FactoryCampoEspecial();
    return factoryCampoEspecial;
}
```

2.2 Iterator

Para encapsular a iteração realizada nas cartas de sorte ou revés, obtendo conhecimento da primeira carta ou da próxima a ser chamada, fora implementado o padrão Iterator na classe **ControlSorteOuReves**. A implementação deste padrão depende que a classe implemente a interface **Iterator**, realizando sobrescrita de métodos presentes nesta classe, pré-determinados na criação deste último.

Iterator

```
public interface Iterator {
    Object first();
    Object next();
    boolean isDone();
}
```

ControlSorteOuReves

```
public class ControlSorteOuReves implements Iterator {
```

```java
@Override
public Object first() { return this.cartas.get(0); }

@Override
public Object next() {
    if (cartas.isEmpty()) {
        this.criarCartas();
    }
    Random random = new Random();
    int i = random.nextInt(this.cartas.size());
    return this.cartas.remove(i);
}

@Override
public boolean isDone() { return false; }
```

2.3 Observer

Para saber quando o jogador vai a falência tivemos de implementar o padrão de projeto observer, que escutará quando uma determinada pessoa falir e irá tomar ações para que o mesmo (usuário) seja notificado disto. Para implementar um observer as interfaces **ObserverJogador** e **SubjectObserver** foram construidas e elas são implementadas por **JogadorHumano** (que é o sujeito observado) e **ControlBancoImobiliario** (que é o observador).

ObserverJogador

```java
public interface ObserverJogador {
    void update(JogadorHumano jogador, float valor);
}
```

SubjectObserver

```java
public interface SubjectObserver {
    void addObserver(ObserverJogador o);
    void removeObserver(ObserverJogador o);
    void notifyObserver(float valor);
}
```

JogadorHumano

```
public class JogadorHumano extends Jogador implements SubjectObserver {
```

```
@Override
public void addObserver(ObserverJogador o) {
    this.observerCollection.add(o);
    System.out.println(this.observerCollection.size());
}

@Override
public void removeObserver(ObserverJogador o) { this.observerCollection.remove(o); }

@Override
public void notifyObserver(float valor) {
    for (ObserverJogador object: this.observerCollection) {
        object.update( jogador: this, valor);
    }
}
```

ControlBancoImobiliario

```
public class ControlBancoImobiliario implements ObserverJogador {
```

```
@Override
public void update(JogadorHumano jogador, float valor) {
    System.out.println(jogador.getNome()+" está devendo R$ "+valor);
    float propriedadesVendidas = 0;

    if (jogador.getPropriedades().isEmpty()) {
        JOptionPane.showMessageDialog( parentComponent: null, message: jogador.getNome() + " foi à falência (Sem dinheiro para pagar os credores).");
        this.jogadoresAtivos.remove(jogador.getId());
        jogador.getPeca().obterLocalizacao().removerJogador(jogador);
    } else {
        for (Propriedade propriedade: jogador.getPropriedades()) {
            if (propriedade instanceof Terreno) {
                if (((Terreno) propriedade).isHasHotel()) {
                    propriedadesVendidas += ((4 * ((Terreno) propriedade).getPrecoCasa() + ((Terreno) propriedade).getPrecoHotel())/2);
                    propriedadesVendidas += propriedade.getPreco();
                } else {
                    propriedadesVendidas += ((((Terreno) propriedade).getNumCasas() * ((Terreno) propriedade).getPrecoCasa())/2);
                    propriedadesVendidas += propriedade.getPreco();
                    System.out.println("Sem Hotel: "+propriedadesVendidas);
                }
            } else {
                propriedadesVendidas += propriedade.getPreco();
            }
            JOptionPane.showMessageDialog( parentComponent: null, message: jogador.getNome()+" vendeu "+propriedade.getNome()+" por "+propriedadesVendidas+" para pagar suas divi
            if (propriedadesVendidas >= valor) {
                Banco.getInstance().pagar(propriedadesVendidas);
                jogador.receber(propriedadesVendidas);
                JOptionPane.showMessageDialog( parentComponent: null, message: jogador.getNome()+" pagou sua divida e continua no jogo!");
                break;
            } else {
                JOptionPane.showMessageDialog( parentComponent: null, message: "Dinheiro Insuficiente Para Pagar O Credor");
            }
        }
    }
```

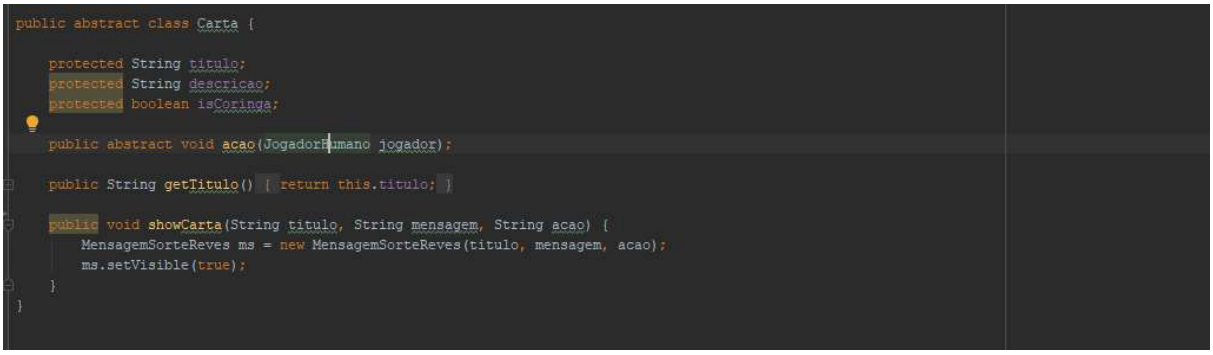
```
        if (this.jogadoresAtivos.size() == 1) {
            JOptionPane.showMessageDialog( parentComponent: null, message: "<Alguém> ganhou essa partida");
            TelaPrincipal.getInstance().closeJanela();
        }
    }
}
```

2.4 Strategy

Para executar comportamentos diferentes relacionados a ação de cada carta, tivemos que implementar o padrão strategy. Para implementar esse padrão utilizamos de uma classe abstrata **Carta** que é estendida por **CartaAposta** e **CartaSaiaDaPrisão**. Conforme pode ser observado abaixo o método **acao(…)** é implementado de forma

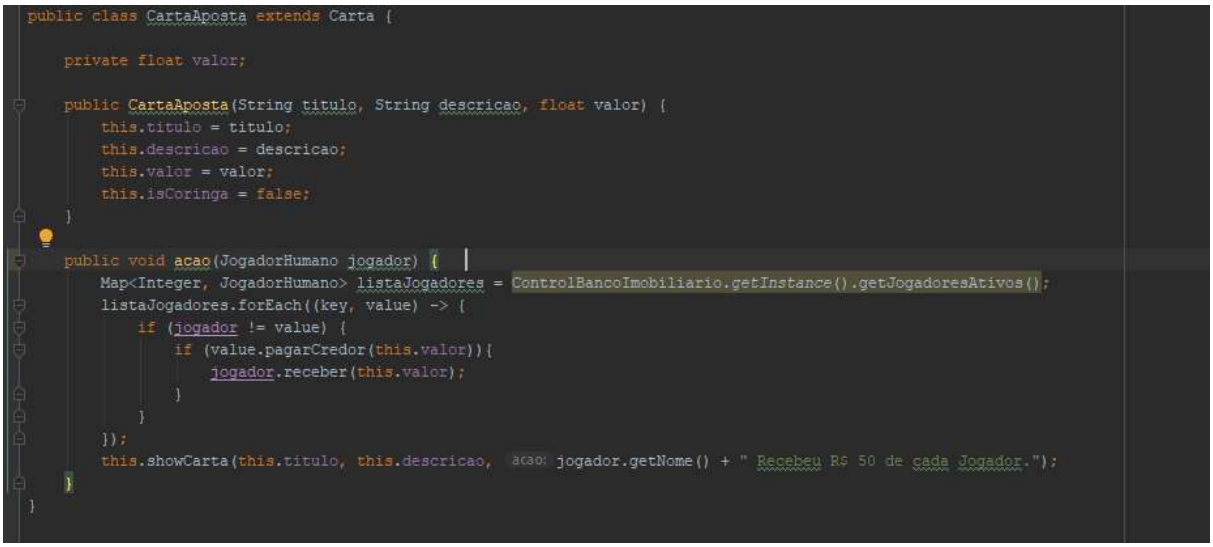
diferente em cada uma das classes que herdam de **Carta**. Pode ser observada cada classe implementada abaixo:

Carta



```
public abstract class Carta {

    protected String titulo;
    protected String descricao;
    protected boolean isCoringa;

    public abstract void acao(JogadorHumano jogador);

    public String getTitulo() { return this.titulo; }

    public void showCarta(String titulo, String mensagem, String acao) {
        MensagemSorteReves ms = new MensagemSorteReves(titulo, mensagem, acao);
        ms.setVisible(true);
    }
}
```

CartaAposta



```
public class CartaAposta extends Carta {

    private float valor;

    public CartaAposta(String titulo, String descricao, float valor) {
        this.titulo = titulo;
        this.descricao = descricao;
        this.valor = valor;
        this.isCoringa = false;
    }

    public void acao(JogadorHumano jogador) {
        Map<Integer, JogadorHumano> listaJogadores = ControlBancoImobiliario.getInstance().getJogadoresAtivos();
        listaJogadores.forEach((key, value) -> {
            if (jogador != value) {
                if (value.pagarCredor(this.valor)){
                    jogador.receber(this.valor);
                }
            }
        });
        this.showCarta(this.titulo, this.descricao,  acao: jogador.getNome() + " Recebeu R$ 50 de cada Jogador.");
    }
}
```

CartaSaiaDaPrisao

```java
public class CartaSaiaDaPrisao extends Carta {

    public CartaSaiaDaPrisao(String titulo, String descricao) {
        this.titulo = titulo;
        this.descricao = descricao;
        this.isCoringa = true;
    }

    public void acao(JogadorHumano jogador) {
        if(ControlBancoImobiliario.getInstance().getJogadoresPresos().containsValue(jogador)) {
            ControlBancoImobiliario.getInstance().soltarJogador(jogador);
            this.showCarta(this.titulo, this.descricao,  acao: jogador.getNome() + " foi solto da Prisão.");
        } else {
            jogador.setCartaPrisao(this);
            this.showCarta(this.titulo, this.descricao,  acao: jogador.getNome() + " ganhou: 'Saida Livre da Prisão'");
        }
    }

}
```

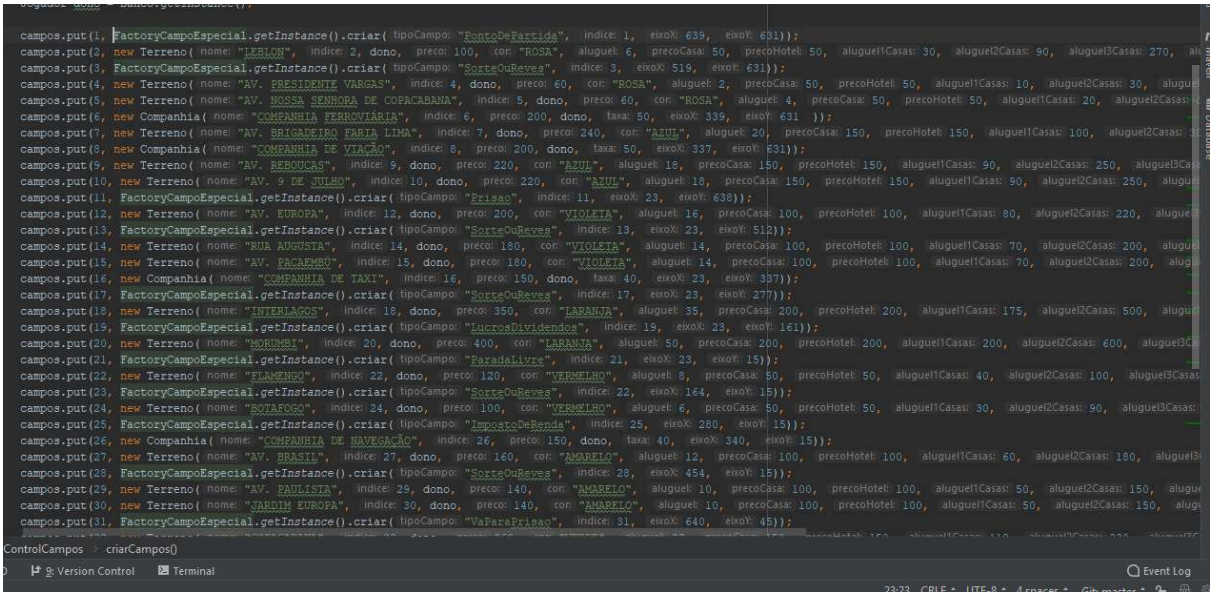
2.5 Factory

Para evitar vários casos condicionais para produção de um campo no tabuleiro criamos uma fábrica. Utilizamos duas classes para realizar a produção dos campos previamente citados, são elas: **FactoryCampoEspecial** (que contém os condicionais para campos especiais) e **ControlCampos** (que "instancia" por meio da fábrica os campos especiais). As implementações destas classes pode ser observadas abaixo:

FactoryCampoEspecial

```java
public class FactoryCampoEspecial {
    private static FactoryCampoEspecial factoryCampoEspecial = null;

    private FactoryCampoEspecial() {}

    public static synchronized FactoryCampoEspecial getInstance() {
        if(factoryCampoEspecial == null) factoryCampoEspecial = new FactoryCampoEspecial();
        return factoryCampoEspecial;
    }

    public Campo criar(String tipoCampo, int indice, int eixoX, int eixoY) {
        if (tipoCampo.equals("PontoDePartida")) {
            return new PontoDePartida(indice, eixoX, eixoY);
        } else if (tipoCampo.equals("ImpostoDeRenda")) {
            return new ImpostoDeRenda(indice, eixoX, eixoY);
        } else if (tipoCampo.equals("LucrosDividendos")) {
            return new LucrosDividendos(indice, eixoX, eixoY);
        } else if (tipoCampo.equals("ParadaLivre")) {
            return new ParadaLivre(indice, eixoX, eixoY);
        } else if (tipoCampo.equals("VaParaPrisao")) {
            return new VaParaPrisao(indice, eixoX, eixoY);
        } else if (tipoCampo.equals("Prisao")) {
            return new Prisao(indice, eixoX, eixoY);
        } else if (tipoCampo.equals("SorteOuReves")) {
            return new SorteOuReves(indice, eixoX, eixoY);
        }
        return null;
    }
}
```

ControlCampos *(apenas parte do uso, o restante pode ser observado no código)*



3. Especificação de responsabilidades

a. ControlBancoImobiliario

i. Inicia um novo jogo
ii. Pede renderização das telas de solicitação de quantidade de jogadores, nomes e cores dos jogadores e tela principal, onde o jogo ocorre
iii. Inicia uma nova rodada
iv. Mapeia e pede alteração de posição do jogador no tabuleiro
v. Mapeia e pede execução das ações realizadas nos campos
vi. Observa se algum jogador veio a falência

b. Tabuleiro

i. Pede ao *ControlCampos* para instânciar todos os campos;
ii. Devolve a quem solicitar o campo que o jogador deve avançar

c. Terreno

i. Executa o *EfeitoEspecial* referente a ele
ii. Constrói casa ou hotel
iii. Mantém informações sobre o dono
iv. Realiza ações para os visitantes do terreno *(como pagar o aluguel)*

d. Companhia

i. Executa o *EfeitoEspecial* referente a ele
ii. Mantém informações sobre o dono
iii. Realiza ações para os visitantes do terreno *(como pagar o taxa)*

e. Carta

i. Aplica o *EfeitoEspecial* ao jogador solicitado

f. Jogador

i. Compra propriedade
ii. Vende propriedade
iii. Recebe dinheiro e realiza pagamentos
iv. Lança dados
v. Guarda sua peça

g. Peca

i. Guarda um campo
ii. É renderizada pela *TelaPrincipal* conforme suas coordenadas

h. FichaCriminal

i. Controla os delitos do jogador (quantas vezes ele tirou o mesmo conjunto de faces iguais seguidos) para saber a quantas rodas o mesmo está preso

4. Conclusão

Durante as atividades de planejamento e execução pudemos identificar quais padrões poderiam ser mais adequados ao conjunto de objetos do sistema. De forma geral, tivemos algumas dúvidas sobre as responsabilidades do controlador geral do sistema e sobre a comunicação entre os objetos e as views. As implementações mais críticas foram: o gerenciamento dos jogadores presos, as validações para construção de casas e hotéis, a lógica de verificação dos jogadores falidos e finalização do jogo com um vencedor. Por fim, nossos maiores desafios foram gerenciar o tempo e as atividades acumuladas no final do semestre, estamos felizes por ter concluído a atividade. Até uma próxima oportunidade.