# RELATÓRIO FINAL — TRUCO ROGUE LIKE

## 1. Introdução

O projeto **Truco Rogue Like** consiste em um jogo digital inspirado no tradicional jogo de Truco, combinando mecânicas de cartas com elementos de jogos roguelike, como progressão por níveis, aumento gradual de dificuldade e aquisição de habilidades especiais (cartas especiais).

O objetivo principal foi desenvolver um sistema modular, extensível e organizado, utilizando conceitos de **Programação Orientada a Objetos (POO)**, tais como herança, composição, encapsulamento e padrões de projeto.

---

# 2. Arquitetura Geral do Sistema

A arquitetura do sistema foi organizada em camadas conceituais:

1. **Camada de Aplicação**
    - App
    - Jogo
2. **Camada de Lógica da Partida**
    - Partida
    - ContextoJogo
3. **Camada de Jogadores**
    - Jogador
    - Computador (herda de Jogador)
    - FabricaDeBots
4. **Camada de Cartas**
    - Carta
    - CartaEspecial

    - Baralho
    - GerenciadorCartasEspeciais

5. **Enums auxiliares**

    - Personalidade
    - NivelDificuldade

Essa organização permite separar responsabilidades e tornar o código modular e fácil de manter.

---

# 3. Descrição das Classes

## 3.1 App

Classe responsável por inicializar a aplicação.
Chama instâncias de *Jogo* e controla o ponto de entrada (**main**).

## 3.2 Jogo

Gerencia o fluxo geral do jogo, alternando entre partidas e níveis.
Interage com *FabricaDeBots* e *GerenciadorCartasEspeciais*.

## 3.3 Partida

É o coração do sistema. Responsabilidades:

- Gerenciar pontuação
- Controlar rodadas e distribuição de cartas
- Receber jogadas do jogador e do bot
- Controlar aumentos de aposta (truco, 6, 9, 12)

Mantém instâncias de:

- Jogador
- Computador
- Baralho
- ContextoJogo

### 3.4 Jogador

Define o comportamento comum:

- Receber cartas
- Jogar cartas
- Limpar mão
- Atributos como nome, mão e pontuação

### 3.5 Computador (Bot)

Herda de Jogador.
Possui atributos adicionais:

- Personalidade
- Nível de dificuldade
- Estratégias de risco/recompensa

Através de sua IA, decide:

- Qual carta jogar
- Quando pedir truco
- Reações a aumentos

### 3.6 Baralho

Responsável por:

- Criar cartas básicas
- Embaralhar
- Distribuir cartas
- Definir a manilha de cada rodada

### 3.7 Carta / CartaEspecial

**Carta:** possui naipe, valor e peso.
**CartaEspecial:** adiciona um *efeito* representado por `Consumer<ContextoJogo>` que modifica regras temporariamente:

- Aumentar pontuação
- Alterar força das cartas
- Recuperar vida / recursos (dependendo da versão)

### 3.8 GerenciadorCartasEspeciais

Responsável por:

- Registrar e armazenar cartas especiais disponíveis
- Sortear recompensas
- Garantir que efeitos sejam aplicados corretamente

### 3.9 FabricaDeBots

Gera adversários com base em:

- Personalidade (valente, medroso, calculista etc.)
- Nível de dificuldade (fácil, médio, difícil)

### 3.10 ContextoJogo

Objeto que transporta informações essenciais para efeitos das cartas especiais:

- Pontuação
- Estado da rodada
- Mão do jogador
- Mão do bot
- Peso da manilha
- Histórico de jogadas

---

# 4. Principais Decisões de Projeto

## 4.1 Uso de herança para o comportamento dos jogadores

A decisão de criar `Computador` como uma extensão de `Jogador` foi acertada, porque:

- Compartilha atributos (mão, nome)
- Permite sobrescrever estratégias de jogada

Essa arquitetura facilita adicionar outros tipos de jogadores futuramente.

## 4.2 Criação de um sistema de cartas especiais baseado em funções

A escolha de usar `Consumer<ContextoJogo>` para efeitos foi uma ótima solução:

- Evita múltiplas subclasses para cada tipo de carta especial
- Permite adicionar efeitos sem alterar o código base (Open/Closed Principle)
- Torna o sistema escalável

### 4.3 Uso de um gerenciador dedicado

`GerenciadorCartasEspeciais` concentra regras de recompensas e facilita manutenção.

### 4.4 Separação da lógica de partida e da lógica de jogo

A classe `Partida` controla apenas a rodada ativa;
A classe `Jogo` controla progresso global.
Isso evita classes gigantes e melhora organização.

### 4.5 Utilização de enums para comportamentos complexos

Enums como `Personalidade` e `NivelDificuldade` permitem:

- Modularizar estratégias
- Padronizar comportamento
- Evitar uso excessivo de números mágicos

---

# 5. Desafios Encontrados

### 5.1 Complexidade da IA do bot

Implementar personalidades com lógica diferente exigiu criar estratégias variadas de risco e reação, aumentando a complexidade da classe *Computador*.

### 5.2 Regras do Truco são extensas

- Pontuação variável
- Hierarquia de cartas
- Manilhas dinâmicas

A necessidade de representar essas regras fez surgir métodos extensos na classe *Partida*.

### 5.3 Sincronização entre cartas especiais e estado da partida

O uso de `ContextoJogo` foi essencial para manter o código limpo, mas exigiu padronização de grande parte das funções de partida.

### 5.4 Manutenção de responsividade no console

Como o jogo é textual, foi preciso separar lógica de exibição e lógica interna para evitar acoplamento.

---

# 6. Sugestões de Melhoria (Evolução futura)

### 6.1 Criar uma Interface Gráfica

O jogo pode ser expandido com:

- JavaFX
- Swing
- LibGDX

Isso tornaria a experiência mais rica.

### 6.2 Aplicar Padrão Strategy para IA

Cada personalidade poderia ser uma classe concreta implementando interface:

```
interface EstrategiaBot {
    Carta escolherJogada(ContextoJogo);
    boolean pedirTruco(ContextoJogo);
}
```

Isso deixaria o código modular e testável.

### 6.3 Implementar logs (histórico detalhado de jogadas)

Facilita debug e auditoria de comportamento da IA.

### 6.4 Criar testes automatizados (JUnit)

Cobrir:

- Peso das cartas
- Regras de manilha
- IA básica
- Cartas especiais

### 6.5 Persistência de dados

Salvar:

- Progresso do jogador
- Estatísticas
- Histórico de runs

### 6.6 Balanceamento das cartas especiais

Criar tiers (raras, lendárias) e sistemas probabilísticos.

### 6.7 Modo multiplayer local

Permitir dois jogadores alternarem no mesmo teclado.

---

# 7. Conclusão

O projeto **Truco Rogue Like** apresenta uma arquitetura sólida, bem estruturada e com forte aplicação dos princípios de POO. O uso de herança, enums, composição e padrões de projeto como "factory" elevam a qualidade do sistema. A implementação de cartas especiais usando funções (lambdas) permite grande flexibilidade.

O jogo se encontra em um estado funcional e expansível, com grande potencial para receber melhorias como IA mais complexa, interfaces gráficas e persistência de dados.