

1. Tema do jogo e regras gerais

Tema do jogo: O projeto consiste na implementação de um sistema de combate baseado no universo de *Jujutsu Kaisen*, estruturado inteiramente em Java e executado via console. O sistema simula batalhas entre dois tipos de personagens: Feiticeiros Jujutsu e Maldições, cada um possuindo atributos próprios, técnicas inatas e ações disponíveis durante a partida.

Visão geral das regras:

- Cada jogador é um Jogador do tipo: Feiticeiro / Maldição
- Cada jogador possui:
 - Nome
 - Energia amaldiçoada
 - Grau (Grau 4, Grau 3, Grau 2, Grau 1, Grau Especial)
 - Força física
 - Vida máxima e vida atual
 - Agilidade
 - Uma técnica inata (Técnica) escolhida entre 14 opções do enum `ListaTecnicas`
- O combate é em turnos:
 - Em cada turno, o jogador pode:
 - i. Socar o oponente
 - ii. Usar técnica inata
 - iii. Concentrar energia para aumentar a força física
 - iv. Regenerar vida usando energia (via interface `Regeneravel`)
 - v. Expandir domínio, se cumprir os requisitos.
- Há um sistema de esquiva baseado em agilidade para socos e técnicas:
 - Atacante: dado d10 + Agilidade do atacante
 - Defensor: dado d10 + Agilidade do defensor
 - Se o total do defensor for maior, ele desvia.
- Existe um sistema de Zona / Black Flash (`Kokusen`):
 - No soco há chance de `Kokusen` que aumenta muito o dano e dá mais pontos.
 - Ao entrar em "zona", o personagem ganha energia extra e aumenta a chance de Black Flash.
- Expansão de Domínio:
 - Só pode ser feita com vida abaixo de 30 e pelo menos 50 de energia.
 - Garante acerto das técnicas (ignora esquiva).
 - Se os dois expandirem domínio, ocorre um Choque de Domínios.
- **Choque de Domínios (Domain Clash):**
 - Sistema de melhor de 3 rodadas com dado + poder da técnica de cada jogador.

- Quem vencer 3 rodadas mantém o domínio; o outro perde energia e não consegue expandir.
- **Pontuação (Placar):**
 - Soco normal: +10 pontos
 - Black Flash (Kokusen): +100 pontos
 - Técnica inata: +20 pontos
- A partida termina quando um dos jogadores tem vida igual a 0; o placar final é exibido.

2. Classes de domínio e enums

2.1 Classes de domínio (abstração)

As classes principais de domínio são:

- Jogador (abstrata)
- Feiticeiro
- Maldição
- Técnica
- Partida
- Participacao
- Placar

Um exemplo de abstração na superclasse **Jogador** é apresentado a seguir:

```
package Jogador;

public abstract class Jogador {

    protected String Nome;

    protected int Energia;

    protected Grau Grau;

    protected int Forca;

    protected int vidaMaxima;

    protected int vidaAtual;

    protected int Agilidade;
```

```

protected Tecnica tecnica;

protected boolean zona = false;

protected int rodadasZonaRestantes;

protected boolean energiaConcentrada = false;

protected int rodadasConcentradasRestantes;

protected int forcaBase;

    public Jogador (String nome, int energia, Grau grau, int forca, int
vidaMaxima, int agilidade, Tecnica tecnica) {

        this.Nome = nome;

        this.Energia = energia;

        this.Grau = grau;

        this.Forca = forca;

        this.forcaBase = forca;

        this.vidaMaxima = vidaMaxima;

        this.vidaAtual = vidaMaxima;

        this.Agilidade = agilidade;

        this.tecnica = tecnica;

    }

    public abstract int usarTecnicaInata (Jogador alvo);

}

```

Outras responsabilidades:

- **Feiticeiro / Maldição:** especializam regras (custo de regeneração, mesma interface Regeneravel).
- **Tecnica:** encapsula nome, poder, estado de domínio, lógica de uso da técnica (**UsarTecnica**).

- **Partida:** controla rodadas, turnos, ordem de ação, expansão de domínio, choque de domínios e interage com o Placar e as Participacao.
- **Participacao:** representa a participação de um Jogador em uma Partida com danoTotal e se venceu.
- **Placar:** controla pontuação de cada jogador.

2.2 Enums de domínio

Há dois enums:

Grau - enum principal de domínio (hierarquia de poder):

```
package Jogador;

public enum Grau {

    Grau_4(4, "Grau 4"),

    Grau_3(3, "Grau 3"),

    Grau_2(2, "Grau 2"),

    Grau_1(1, "Grau 1"),

    Grau_Esp(0, "Grau Especial");

    private int id;

    private String descricao;

    private Grau(int id, String descricao) {

        this.id = id;

        this.descricao = descricao;

    }

    public String getGraus() { return descricao; }

    public int getId() { return id; }

}
```

ListaTécnicas - catálogo de 14 técnicas inatas:

```

package Jogador;

public enum ListaTecnicas {

    ILIMITADO(new Tecnica ("Ilimitado", 25)),

    TRANSFIGURACAO_INERTE(new Tecnica ("Transfiguração inerte", 20)),

    SANTUARIO(new Tecnica ("Santuario", 20)),

    PRINCIPE_DOS_RAIOS_NEGROS (new Tecnica ("Principe dos Raios Negros",
25)),

    // ...

    CHAMAS_DO_DESASTRE (new Tecnica ("Chamas do desastre", 20)),

    CONSTRUCAO(new Tecnica ("Construcao", 17)),

    COPIA(new Tecnica ("Copia", 0));

    private Tecnica tecnica;

    private ListaTecnicas (Tecnica tecnica) {

        this.tecnica = tecnica;

    }

    public Tecnica getTecnica() { return tecnica; }

}

```

3. Pilares de POO no projeto

3.1 Abstração

- A classe abstrata **Jogador** concentra os atributos e comportamentos comuns:
 - Nome, energia, grau, força, vida, agilidade, técnica.
 - Lógica de soco (**Socar**), entrada em zona (**entrarZona**), atualização de fim de turno, concentração de energia.

- Método abstrato `usarTecnicaInata (Jogador alvo)`, que cada subclasse especializa.
- `Tecnica` abstrai toda regra de uso de técnicas, incluindo:
 - Consumo de energia
 - Esquiva baseada em agilidade
 - Efeitos específicos ("Ilimitado", "Chamas do desastre", "Cópia", etc.)
 - Controle de domínio expandido e de choque de domínios.

Trecho da classe `Tecnica`:

```
package Jogador;

public class Tecnica {

    private String nome;

    private int poder;

    private boolean IsInDomain;

    private static boolean IsDomainClash = false;

    private int vitoriasClash = 0;

    public Tecnica (String nome, int poder) {

        this.nome = nome;

        this.poder = poder;

    }

    public int UsarTecnica (Jogador inimigo, Jogador usuario, int consumo)
    {

        // valida energia, aplica consumo e esquiva, etc.

        // ...

        return 0; // Exemplo de retorno

    }

    public void ExpandirDominio() {
```

```

        this.IsInDomain = true;

        System.out.println("Expansão de Domínio!");

        System.out.println("Acerto garantido ativado!");
    }

    public void FecharDominio() {

        this.IsInDomain = false;

        this.vitoriasClash = 0;

        System.out.println("Domínio fechado.");
    }
}

```

3.2 Encapsulamento

- Campos privados ou protegidos e acesso controlado via getters/setters:
 - `Tecnica`, `Placar`, `Participacao`, `Grau` usam atributos `private`.
 - `Jogador` usa `protected` para permitir acesso em subclasses, mas o mundo externo acessa via `getters`.
- Exemplo de validação de estado (invariantes) em `Jogador`:

```

public int setVidaAtual(int poder) {

    System.out.println(Nome + " Leva o ataque");

    vidaAtual -= poder;

    if (vidaAtual < 0) {

        System.out.println(Nome + " Faleceu :c");

        vidaAtual = 0;
    }

    return vidaAtual;
}

```

```
}
```

- Em **Feiticeiro / Maldição** (regeneração):

```
@Override

public void regenerarVida (int vidaDesejada) {

    if (!podeRegenerarVida (vidaDesejada)) {

        System.out.println(Nome + " não pode regenerar " + vidaDesejada + "
de vida!");

        return;

    }

    // garante que não passa da vida máxima e não gasta mais energia do que
tem

    // ...

}
```

3.3 Herança

Hierarquia principal:

```
public abstract class Jogador {

    // ...

    public abstract int usarTecnicaInata (Jogador alvo);

}

public class Feiticeiro extends Jogador implements Regeneravel {

    public Feiticeiro (String nome, int energia, Grau grau, int forca, int
vidaMaxima, int agilidade, Tecnica tecnica) {

        super(nome, energia, grau, forca, vidaMaxima, agilidade, tecnica);

    }

}
```

```

@Override

public int usarTecnicaInata (Jogador alvo) {

    return this.tecnica.UsarTecnica (alvo, this, 10);

}

// implementação específica de regeneração...
}

public class Maldicao extends Jogador implements Regeneravel {

    public Maldicao (String nome, int energia, Grau grau, int forca, int
vidaMaxima, int agilidade, Tecnica tecnica) {

        super (nome, energia, grau, forca, vidaMaxima, agilidade, tecnica);

    }

@Override

public int usarTecnicaInata (Jogador alvo) {

    return this.getTecnica().UsarTecnica (alvo, this, 10);

}

// implementação específica de regeneração...
}

Também há herança de interface:public class Partida implements
Comparator<Jogador> {

@Override

public int compare (Jogador j1, Jogador j2) {

    return Integer.compare(j2.getAgilidade(), j1.getAgilidade());

}

}
}

```

3.4 Polimorfismo

Sobrescrita (override):

- `Feiticeiro` e `Maldição` sobreescrevem `usarTecnicaInata`.
- Ambas implementam os métodos da interface `Regeneravel` com comportamentos distintos (custos diferentes).

Chamadas polimórficas reais:

1. Collections de `Jogador` em `Partida`

```
private List<Jogador> jogadores = new ArrayList<>();

// Lista contém tanto Feiticeiro quanto Maldicao
public Partida(Jogador jogador1, Jogador jogador2) {
    // ...
    jogadores.add(jogador1);
    jogadores.add(jogador2);
}
```

Na hora de executar ações:

```
for (int i = 0; i < jogadores.size(); i++) {
    Jogador jogador = jogadores.get(i);
    // chamada polimórfica:
    int pontosTecnica = jogador.usarTecnicaInata(jogadores.get(i == 0 ? 1 :
0));
}
```

2. Interface `Regeneravel`

```
Regeneravel[] regeneraveis = { todoFerido, hanami2 };

for (Regeneravel r: regeneraveis) {

    // chamada polimórfica: pode ser Feiticeiro ou Maldicao

    if (r.podeRegenerarVida(30)) {

        r.regenerarVida (30);

    }

}
```

3. `Comparator` (`Partida implements Comparator<Jogador>`) também é polimorfismo de interface quando usado em ordenações.

Observação honesta: o foco do projeto foi mais em sobrescrita e polimorfismo por herança/interface do que em sobrecarga explícita de métodos com várias assinaturas.

4. Relacionamentos entre classes

4.1 Cardinalidades

- **1:1 - Partida ↔ Placar**
 - Cada `Partida` cria exatamente um `Placar`

```
public class Partida implements Comparator<Jogador> {  
  
    private Placar placar;  
  
    public Partida (Jogador jogador1, Jogador jogador2) {  
  
        // ...  
  
        this.placar = new Placar (jogador1, jogador2);  
  
    }  
  
}
```

- **1:N - Partida → Jogador**

```
private List<Jogador> jogadores = new ArrayList<>();  
  
public Partida (Jogador jogador1, Jogador jogador2) {  
  
    if (!contemJogador(jogador1)) {  
  
        jogadores.add(jogador1);  
  
        participacoes.add(new Participacao (jogador1, this, 0, false));  
  
    }  
  
    if (!contemJogador (jogador2)) {
```

```

        jogadores.add(jogador2);

        participacoes.add(new Participacao (jogador2, this, 0, false));
    }
}

```

- **N:N - Jogador ↔ Partida via Participacao**

```

public class Participacao {

    private Jogador jogador;

    private Partida partida;

    private int danoTotal;

    private boolean venceu;

    // ...

}

```

- Um **Jogador** pode participar de várias **Partida** (cada uma gerando uma **Participacao** diferente).
- Uma **Partida** tem várias **Participacao**, uma por jogador.

4.2 Direcionamento

- **Unidirecional:**
 - **Placar** conhece os **Jogador**, mas **Jogador** não conhece o **Placar**

```

public class Placar {

    private Map<Jogador, Integer> pontos = new HashMap<>();

    public Placar (Jogador jogador1, Jogador jogador2) {

        pontos.put(jogador1, 0);

        pontos.put(jogador2, 0);

    }

}

```

```
}
```

- **Bidirecional:**
 - **Partida** tem uma lista de **Participacao**
 - **Participacao** conhece a **Partida**.

```
public class Partida {  
  
    private List<Participacao> participacoes = new ArrayList<>();  
  
    // ...  
}  
  
public class Participacao {  
  
    private Partida partida;  
  
    // ...  
}
```

4.3 Composição vs Agregação

- **Composição:**
 - **Partida** compõe um **Placar** e sua lista de **Participacao**:
 - Eles são criados no construtor da **Partida** e só fazem sentido dentro dela.
- **Agregação:**
 - **Partida** agrega **Jogador**:
 - Os objetos **Feiticeiro** e **Maldição** são criados fora e apenas "conectados" à **Partida**.
 - Se a partida acabar, os jogadores continuam existindo (podem participar de outras partidas ou ser usados em Roteiro).

5. Interface própria: Regeneravel

Interface:

```
package Jogador;
```

```
public interface Regeneravel {  
  
    void regenerarVida (int vidaDesejada);  
  
    boolean podeRegenerarVida (int vidaDesejada);  
  
    int getCustoRegeneracao();  
  
}
```

Implementação em Feiticeiro: `public class Feiticeiro extends Jogador implements Regeneravel {`

```
// ...
```

```
@Override
```

```
public boolean podeRegenerarVida (int vidaDesejada) {  
  
    int vidaFaltando = vidaMaxima - vidaAtual;  
  
    int vidaARegenerar = Math.min(vidaDesejada, vidaFaltando);  
  
    int custo = getCustoRegeneracao();  
  
    int energiaNecessaria = vidaARegenerar * custo;  
  
    return Energia >= energiaNecessaria && vidaARegenerar > 0;  
  
}
```

```
@Override
```

```
public int getCustoRegeneracao() {  
  
    return 2; // Feiticeiros gastam 2 de energia por 1 de vida  
  
}
```

```
}
```

Implementação em Maldição: `public class Maldicao extends Jogador implements Regeneravel {`

```
// ...
```

```

@Override

public int getCustoRegeneracao() {

    return 1; // Maldições gastam 1 de energia por 1 de vida

}

}

Uso polimórfico em Roteiro:Regeneravel[] regeneraveis = { todoFerido,
hanami2 };

for (Regeneravel r: regeneraveis) {

    if (r.podeRegenerarVida (30)) {

        r.regenerarVida(30);

    }

}

```

6. Collections, duplicidade e ordenação

6.1 Uso de List e Map

- Em **Partida**

```

private List<Jogador> jogadores = new ArrayList<>();
private List<Participacao> participacoes = new ArrayList<>();

```

- Em **Placar**

```

private Map<Jogador, Integer> pontos = new HashMap<>();

public void addPontos(Jogador jogador, int qtd) {
    pontos.put(jogador, pontos.get(jogador) + qtd);
}

```

6.2 Verificação de duplicidade

Em **Partida**, antes de adicionar jogador, é verificado se ele já está na lista:

```
private boolean contemJogador (Jogador jogador) {  
    for (Jogador j: jogadores) {  
        if (j.getNome().equals(jogador.getNome())) {  
            System.out.println("Jogador " + jogador.getNome() + " já está  
na partida!");  
            return true;  
        }  
    }  
    return false;  
}
```

Isso evita duplicidade de jogadores na mesma partida.

6.3 Ordenação

Ordenação dos jogadores por agilidade antes dos turnos (da maior para a menor):

```
public void acoesTurno() {  
    // Ordena jogadores por agilidade (maior primeiro)  
    jogadores.sort((j1, j2) -> Integer.compare(j2.getAgilidade(),  
j1.getAgilidade()));  
    System.out.println("ordem dos turnos: " + jogadores.get(0).getNome() +  
" e depois " + jogadores.get(1).getNome());  
    // ...  
}
```

E ainda há a implementação de `Comparator<Jogador>` em `Partida@Override`

```
public int compare (Jogador j1, Jogador j2) {  
  
    return Integer.compare(j2.getAgilidade(), j1.getAgilidade());  
  
}
```

7. Aplicativos principais (duas execuções)

7.1 App Interativo - Main

- Responsável por:
 - Ler nome dos jogadores com validação.
 - Permitir que cada jogador escolha sua técnica inata (1 a 14).
 - Criar os objetos **Feiticeiro** e **Maldição**.
 - Criar uma **Partida** e delegar a lógica para **Partida.acoesTurno()**.

Trecho:

```
import Jogador.*;  
  
import java.util.Scanner;  
  
public class Main {  
  
    public static Tecnica escolherTecnica (Scanner input, String  
nomeJogador) {  
  
        System.out.println("\n" + nomeJogador + ", escolha sua técnica  
inata:");  
  
        System.out.println("1 - Ilimitado");  
  
        // ...  
  
        System.out.println("14 - Cópia");  
  
        // leitura, validação e retorno de ListaTecnicas.X.getTecnica()  
  
        return ListaTecnicas.ILIMITADO.getTecnica(); // Exemplo
```

```

}

public static void main(String[] args) {

    Scanner input = new Scanner(System.in);

    System.out.println("Qual o nome do jogador 1?");

    String Nome1 = input.nextLine();

    // valida nome

    System.out.println("Qual o nome do jogador 2?");

    String Nome2 = input.nextLine();

    // valida nome

    Tecnica tecnicaJogador1 = escolherTecnica (input, Nome1);

    Tecnica tecnicaJogador2 = escolherTecnica (input, Nome2);

    int random_energy1 = (int) (Math.random() * 100) + 50;

    int random_energy2 = (int) (Math.random() * 100) + 50;

    Feiticeiro jogador1 = new Feiticeiro(Nome1,
random_energy1,Grau.Grau_1, /* forca */ ..., /* vida */ ..., /* agi */ ...,
tecnicaJogador1);

    Maldicao jogador2 = new Maldicao(Nome2, random_energy2,Grau.Grau_1,
/* forca */ ..., /* vida */ ..., /* agi */ ..., tecnicaJogador2);

    Partida partida = new Partida (jogador1, jogador2);

    partida.passarRodada();

    partida.acoesTurno();

}

}

```

Notar que a regra de negócio (esquiva, dano, domínio, etc.) está nas classes de modelo (**Jogador**, **Tecnica**, **Partida**), e o **Main** só orquestra entrada/saída.

7.2 App Roteiro - Roteiro

- Demonstra o funcionamento do sistema de forma determinística, sem input de usuário.
- Estrutura o output em vários blocos:
 - Criação de personagens (**Feiticeiro** e **Maldição**).
 - Exibição de status (**Showall**).
 - Socos e seus efeitos na vida.
 - Concentração de energia.
 - Uso de técnicas inatas.
 - Sistema de Zona / Black Flash.
 - Atualização de estados no fim do turno.
 - Variedade de técnicas (vários personagens com técnicas diferentes).
 - Sequência completa de combate.
 - Demonstração dos enums (**Grau** e **ListaTecnicas**).
 - Demonstração de **Partida** (turnos).
 - Sistema de esquiva comparando agilidade alta vs baixa.
 - Demonstração da interface **Regeneravel** com polimorfismo.
 - Sistema de pontuação (**Placar**).

Exemplo de criação no roteiro:

```
import Jogador.*;

public class Roteiro {

    public static void main(String[] args) {

        System.out.println("=== ROTEIRO DE DEMONSTRAÇÃO SISTEMA DE BATALHA  
JUJUTSU KAISEN ===\n");

        Feiticeiro gojo = new Feiticeiro(

            "Satoru Gojo",

            100,

            Grau.Grau_Esp,

            15,

            200,

            10,
```

```

        ListaTecnicas.ILIMITADO.getTecnica()

    );

    Maldição sukuna = new Maldicao(

        "Ryomen Sukuna",

        120,

        Grau.Grau_Esp,

        18,

        180,

        12,

        ListaTecnicas.SANTUARIO.getTecnica()

    );

    // demais demonstrações

}

}

```

8. Checklist das exigências e onde aparecem

Segue a tabela de requisitos e sua implementação no projeto:

Exigência	Status	Localização/Descrição
6 classes concretas de domínio	✓	Feiticeiro, Maldição, Tecnica, Partida, Participacao, Placar
Enum de domínio	✓	Grau (hierarquia de poder) e ListaTecnicas (técnicas inatas)

Exigência	Status	Localização/Descrição
Classe abstrata com métodos abstratos e concretos	✓	<code>Jogador</code> (abstrata, com <code>usarTecnicaInata</code> abstrato e vários métodos concretos)
Encapsulamento	✓	Atributos privados/protegidos com <code>getters/ setters</code> e validações (vida nunca negativa, energia suficiente, etc.)
Herança	✓	<code>Feiticeiro</code> e <code>Maldição</code> estendem <code>Jogador</code> ; <code>Partida</code> implementa <code>Comparator<Jogador></code>
Polimorfismo (sobrescrita + chamadas polimórficas)	✓	Sobrescrita de <code>usarTecnicaInata</code> . Interface <code>Regeneravel</code> com <code>Feiticeiro</code> e <code>Maldição</code> . <code>List<Jogador></code> contendo instâncias de subclasses. <code>Regeneravel[]</code> em <code>Roteiro</code>
Cardinalidades (1:1)	✓	<code>Partida</code> ↔ <code>Placar</code>
Cardinalidades (1:N)	✓	<code>Partida</code> → <code>Jogador</code> , <code>Partida</code> → <code>Participacao</code>
Cardinalidades (N:N)	✓	<code>Jogador</code> ↔ <code>Partida</code> via <code>Participacao</code>
Bidirecional	✓	<code>Partida</code> ↔ <code>Participacao</code>
Composição / Agregação	✓	Composição: <code>Partida</code> com <code>Placar</code> e <code>Participacao</code> . Agregação: <code>Partida</code> com <code>Jogador</code>
Interface própria implementada por ≥ 2 classes	✓	<code>Regeneravel</code> implementada por <code>Feiticeiro</code> e <code>Maldição</code>

Exigência	Status	Localização/Descrição
Collections com verificação de duplicidade	✓	<code>List<Jogador></code> + método <code>contemJogador</code> . <code>Map<Jogador, Integer></code> em <code>Placar</code>
Ordenação demonstrada	✓	<code>jogadores.sort(...)</code> por agilidade em <code>Partida.acoesTurno</code> , além de <code>compare</code> em <code>Partida</code>
App Interativo (menu + Scanner)	✓	<code>Main</code> cria os jogadores, escolhe técnica e inicia <code>Partida.acoesTurno()</code> (que contém o menu de ações)
App Roteiro (sem Scanner, saídas determinísticas)	✓	<code>Roteiro</code> mostra cenário fixo com várias demonstrações em sequência

9. Diagrama de classes

Esse é o diagrama de classes do projeto, onde mostra todas as relações entre as classes, métodos e atributos das mesmas.

