

Universidade de São Paulo (USP)

Escola de Artes, Ciências e Humanidades

Exercício Programa: Aplicação de princípios de POO a um código procedural

Disciplina: Computação Orientada a Objetos

Professor: Flávio Luiz Coutinho

Alunos:

Bruno Hideo Ionedá - 15573619

Guilherme Samuel Lemos Segura - 15575611

Higor Rangel Viani Lopes - 15552946

João de Melo Fantini - 15462550

São Paulo, Julho de 2025

Sumário

1	Introdução	2
2	Análise Crítica do Código Legado	3
3	A Arquitetura Orientada a Objetos da Pasta src	3
3.1	Polimorfismo	3
3.2	A Hierarquia de Entidades e o Encapsulamento	5
3.3	A Hierarquia de Inimigos	6
3.3.1	O Template <code>Enemy</code> Abstrato	6
3.3.2	Inimigos Comuns: <code>Enemy1</code> e <code>Enemy2</code>	6
3.3.3	A Sub-hierarquia dos Chefes: <code>Boss1</code> e <code>Boss2</code>	7
3.3.4	Interfaces	7
3.4	O Subsistema de Power-ups	7
4	O Sistema de Vida do Jogador e a <code>ExplodableEntity</code>	8
5	O Pacote <code>utils</code>	8
6	O Pacote <code>game</code>	9
7	O Pacote <code>graphics</code>	9
8	A Classe <code>Game</code>	9
9	Considerações Finais e Execução do Programa	10
9.1	Como Executar o Programa	10
9.2	Estrutura de Arquivos	11

1 Introdução

O projeto tem a seguinte estrutura:

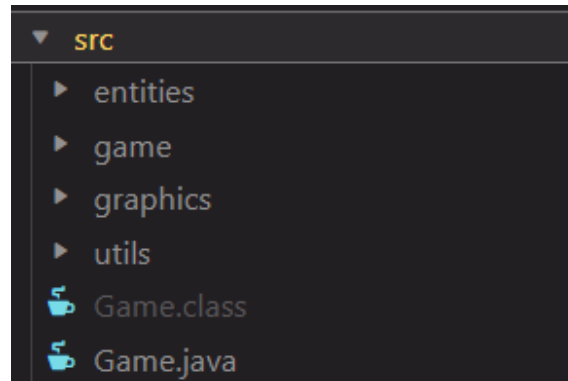


Figura 1: Estrutura dos pacotes

Breve introdução sobre cada pacote:

- entities: esse pacote possui as classes e interface responsáveis por modelar as entidades do sistema
- game: esse pacote possui os arquivos de configuração das fases e as classes responsáveis por ler esses arquivos e converter em objetos
- graphics: para cada classe no pacote entidade existe uma classe correspondente neste pacote que é responsável por desenhar na tela os gráficos
- utils: esse pacote possui classes, algumas estáticas, que servem como bibliotecas para auxiliar o código como por exemplo a classe que representa coordenadas

No tópico A Hierarquia de Entidades e o Encapsulamento é possível analisar como ficou a estrutura das classes do pacote entidades

2 Análise Crítica do Código Legado

A versão inicial do projeto que, embora funcional, foi desenvolvida utilizando práticas de programação que são amplamente consideradas ineficientes e difíceis de manter em projetos de software modernos. A análise do arquivo `Main.java` revela uma abordagem procedural, onde a lógica inteira do jogo—incluindo controle de tempo, estados de entidades, verificação de colisões, renderização e entrada do usuário—está contida em um único e massivo método `main`. Essa centralização excessiva torna a depuração, a modificação e a extensão do código tarefas extremamente complexas e propensas a erros.

A principal deficiência dessa abordagem é a completa ausência de encapsulamento e dos princípios da orientação a objetos. Em vez de modelar as entidades do jogo (como jogador, inimigos e projéteis) como objetos com seus próprios dados e comportamentos, o código utiliza uma série de "arrays paralelos" para armazenar suas propriedades. Por exemplo, para um inimigo, seu estado, posição X, posição Y e velocidade são armazenados em arrays distintos (`enemy1_states`, `enemy1_X`, `enemy1_Y`, `enemy1_V`). Essa separação entre dados e comportamento é a antítese do encapsulamento e leva a um código confuso, onde a lógica para atualizar um único inimigo está espalhada por múltiplos loops e acessos a diferentes arrays.

Adicionalmente, o código está repleto de "números mágicos", que são valores literais inseridos diretamente na lógica sem qualquer explicação contextual. Por exemplo, a duração de uma explosão é definida como `currentTime + 2000`. Essa prática prejudica a legibilidade e a manutenção, pois para alterar um parâmetro do jogo, como a duração de uma explosão ou a velocidade de um projétil, um desenvolvedor precisaria encontrar e substituir todos os locais onde aquele número aparece. A estrutura do jogo, incluindo a ordem e o tempo de surgimento dos inimigos, também é "hardcoded" (fixa no código), o que impede qualquer flexibilidade ou a criação de novas fases sem uma reescrita significativa do código-fonte.

3 A Arquitetura Orientada a Objetos da Pasta `src`

3.1 Polimorfismo

A classe `Game` é a principal beneficiária da arquitetura polimórfica do projeto. Ela explora essa característica para gerenciar todas as entidades de forma genérica e coesa, resultando em um código mais limpo e com maior manutenibilidade.

Uma das aplicações mais diretas do polimorfismo é na atualização das entidades. A classe mantém listas como `List<Enemy> enemies`, que pode conter instâncias de `Enemy1` e `Enemy2`. Dentro do loop principal, ao iterar sobre essa lista, a classe `Game` faz chamadas de método como `enemy.update(delta)` e `enemy.shoot(...)`. Graças à ligação dinâmica, a implementação correta do método é executada para cada objeto. Isso elimina a necessidade de blocos condicionais para tratar cada tipo de inimigo individualmente.

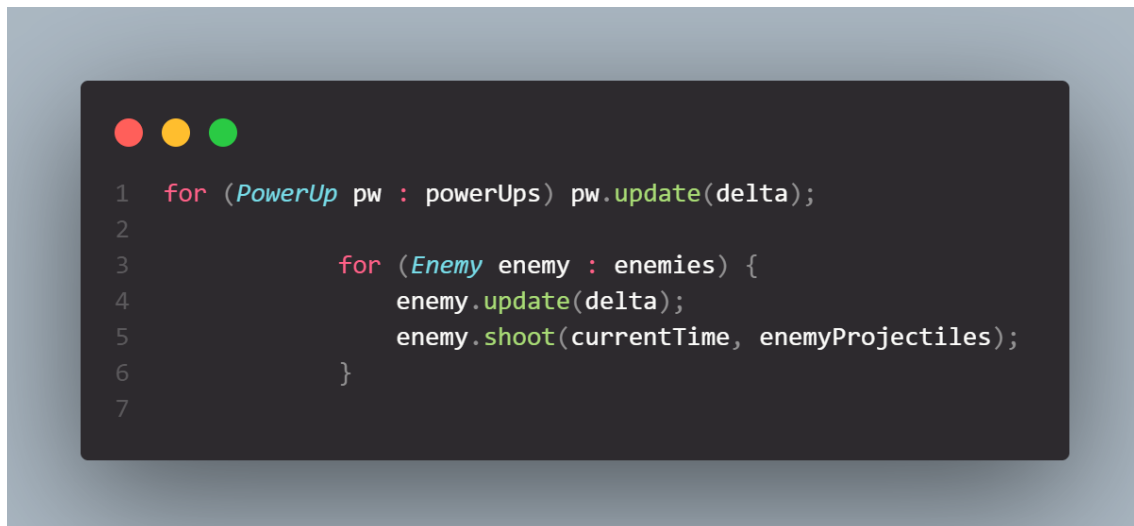


Figura 2: Exemplo de polimorfismo

O sistema de verificação de colisões é outro exemplo poderoso. A classe `Game` não implementa a lógica de cálculo de distância; ela delega essa tarefa ao método estático `Collision.VerifyCollision(ICollidable a, ICollidable b)`. Como os parâmetros são do tipo da interface `ICollidable`, a classe `Game` pode passar qualquer combinação de objetos que implementem este contrato: um `Projectile` e um `Enemy`, um `Player` e um `PowerUp`, etc. O método opera sobre a abstração, tornando o código na classe `Game` extremamente reutilizável.



Figura 3: Código que verifica as colisões dos projéteis com o player ou do player com os inimigos

Finalmente, o polimorfismo simplifica o gerenciamento do ciclo de vida das entidades. A remoção de entidades inativas é feita de forma elegante através de `enemies.removeIf(e -> e.getState() == States.INACTIVE)`. Essa operação funciona perfeitamente porque o método `getState()` e o enum `States` são

partes do contrato comum definido na classe base `Entity`, da qual todas as entidades herdam. A lógica de limpeza é, portanto, genérica e desacoplada dos detalhes de cada tipo de entidade. Em total contraste com a versão legada, a estrutura de código na pasta `src` adota uma arquitetura orientada a objetos robusta e bem planejada. Ela utiliza os pilares da herança, encapsulamento e polimorfismo, juntamente com interfaces, para criar um sistema flexível, reutilizável e muito mais fácil de entender e manter.

3.2 A Hierarquia de Entidades e o Encapsulamento

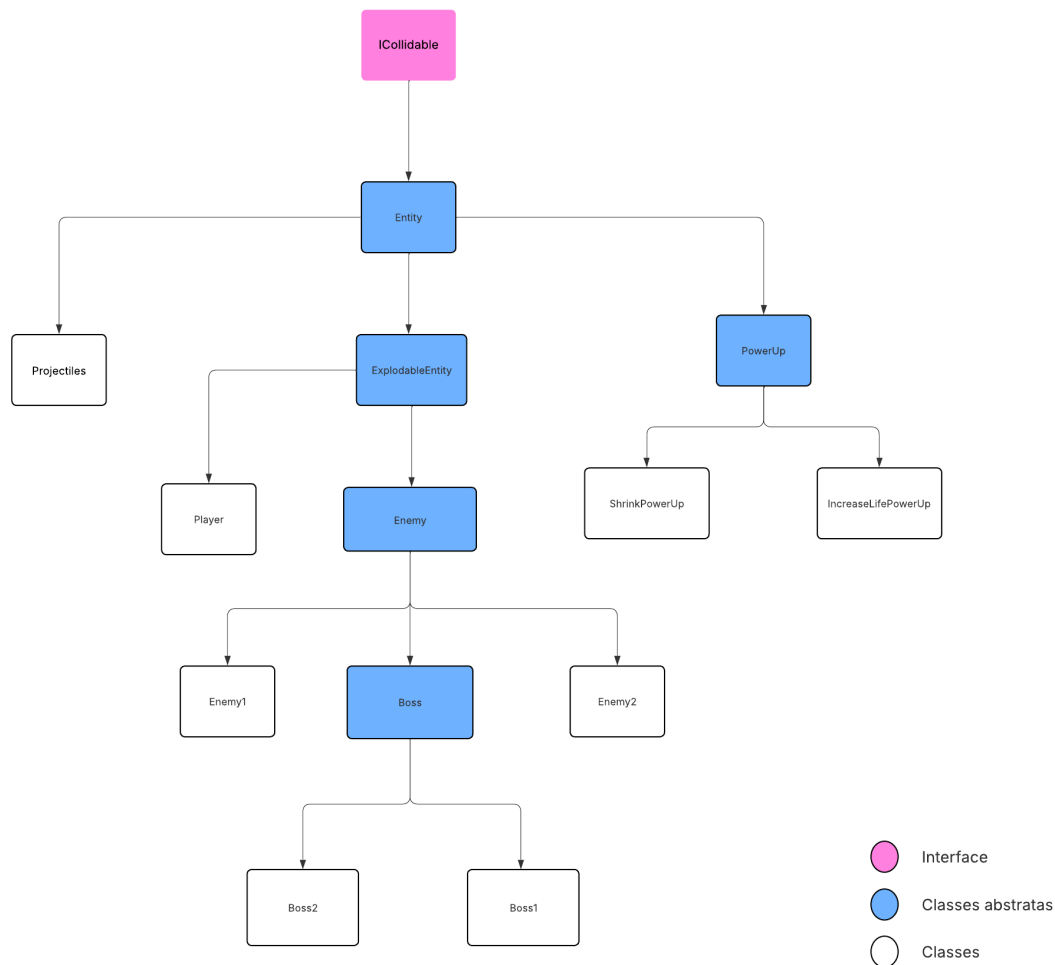


Figura 4: Estrutura das classes do pacote entidade

A ideia central dessa estrutura foi permitir o máximo possível abstração e reutilização de código com isso foi implementada a interface `ICollidable` a fim de facilitar as Colisões, e depois disso a partir de várias estruturas hierárquicas com classes abstratas foi implementado as outras entidades.

O uso das classes abstratas se deu, pois nesses casos existe uma relação forte de "é isso" e além disso as classes abstratas permitem a maior reutilização de código e

como esse código não será reutilizado em nenhum outro projeto, o projeto tem um escopo fechado, o maior acoplamento da herança não é um grande problema. as principais classes:

- **Entity**: essa Classe abstrata é uma base para todas as outras entidades do jogo, por isso ela só implementa funcionalidades básicas. Parâmetros: Coordenadas, velocidade, estado e raio (que define o tamanho de todas as entidades mesmo que não necessariamente sejam um círculo). Métodos: getters e setters dos parâmetro e update (responsável por movimentar, verificar explosão e atualizar seus estados.
- **ExplodableEntity**: essa classe é responsável por definir os métodos que gerenciam a explosão e a vida das entidades que estendem ela. código da classe `ExplodableEntity`
- **Enemy**: Classe que define comportamentos padrão da classe `Enemy`. Atributos: `nextShotTime`, `angle` e `rotationalVelocity`. Métodos como: `shouldShoot` ou `shouldSpawnPowerUp`. código da classe `Enemy`
- **Boss**: uma classe padrão que define atributos como `lastAttackTime` e `attackCooldown`. código da classe `Boss`

3.3 A Hierarquia de Inimigos

Assim como o jogador, os adversários do jogo são modelados através de uma hierarquia lógica que promove o reuso de código e permite a criação de comportamentos variados.

3.3.1 O Template Enemy Abstrato

A classe abstrata `Enemy` funciona como um "contrato" para todos os inimigos do jogo . Por herdar de `ExplodableEntity`, ela garante que todo inimigo possua um sistema de vida. Sua principal função é definir métodos abstratos como `shouldShoot()` e `shoot()`, forçando as subclasses a implementar sua própria lógica de ataque.

3.3.2 Inimigos Comuns: `Enemy1` e `Enemy2`

As implementações concretas `Enemy1` e `Enemy2` dão vida ao template `Enemy`. O `Enemy1` (ciano) tem seu gatilho de tiro definido pela constante `SHOOTING_HEIGHT_RATIO`, atirando apenas quando está na faixa intermediária da tela . O `Enemy2` (magenta) possui uma lógica mais elaborada: seu comportamento de rotação é ativado ao cruzar o `ACTIVATION_THRESHOLD` (30% da altura da tela), com sua velocidade rotacional sendo definida por `LEFT_SIDE_RV` ou `RIGHT_SIDE_RV`. Ele só atira quando seu ângulo de voo se alinha a 0 ou 3π radianos, com uma tolerância definida por `ANGLE_TOLERANCE` . A representação visual de cada um é tratada separadamente na classe `EnemyGraphics` .

3.3.3 A Sub-hierarquia dos Chefes: Boss1 e Boss2

Os chefes de fase são implementados como especializações de **Boss**. A classe **Boss1** introduz um inimigo com alta durabilidade e múltiplos padrões de ataque que se alternam via **switch** (**performAttack**) Ataque do Boss1. A classe **Boss2** herda de **Boss** e implementa um ataque na teleguiado. Para isso, ela calcula o vetor direção (**direction**) subtraindo as coordenadas do chefe das do jogador e o normaliza, criando um ataque que persegue ativamente o jogador e utiliza a interface **IPlayerCoord** para garantir que a classe **Boss** só tenha acesso as coordenadas do **Player** Ataque do Boss2.

3.3.4 Interfaces

Para complementar a herança, o projeto faz um uso inteligente de interfaces para definir “contratos”, o que fragmenta comportamentos complexos em capacidades modulares e reutilizáveis. Essa abordagem resulta em um design de código mais flexível, desacoplado e robusto.

- **ICollidable**: É o contrato mais fundamental para a física do jogo. Ele define que qualquer objeto “colidível” deve, obrigatoriamente, fornecer sua **Coordinate**, **State** e **Radius**. O poder dessa abordagem é demonstrado no método estático **Collision.VerifyColision(ICollidable a, ICollidable b)**, que pode verificar colisões entre *quaisquer* duas entidades que implementem a interface (como um **Player** e um **Enemy**, ou um **Tiro** e um **PowerUp**), tornando o sistema de colisão totalmente genérico.
- **IPlayerCoord**: Um exemplo de interface mínima e altamente específica. Seu único propósito é fornecer a coordenada do jogador. Isso permite que sistemas complexos, como mísseis teleguiados ou interfaces de usuário, dependam apenas da localização do jogador, sem precisar de acesso ao objeto **Player** completo, promovendo um baixo acoplamento no código.

3.4 O Subsistema de Power-ups

O sistema de power-ups adiciona uma camada estratégica e dinâmica ao jogo. A base para esse sistema é a classe abstrata **PowerUp**, que herda de **Entity** e define o contrato polimórfico **onCollected(Player player)**. Esse método dita o comportamento específico de cada power-up quando coletado pelo jogador, permitindo uma variedade de efeitos.

Existem diferentes tipos de power-ups, cada um com uma função única:

- **IncreaseLifePowerUp**: Concede ao jogador um bônus de vida instantâneo. Ao ser coletado, ele calcula 20% da vida máxima do jogador e a adiciona à sua vida atual. Caso a cura exceda o máximo, a vida do jogador é simplesmente preenchida por completo. Seu efeito é imediato e permanente, ativado uma única vez no momento da coleta.

- **ShrinkPowerUp**: Oferece uma vantagem tática temporária. Ao ser ativado, ele reduz o tamanho do jogador, tornando-o um alvo mais difícil de atingir por 5 segundos. Sua lógica de tempo é encapsulada no método `update`: ele armazena o tempo inicial (`startTime`) e, a cada quadro, verifica se `startTime + 5000 <= System.currentTimeMillis()`. Quando a condição é satisfeita, o efeito é revertido, e o jogador retorna ao seu tamanho normal.

4 O Sistema de Vida do Jogador e a `ExplodableEntity`

A implementação do sistema de vida e dano do jogador é um excelente caso de estudo sobre como a herança e a especialização de comportamento funcionam juntas. A base para este sistema é a classe abstrata `ExplodableEntity`, que centraliza a lógica fundamental de ter pontos de vida e ser destruído. Ela introduz o atributo `health` e os métodos `getHealth()` e `setHealth()`, além de um método `takeDamage(int damage)` que encapsula a lógica de redução de vida e a transição para o estado `EXPLODING` quando a vida chega a zero.

A classe `Player` estende `ExplodableEntity`, herdando todo esse sistema de saúde sem a necessidade de reescrever código. No entanto, ela especializa esse comportamento ao sobrescrever (`@Override`) o método `takeDamage(int damage)`. Dentro de sua própria implementação, antes de reduzir a vida, ela primeiro verifica se o jogador está invulnerável (`isInvulnerable()`). Se não estiver, ela prossegue com a lógica de dano e então ativa seu próprio sistema de invulnerabilidade e feedback visual (o piscar), definindo `lastHitTime` e `blinkStartTime`.

Este sistema é ainda complementado pela interação com os power-ups. A classe `IncreaseLifePowerUp`, ao ser coletada, executa seu método `onCollected(Player player)`. Este método interage com o sistema de vida do jogador: ele chama `player.getHealth()` e `player.getMaxHealth()` para verificar se a cura é necessária e, em caso afirmativo, chama `player.setHealth(player.getHealth() + 1)`.

Para manter a organização do código, a lógica de desenho de todos os power-ups, independentemente de seus efeitos, é centralizada e encapsulada na classe `PowerUpGraphics`, que cuida de sua representação visual na tela.

5 O Pacote `utils`

O pacote `utils` auxilia na parte gráfica e possui algumas classes que servem como bibliotecas externas para o sistema do jogo. A classe `GameLib` atua como uma *facade* para a biblioteca gráfica (Java Swing/AWT), encapsulando a criação da janela (`MyFrame`) e o tratamento de entrada (`MyKeyAdapter`), que traduz os `KeyEvent`s brutos em um array de booleanos de fácil consulta. A classe `Coordinate` agrupa os valores `x` e `y` em um objeto único e com tipo definido. A enumeração `States` substitui constantes inteiras por um tipo enumerado (`ACTIVE`, `INACTIVE`, `EXPLODING`), o que garante segurança de tipo e torna o código auto-documentado.

6 O Pacote game

Este sistema desacopla o design das fases do código-fonte. A classe `Loader` Código da classe `Loader` lê o arquivo de configuração principal, `game_config.txt`, para obter parâmetros globais e os caminhos para os arquivos de cada fase. Para cada caminho, um objeto `Level` Código da classe `Level` é instanciado, que por sua vez analisa o arquivo de texto da fase (ex: `fase1.txt`). Cada linha neste arquivo é convertida em um objeto `SpawnEvent`, que armazena os dados de forma estruturada. A classe `Game` então consome essa lista de eventos, permitindo que o design das fases seja alterado apenas editando arquivos de texto.

7 O Pacote graphics

O pacote `graphics` isola completamente a lógica de renderização da lógica de jogo. Classes utilitárias estáticas como `PlayerGraphics`, `EnemyGraphics` e `BossGraphics` recebem os objetos de jogo e usam `GameLib` para renderizá-los. `PlayerGraphics`, por exemplo, encapsula a complexa lógica de desenhar os corações de vida com um estilo pixel-art. `Boss2Graphics` contém métodos como `drawWing`, que utiliza `Math.sin(currentTime * 0.004)` para criar uma animação periódica de bater de asas, dando vida aos chefes.

8 A Classe Game

A arquitetura do jogo é coordenada pela classe `Game`, que centraliza o loop principal e a interação entre todos os sistemas e entidades.

Inicialização

O jogo começa no método `main`, onde as classes `Loader` e `LevelLoader` carregam as configurações e os dados dos níveis a partir de arquivos. Em seguida, o `Player` é criado e as listas que armazenarão as entidades dinâmicas (`Projectiles`, `Enemy`, `PowerUp`) são inicializadas.

O Game Loop

O loop principal (`while (running)`) é o coração do jogo, executando continuamente a cada quadro e seguindo uma sequência lógica clara:

1. **Verificação de Estado:** O loop primeiro checa estados globais como `gameOver`, `levelCompleted` ou `isGameWon`. Se algum for verdadeiro, ele exibe a tela apropriada (ex: `GameOverGraphics`) e pula o resto da lógica do quadro. Verificação de Estado
2. **Atualização de Entidades:** Todas as entidades ativas (`Player`, `Enemy`, `Boss`, etc.) têm seus métodos `update(delta)` chamados, o que recalcula

suas posições e estados. Ações como atirar (**shoot**) também são executadas aqui. Atualização de Entidades

3. **Gerenciamento de Spawns:** A classe **Spawner** é consultada para processar eventos de **spawn**, adicionando novos inimigos e chefes ao jogo conforme definido pelo nível atual. O método `processSpawnevents` além de processar os spawns ele também retorna o boss caso seja a hora certa. Também existe um método específico para lidar com o spawn do **Enemy2**. Gerenciamento de Spawns
4. **Detecção de Colisão:** O `loop` verifica colisões entre as diferentes entidades usando o método estático `Collision.VerifyCollision()`. As interações incluem projéteis contra inimigos, jogador coletando power-ups e contato direto entre o jogador e os adversários. Detecção de Colisão
5. **Remoção de Entidades:** No final, entidades marcadas como **INACTIVE** (como inimigos destruídos ou projéteis que atingiram um alvo) são eficientemente removidas das listas usando `removeIf`, otimizando o desempenho. Remoção de Entidades
6. **Renderização:** Por fim, as classes de **Graphics** (**PlayerGraphics**, **EnemyGraphics**, etc.) são chamadas para desenhar todas as entidades ativas na tela, que é atualizada com `GameLib.display()`. Renderização

Essa estrutura garante uma **separação de responsabilidades**, onde a classe **Game** foca no fluxo de alto nível, enquanto a lógica específica de cada componente (gráficos, carregamento de níveis, comportamento de entidades) é encapsulada em suas próprias classes.

9 Considerações Finais e Execução do Programa

O projeto desenvolvido representa uma significativa evolução em relação ao código procedural original, demonstrando na prática os benefícios da orientação a objetos. Através da aplicação do encapsulamento e do polimorfismo, criamos um sistema modular, extensível e de fácil manutenção. A separação clara entre lógica de jogo, renderização e configuração permite que cada aspecto possa ser modificado independentemente, seguindo o princípio da responsabilidade única.

9.1 Como Executar o Programa

Para executar o jogo, siga os seguintes passos:

1. **Compilação:**
 - Navegue até o diretório raiz do projeto: a pasta **src**
 - Execute o comando:

```
javac Game.java
```

2. Execução:

- A partir do diretório raiz, execute:

```
java Game
```

3. Controles:

- **Teclas direcionais:** Movimentação do jogador
- **CTRL:** Disparar projéteis
- **ESC:** Sair do jogo

9.2 Estrutura de Arquivos

O projeto está organizado da seguinte forma:

- **EP/:** Diretório para as pastas e arquivos.
 - **src/:** Código fonte organizado em pacotes
 - * **entities/:** Classes e interfaces das entidades do jogo
 - **interfaces/:** Interfaces
 - * **graphics/:** Renderização dos elementos visuais
 - * **utils/:** Classes utilitárias e constantes
 - * **game/:** Arquivos de configuração do jogo e carregamento das fases
 - * **Game.java:** Lógica principal do jogo

```

1  public abstract class ExplodableEntity extends Entity {
2      private int health;
3      private double explosionStart;
4      private double explosionEnd;
5
6      public ExplodableEntity(Coordinate coordinate, Coordinate velocity, States state, double radius, int health) {
7          super(coordinate, velocity, state, radius);
8          this.health = health;
9          this.explosionStart = 0;
10         this.explosionEnd = 0;
11     }
12
13     public int getHealth() {
14         return health;
15     }
16
17     public void setHealth(int health) {
18         this.health = health;
19     }
20
21     public void takeDamage(int damage) {
22         this.health -= damage;
23         if (this.health <= 0) {
24             this.setState(States.EXPLODING);
25         }
26     }
27
28     public double getExplosionStart() {
29         return explosionStart;
30     }
31
32     public void setExplosionStart(double explosionStart) {
33         this.explosionStart = explosionStart;
34     }
35
36     public double getExplosionEnd() {
37         return explosionEnd;
38     }
39
40     public void setExplosionEnd(double explosionEnd) {
41         this.explosionEnd = explosionEnd;
42     }
43
44     public boolean isExploding(Long currentTime) {
45         return this.getState() == States.EXPLODING &&
46             currentTime >= explosionStart &&
47             currentTime <= explosionEnd;
48     }
49
50     public boolean hasFinishedExploding(Long currentTime) {
51         return this.getState() == States.EXPLODING &&
52             currentTime > explosionEnd;
53     }
54     public void explosion(Long currentTime)
55     {
56         setState(States.EXPLODING);
57         setExplosionStart(currentTime);
58         setExplosionEnd(currentTime + 500);
59     }
60
61 }

```

Figura 5: código da classe ExplodableEntity

```

1  public abstract class Enemy extends ExplodableEntity{
2      protected static final Long DEFAULT_SHOOT_COOLDOWN = 1000;
3
4      protected Long nextShotTime;
5      protected double angle;
6      protected double rotationalVelocity;
7
8      public Enemy(Coordinate coordinate, Coordinate velocity, States state, double radius, int health) {
9          super(coordinate, velocity, state, radius, health);
10         this.nextShotTime = System.currentTimeMillis() + DEFAULT_SHOOT_COOLDOWN;
11         this.angle = (3 * Math.PI) / 2.0;
12     }
13
14     public double getAngle() {
15         return angle;
16     }
17
18     public void setAngle(double angle) {
19         this.angle = angle;
20     }
21
22     public double getRotationalVelocity() {
23         return rotationalVelocity;
24     }
25
26     public void setRotationalVelocity(double rotationalVelocity) {
27         this.rotationalVelocity = rotationalVelocity;
28     }
29
30     public boolean isShotCooldownOver(Long currentTime) {
31         return currentTime > nextShotTime;
32     }
33
34     public void resetShotCooldown(Long currentTime) {
35         this.nextShotTime = currentTime + DEFAULT_SHOOT_COOLDOWN;
36     }
37
38     /*Serão implementadas por Enemy1 e Enemy2*/
39     public abstract boolean isOnScreen();
40
41     public abstract boolean shouldShoot();
42
43     public PowerUp shouldSpawnPowerUp(double x, double y) {
44         if (Math.random() < 0.15) { // 15% de chance de spawn
45
46             double speedY = 0.1;
47             Coordinate coord = new Coordinate(x, y);
48             Coordinate velocity = new Coordinate(0, speedY);
49
50             // Decide entre os dois tipos de power up
51             if (Math.random() < 0.5) {
52                 return new IncreaseLifePowerUp(coord, velocity, 10.0);
53             } else {
54                 return new ShrinkPowerUp(coord, velocity, 10.0);
55             }
56         }
57         return null;
58     }
59
60     public abstract void shoot(Long currentTime, List<Projectiles> pPlayerProjectiles);
61
62 }

```

Figura 6: código da classe Enemy

```

1  public abstract class Boss extends Enemy{
2
3      private Long lastAttackTime;
4      Long attackCooldown = 500; // 2 seconds
5      private int maxHealth;
6      private static final double MAX_RADIUS = 30.0; //tamanho dos Bosses
7
8      //ATRIBUTOS DE VELOCIDADE
9      private static final double MAX_VX = 0.1;
10     private static final double MAX_VY = 0.05;
11
12     public Boss(Coordinate coordinate,int maxHealth)
13     {
14         super(coordinate, new Coordinate(MAX_VX, MAX_VY), States.ACTIVE, MAX_RADIUS, maxHealth);
15         this.lastAttackTime = System.currentTimeMillis();
16         this.maxHealth = maxHealth;
17     }
18
19     public boolean canAttack() {
20         return System.currentTimeMillis() - lastAttackTime > attackCooldown && getState() == States.ACTIVE;
21     }
22
23     public boolean isOnScreen() {
24         return getX() > -getRadius() && getX() < GameLib.WIDTH + getRadius() &&
25             getY() > -getRadius() && getY() < GameLib.HEIGHT + getRadius();
26     }
27
28     public boolean shouldShoot() {
29         return canAttack();
30     }
31
32     public int getMaxHealth() {
33         return maxHealth;
34     }
35
36     public double getMaxRadius() {
37         return MAX_RADIUS;
38     }
39
40     public Long getLastAttackTime()
41     {
42         return lastAttackTime;
43     }
44
45     public void setLastAttackTime(Long L)
46     {
47         this.lastAttackTime = L;
48     }
49
50     public void setAttackCooldown(Long L)
51     {
52         attackCooldown = L;
53     }
54
55     public void shoot(Long currentTime, List<Projectiles> enemyProjectiles)
56     {
57         if (shouldShoot()) {
58             performAttack(enemyProjectiles);
59         }
60     }
61
62     public abstract void update(Long delta);
63     public abstract void performAttack(List<Projectiles> enemyProjectiles);
64
65     public void takeDamage(int damage) {
66         if (getState() != States.ACTIVE) return;
67
68         setHealth(getHealth() - damage);
69         if (getHealth() <= 0) {
70             setHealth(0);
71             explosion(System.currentTimeMillis());
72             setState(States.INACTIVE);
73         }
74     }
75 }
76

```

```

1  public void performAttack(List<Projectiles> enemyProjectiles) {
2      super.setLastAttackTime(System.currentTimeMillis());
3      switch (attackPattern) {
4          case 0:
5              // Disparo em Leque
6              for (int i = -2; i <= 2; i++) {
7                  double angle = Math.PI / 2 + i * (Math.PI / 8);
8                  double vx = Math.cos(angle) * 0.3;
9                  double vy = Math.sin(angle) * 0.3;
10                 enemyProjectiles.add(new Projectiles(
11                     new Coordinate(getX(), getY()),
12                     new Coordinate(vx, vy),
13                     5.0,
14                     Projectiles.ENEMY_PROJECTILE,
15                     1
16                 ));
17             }
18             break;
19
20         case 1:
21             // Disparo único forte
22             enemyProjectiles.add(new Projectiles(
23                 new Coordinate(getX(), getY()),
24                 new Coordinate(0, 0.2),
25                 20.0,
26                 Projectiles.ENEMY_PROJECTILE,
27                 4
28             ));
29             break;
30     }
31
32     attackPattern = (attackPattern + 1) % 2;
33 }

```

Figura 8: Ataque do Boss1


```

1  public void performAttack(List<Projectiles> enemyProjectiles) {
2      if (!canAttack()) return;
3
4
5      if(cycle_counter <= 50)
6      {
7          if(cycle_counter % 25 == 0){
8              Coordinate direction = new Coordinate(player.getCoordinate().getX() - getX(), player.getCoordinate().getY() - getY());
9              double size_direction = Math.sqrt((direction.getX()* direction.getX()) + (direction.getY() * direction.getY()));
10             Coordinate normdirection = new Coordinate(direction.getX()/size_direction, direction.getY()/size_direction);
11
12             enemyProjectiles.add(new Projectiles(
13                 new Coordinate(getX() + 40, getY()),
14                 new Coordinate(normdirection.getX() * Math.random() * 1.1, normdirection.getY() * Math.random() * 1.1),
15                 10,
16                 Projectiles.ENEMY_PROJECTILE,
17                 3
18             ));
19         }
20         cycle_counter++;
21         return;
22     }
23     enemyProjectiles.add(new Projectiles(
24         new Coordinate(getX() + 40, getY()),
25         new Coordinate(0, 0.4),
26         10,
27         Projectiles.ENEMY_PROJECTILE,
28         2
29     ));
30     cycle_counter++;
31     if(cycle_counter > 600) cycle_counter = 0;
32     setlastAttackTime(System.currentTimeMillis());
33 }

```

Figura 9: Ataque do Boss2

```

1  public class Loader {
2
3      private int playerLife;
4      private List<Level> levels;
5
6      public Loader(Path configFilePath) throws IOException {
7          levels = new ArrayList<>();
8          loadGameConfig(configFilePath);
9      }
10
11
12      private void loadGameConfig(Path configFilePath) throws IOException {
13
14          List<String> lines = Files.readAllLines(configFilePath);
15
16          if(lines.size() < 2){
17              throw new IllegalArgumentException("Configuração inválida: deve conter pelo menos a vida do jogador e número de fases.");
18          }
19
20          // A primeira linha representa a vida do jogador. Hint: trim() -> remove espaços em branco no início e no fim da string
21          playerLife = Integer.parseInt(lines.get(0).trim());
22
23          // Segunda linha: número de fases
24          int numLevels = Integer.parseInt(lines.get(1).trim());
25
26          // Demais linhas: caminhos dos arquivos de configurações das fases
27          for(int i = 0; i < numLevels; i++){
28              String levelPath = lines.get(i + 2).trim();
29              Level level = new Level(Path.of(levelPath));
30              levels.add(level);
31          }
32      }
33
34  }
35
36  public int getPlayerLife() {
37      return playerLife;
38  }
39
40
41  public List<Level> getLevels(){
42      return levels;
43  }
44  }

```

Figura 10: Código da classe Loader

```

1  public class Level {
2
3      private List<SpawnEvent> events;
4
5      public Level(Path configFilePath) throws IOException{
6          events = new ArrayList<>();
7          loadLevelConfig(configFilePath);
8      }
9
10     private void loadLevelConfig(Path configFilePath) throws IOException {
11         List<String> lines = Files.readAllLines(configFilePath);
12
13         boolean bossFound = false;
14
15         for (String line : lines) {
16             // Basicamente ignora os caracteres inválidos
17             line = line.replaceAll("//.*", "").trim();
18             if (line.isEmpty()) continue;
19
20             String[] parts = line.split("\\s+");
21             String keyword = parts[0];
22
23             try{
24                 if(keyword.equalsIgnoreCase("INIMIGO")){
25                     if(parts.length != 5) throw new IllegalArgumentException("Linha de inimigo mal formatada: " + line);
26
27                     int type = Integer.parseInt(parts[1]);
28                     int time = Integer.parseInt(parts[2]);
29                     int x = Integer.parseInt(parts[3]);
30                     int y = Integer.parseInt(parts[4]);
31
32                     events.add(new SpawnEvent(SpawnEvent.Type.INIMIGO, type, 1, time, x, y));
33
34                 }else if (keyword.equalsIgnoreCase("CHEFE")){
35                     if (parts.length != 6) throw new IllegalArgumentException("Linha de chefe mal formatada: " + line);
36                     if (bossFound) throw new IllegalArgumentException("Mais de um chefe definido na fase!");
37
38                     int type = Integer.parseInt(parts[1]);
39                     int health = Integer.parseInt(parts[2]);
40                     int time = Integer.parseInt(parts[3]);
41                     int x = Integer.parseInt(parts[4]);
42                     int y = Integer.parseInt(parts[5]);
43
44                     events.add(new SpawnEvent(SpawnEvent.Type.CHEFE, type, health, time, x, y));
45                     bossFound = true;
46
47                 }else{
48                     throw new IllegalArgumentException("Linha desconhecida no arquivo da fase: " + line);
49                 }
50             }catch (NumberFormatException e){
51                 throw new IllegalArgumentException("Erro ao converter número na linha: " + line, e);
52             }
53         }
54
55         if (!bossFound) {
56             throw new IllegalArgumentException("A fase necessita obrigatoriamente de um chefe");
57         }
58     }
59
60     public List<SpawnEvent> getEvents(){
61         return events;
62     }
63 }

```

Figura 11: Código da classe Level

```

1  while (running) {
2      delta = System.currentTimeMillis() - currentTime;
3      currentTime = System.currentTimeMillis();
4
5      if (GameLib.isKeyPressed(GameLib.KEY_ESCAPE)) {
6          running = false;
7      }
8
9      if (gameOver) {
10         GameOverGraphics.drawGameOver();
11         GameLib.display();
12         continue;
13     }
14
15     if (levelLoader.isGameWon()) {
16         VictoryGraphics.drawVictory();
17         GameLib.display();
18         continue;
19     }
20
21     if (levelCompleted) {
22         LevelCompletedGraphics.drawLevelCompleted();
23         GameLib.display();
24
25         if (currentTime - levelCompleteTime > 3000) { // Mostra por 2 segundos
26             levelCompleted = false;
27             LevelCompletedGraphics.resetTimer();
28             if (levelLoader.hasMoreLevels()) {
29                 levelLoader.nextLevel();
30                 levelLoader.startLevel(levelLoader.getCurrentLevelIndex(), currentTime);
31                 spawner.reset(currentTime);
32                 playerProjectiles.clear();
33                 enemyProjectiles.clear();
34                 enemies.clear();
35                 powerUps.clear();
36                 player.setRadius(player.getMaxRadius());
37                 player.setHealth(player.getMaxHealth());
38                 boss = null;
39             }
40         }
41         continue;

```

Figura 12: Verificação de Estado




```

1  player.update(delta);
2  player.shoot(currentTime, playerProjectiles);
3
4  if (player.getHealth() <= 0) {
5      gameOver = true;
6      player.setState(States.INACTIVE);
7      continue;
8  }
9
10 for (Projectiles projectile : playerProjectiles) projectile.update(delta);
11 for (Projectiles projectile : enemyProjectiles) projectile.update(delta);
12 for (PowerUp pw : powerUps) pw.update(delta);
13
14 for (Enemy enemy : enemies) {
15     enemy.update(delta);
16     enemy.shoot(currentTime, enemyProjectiles);
17 }
18
19 if (boss != null) {
20     boss.update(delta);
21     boss.shoot(currentTime, enemyProjectiles);
22
23     if (boss.getHealth() <= 0) {
24         boss.explosion(currentTime);
25         levelLoader.setLevelCompleted(true);
26         levelCompleted = true;
27         levelCompleteTime = currentTime;
28     }
29 }

```

Figura 13: Atualização de Entidades



```

1  Boss newBoss = spawner.processSpawnEvents(currentTime, enemies, player);
2      if (newBoss != null) {
3          boss = newBoss;
4      }
5
6      spawner.handleEnemy2SquadSpawning(currentTime, enemies);

```

Figura 14: Gerenciamento de Spawns

```

1  for (Projectiles p : playerProjectiles) {
2      for (Enemy e : enemies) {
3          if (Collision.VerifyCollision(p, e)) {
4              e.expllosion(currentTime);
5              PowerUp pw = e.shouldSpawnPowerUp(e.getX(), e.getY());
6              if (pw != null) {
7                  powerUps.add(pw);
8              }
9              p.setState(States.INACTIVE);
10         }
11     }
12
13     if (boss != null && boss.getState() == States.ACTIVE && Collision.VerifyCollision(p, boss)) {
14         boss.takeDamage(p.getDamage());
15         p.setState(States.INACTIVE);
16     }
17 }
18
19 for (PowerUp pw : powerUps) {
20     if (pw instanceof ShrinkPowerUp) {
21         ((ShrinkPowerUp) pw).update(player);
22     }
23     if (Collision.VerifyCollision(pw, player)) {
24         pw.onCollected(player);
25     }
26 }
27
28 if (player.getState() == States.ACTIVE) {
29     for (Projectiles p : enemyProjectiles) {
30         if (Collision.VerifyCollision(p, player)) {
31             player.takeDamage(1);
32             p.setState(States.INACTIVE);
33         }
34     }
35     for (Enemy e : enemies) {
36         if (Collision.VerifyCollision(e, player)) {
37             player.takeDamage(1);
38         }
39     }
40     if (boss != null && boss.getState() == States.ACTIVE && Collision.VerifyCollision(boss, player)) {
41         player.takeDamage(1);

```

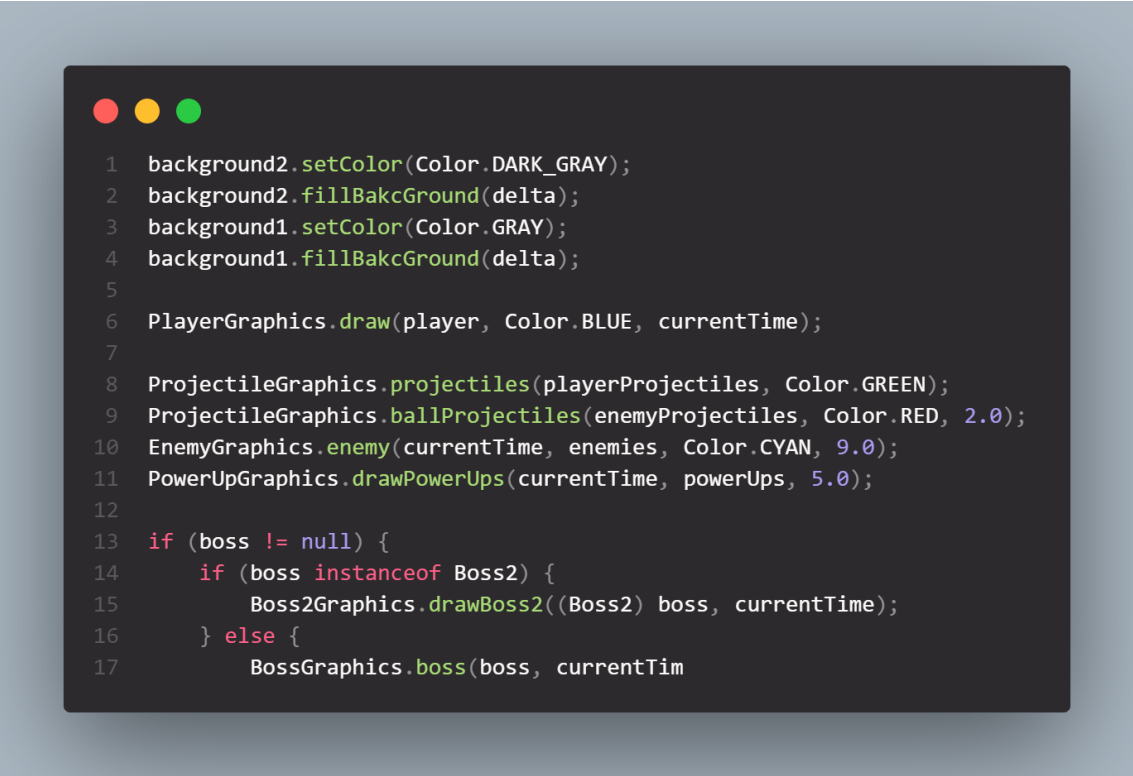
Figura 15: Detecção de Colisão

```

1  playerProjectiles.removeIf(p -> p.getState() == States.INACTIVE);
2  enemyProjectiles.removeIf(p -> p.getState() == States.INACTIVE);
3  enemies.removeIf(e -> e.getState() == States.INACTIVE);
4  powerUps.removeIf(e -> e.getState() == States.INACTIVE);
5

```

Figura 16: Remoção de Entidades



```
1 background2.setColor(Color.DARK_GRAY);
2 background2.fillBakcGround(delta);
3 background1.setColor(Color.GRAY);
4 background1.fillBakcGround(delta);
5
6 PlayerGraphics.draw(player, Color.BLUE, currentTime);
7
8 ProjectileGraphics.projectiles(playerProjectiles, Color.GREEN);
9 ProjectileGraphics.ballProjectiles(enemyProjectiles, Color.RED, 2.0);
10 EnemyGraphics.enemy(currentTime, enemies, Color.CYAN, 9.0);
11 PowerUpGraphics.drawPowerUps(currentTime, powerUps, 5.0);
12
13 if (boss != null) {
14     if (boss instanceof Boss2) {
15         Boss2Graphics.drawBoss2((Boss2) boss, currentTime);
16     } else {
17         BossGraphics.boss(boss, currentTim
```

Figura 17: Renderização