**Relatório COO**

**Críticas ao Código Original**

**1. Uso Excessivo de Código Duplicado**

O código original do jogo contém diversas instâncias de código duplicado, o que torna a manutenção e a extensibilidade do software mais difíceis. Exemplos de código duplicado incluem trechos responsáveis pela manipulação de entidades do jogo, como inimigos e projéteis. Esse tipo de prática não só aumenta o tamanho do código desnecessariamente, mas também aumenta a probabilidade de introduzir erros durante modificações ou atualizações, uma vez que uma mudança precisa ser replicada em múltiplos lugares.

**2. Ausência de Princípios de Orientação a Objetos**

Embora o código esteja escrito em Java, ele não segue os princípios básicos da orientação a objetos (OO). A estrutura atual não utiliza adequadamente conceitos como encapsulamento, herança e polimorfismo. Ao invés disso, o código segue um estilo procedural, onde a lógica do jogo está concentrada em poucas classes sem uma clara separação de responsabilidades. Isso contraria os princípios de OO, que promovem a modularidade e a reutilização de código.

**3. Falta de Encapsulamento**

No código original, muitos atributos e métodos que deveriam ser privados são públicos, expondo desnecessariamente os detalhes internos das classes. Isso viola o princípio do encapsulamento, que visa proteger os dados internos e apenas expor uma interface pública controlada. A falta de encapsulamento pode levar a um código mais vulnerável a erros e mais difícil de modificar ou estender de maneira segura.

**4. Gerenciamento Ineficiente de Entidades do Jogo**

O código original utiliza arrays para gerenciar múltiplas entidades do jogo, como inimigos e projéteis. Arrays são estruturas de dados estáticas que não se ajustam dinamicamente à quantidade de elementos que armazenam. Isso resulta em uma alocação de memória fixa, independentemente do número real de entidades ativas. Além disso, o código realiza uma reutilização manual dos índices dos arrays, o que pode levar a problemas de desempenho e vazamentos de memória se não for gerenciado corretamente. O uso de coleções dinâmicas do Java, como ArrayList e HashMap, seria mais apropriado para este cenário.

**5. Código Monolítico**

A maior parte da lógica do jogo está concentrada na classe Main.java, resultando em uma classe monolítica que faz tudo. Isso contraria o princípio da responsabilidade única, que sugere que cada classe deve ter uma única responsabilidade ou propósito bem definido. Um código monolítico é mais difícil de entender, testar e manter, e impede a reutilização de componentes individuais em outros projetos ou contextos.

**6. Falta de Documentação e Comentários**

O código original carece de comentários e documentação adequada, dificultando o entendimento e a manutenção por parte de novos desenvolvedores. Comentários explicativos e uma documentação clara são essenciais para garantir que o código seja compreensível e que futuras modificações possam ser feitas com confiança.

### Descrição e Justificativa para a Nova Estrutura de Classes/Interfaces

#### 1. Estrutura de Classes e Interfaces

##### **Classe** Main

**Descrição**: Centraliza a lógica principal do jogo.

**Justificativa**: Seguindo o princípio da responsabilidade única, a classe Main é responsável por iniciar, executar e encerrar o jogo, além de coordenar as interações entre os diferentes componentes do jogo.

##### **Interface** Entidade

**Descrição**: Define os métodos essenciais que todas as entidades do jogo (como inimigos, projéteis, e a nave do jogador) devem implementar.

**Justificativa**: Facilita a aplicação de polimorfismo, permitindo que diferentes tipos de entidades sejam tratados de maneira uniforme. Isso melhora a extensibilidade do código, tornando mais fácil adicionar novos tipos de entidades no futuro.

##### **Classe** Player

**Descrição**: Representa a nave controlada pelo jogador.

**Justificativa**: Encapsula todos os comportamentos e estados específicos do jogador, incluindo movimentação, tiro, e pontos de vida. Separar a lógica do jogador em uma classe própria melhora a clareza e a organização do código.

##### **Classe** Enemy **e Subclasses (**Enemy1**,** Enemy2**,** Enemy3**)**

**Descrição**: Representa os inimigos do jogo. Enemy1, Enemy2 e Enemy3 são subclasses que implementam comportamentos específicos.

**Justificativa**: Utilização de herança para definir comportamentos comuns na classe Enemy e comportamentos específicos nas subclasses. Isso reduz a duplicação de código e facilita a introdução de novos tipos de inimigos com comportamentos únicos.

##### **Classe** Projectile

**Descrição**: Representa os projéteis disparados pelo jogador e pelos inimigos.

**Justificativa**: Centraliza a lógica relacionada aos projéteis, facilitando a adição de novos tipos de projéteis ou modificações no comportamento dos mesmos.

##### **Interface** Powerup **e Classe** Hp

**Descrição**: Define o comportamento dos power-ups. Hp é uma implementação específica que aumenta a vida do jogador.

**Justificativa**: A utilização de uma interface permite a criação de diferentes tipos de power-ups com facilidade, promovendo a extensibilidade e modularidade do código.

##### **Classe** Background

**Descrição**: Gerencia a renderização do plano de fundo do jogo.

**Justificativa**: Centraliza a lógica de renderização do plano de fundo, melhorando a separação de preocupações e tornando o código mais modular e fácil de manter.

##### **Uso de Coleções Java**

**Descrição**: Substituição de arrays por coleções como ArrayList e HashMap para gerenciar entidades do jogo.

**Justificativa**: Coleções dinâmicas permitem uma gestão mais eficiente das entidades do jogo, eliminando a necessidade de reutilização manual de índices e prevenindo vazamentos de memória. Isso resulta em um código mais limpo, eficiente e menos propenso a erros.

#### 2. Justificativa das Mudanças

##### **Melhoria da Manutenibilidade e Extensibilidade**

A nova estrutura de classes e interfaces foi projetada para ser modular e extensível. Cada classe tem uma responsabilidade clara e bem definida, o que facilita a manutenção e a adição de novas funcionalidades. Por exemplo, novos tipos de inimigos ou power-ups podem ser adicionados facilmente sem a necessidade de grandes modificações no código existente.

##### **Redução de Código Duplicado**

Ao utilizar herança e interfaces, foi possível eliminar a duplicação de código. Métodos e atributos comuns foram movidos para classes base ou interfaces, permitindo que subclasses e implementações específicas se concentrem em comportamentos únicos.

##### **Aplicação de Princípios de Orientação a Objetos**

A nova estrutura adota plenamente os princípios de orientação a objetos, como encapsulamento, herança, polimorfismo e responsabilidade única. Isso não só melhora a organização e clareza do código, mas também facilita o entendimento e a colaboração entre os desenvolvedores.

##### **Melhoria no Gerenciamento de Entidades**

O uso de coleções dinâmicas do Java para gerenciar entidades do jogo, ao invés de arrays estáticos, resulta em um código mais eficiente e menos propenso a erros. Entidades podem ser adicionadas e removidas de maneira eficiente, e o código se torna mais flexível para lidar com variações no número de entidades.

##### **Facilidade na Implementação de Novas Funcionalidades**

A estrutura orientada a objetos facilita a implementação de novas funcionalidades. Por exemplo, a adição de novos tipos de inimigos e power-ups pode ser feita de maneira simples e direta, aproveitando a estrutura de classes existente e evitando a necessidade de grandes revisões no código.

### Descrição do Uso de Coleções Java para Substituir Arrays

#### Contexto Original

No código original do jogo, arrays foram utilizados extensivamente para gerenciar as entidades que ocorrem em multiplicidade, como inimigos e projéteis. Arrays, sendo estruturas de dados estáticas, requerem uma alocação de memória fixa, o que pode não refletir o número real de entidades ativas em qualquer momento. Além disso, a reutilização manual dos índices dos arrays para reciclar posições inativas torna o código mais complexo e propenso a erros.

#### Substituição por Coleções Dinâmicas

##### **Lista de Inimigos (**ArrayList<Enemy>**)**

**Uso Original**: Arrays eram utilizados para armazenar inimigos, com um tamanho fixo e reciclagem manual de índices.

**Nova Implementação**: Substituímos arrays por ArrayList<Enemy>, uma coleção dinâmica que permite a adição e remoção de elementos de maneira eficiente.

**Benefícios**: O ArrayList cresce automaticamente conforme necessário, eliminando a necessidade de gerenciamento manual de posições. Isso simplifica o código e melhora a eficiência na gestão dos inimigos.

##### **Lista de Projéteis (**ArrayList<Projectile>**)**

**Uso Original**: Arrays eram utilizados para armazenar projéteis, também com um tamanho fixo e reciclagem manual de índices.

**Nova Implementação**: Substituímos arrays por ArrayList<Projectile>, permitindo uma gestão mais flexível e eficiente dos projéteis.

**Benefícios**: A utilização de ArrayList facilita a adição e remoção de projéteis conforme eles são disparados ou saem da tela, sem a necessidade de controle manual de índices.

#### Remoção de Entidades Inativas

Com a substituição dos arrays por coleções dinâmicas, a remoção de entidades inativas se torna mais eficiente. Em vez de reutilizar manualmente os índices dos arrays, simplesmente removemos os objetos inativos das coleções.

### Descrição da Implementação das Novas Funcionalidades e Como o Código OO Ajudou

#### 1. Novo Tipo de Inimigo

##### Implementação

Adicionamos um novo tipo de inimigo, Enemy3, que apresenta comportamentos de ataque e movimentação diferentes dos inimigos existentes (Enemy1 e Enemy2). A nova classe Enemy3 herda da classe base Enemy, aproveitando os métodos e atributos comuns a todos os inimigos, mas implementa seu próprio comportamento específico nos métodos update e attack.

##### Benefícios do Código OO

**Herança**: Permitiu reutilizar código comum a todos os inimigos, evitando duplicação e facilitando a manutenção.

**Polimorfismo**: Facilitou a introdução do novo inimigo no jogo sem necessidade de modificar o código que gerencia os inimigos, pois Enemy3 pode ser tratado como qualquer outro Enemy.

#### 2. Criação de um Power-up

##### Implementação

Implementamos um novo tipo de power-up, Hp, que aumenta a vida do jogador ao ser coletado. A interface Powerup define o comportamento básico que todos os power-ups devem ter. A classe Hp implementa esta interface, definindo o comportamento específico de aumento de vida.

##### Benefícios do Código OO

**Interface**: A interface Powerup permitiu definir um contrato comum para todos os tipos de power-ups, facilitando a adição de novos tipos de power-ups no futuro.

**Encapsulamento**: Cada tipo de power-up encapsula seu comportamento específico, tornando o código mais modular e fácil de entender.

#### 3. Pontos de Vida para o Jogador

##### Implementação

Adicionamos um sistema de pontos de vida ao jogador, permitindo que a nave do jogador suporte múltiplos danos antes de ser destruída. A classe Player foi estendida para incluir um atributo hp e métodos para manipular os pontos de vida (increaseHp, decreaseHp).

Implementamos também uma barra de vida gráfica e um feedback visual quando o jogador é atingido, utilizando os métodos de renderização existentes em GameLib.

##### Benefícios do Código OO

**Encapsulamento**: A lógica de pontos de vida foi encapsulada na classe Player, tornando o código mais modular e facilitando a manutenção.

**Modularidade**: Facilita a implementação de novos sistemas relacionados à vida do jogador, como power-ups que aumentam a vida ou efeitos visuais de dano.

#### 4. Uso de Coleções Dinâmicas

##### Implementação

Substituímos arrays por coleções como ArrayList para gerenciar entidades do jogo (inimigos, projéteis, power-ups), o que simplificou o código e melhorou a eficiência na gestão de entidades dinâmicas.

##### Benefícios do Código OO

**Flexibilidade**: Coleções dinâmicas permitem adições e remoções eficientes de entidades, adaptando-se automaticamente ao número de elementos ativos.

**Manutenção Simplificada**: Reduz a complexidade de gerenciamento manual de índices, tornando o código mais claro e menos propenso a erros.

**Comentário sobre as classes.**

**Classe Background:** Código responsável por gerenciar e renderizar os elementos visuais de fundo que geram o ambiente do jogo, de modo que permitam um cenário dinâmico e com percepção de profundidade.

Atributos:

‘x’: posição horizontal do fundo;

‘y’: posição vertical do fundo;

‘speed’: velocidade com que o fundo se move;

‘layer’: camada do fundo, diferentes camadas podem ter diferentes velocidades, gera um efeito de paralaxe.

Construtor: inicia os atributos da classe com os valores fornecidos.

Métodos:

updateState: atualiza a posição do fundo com base no tempo decorrido.

Render: renderiza o fundo na tela, usa o GameLib.

**Classe Enemy:** Código responsável por gerenciar propriedades e comportamentos dos inimigos, em que são herdados pelos 3 inimígos. Além de incrementar características exclusivas de inimigos, ela herda a classe Entidade, a qual serve de base para todos os elementos do jogo.

Atributos:

‘x’ e ‘y’: coordenadas;

‘v’ e ‘rv’: velocidade do inimigo e velocidade de rotação respectivamente;

‘angle’: ângulo de movimento do inimigo;

‘state’ estado do inimigo;

‘radius’: raio do inimigo;

‘shoot’: próximo disparo;

‘exStart’: início da explosão;

‘exEnd’: término da explosão.

Construtor: Enemy, inicia os atributos com valores padrão.

Métodos:

Possui diversos métodos setters e getters para atributos, os quais precisam ser alterados ou retornados em alguma seção do código

**Classe Enemy1:** herda a classe Enemy, projetada para representar um tipo específico de inimigo. Possui um comportamento de movimentação e disparo na vertical. Não acrescenta novos atributos. Utiliza os métodos definidos em Enemy, além de acrescentar métodos para ser renderizado e atualizado. A classe é utilizada para criar instâncias de inimigos do tipo 1 ao usar arraylist.

Métodos:

updateState: atualiza a posição e estado do enemy, define seu padrão de movimentação, se está enquadrado na tela e se deve disparar.;

render: renderiza o inimigo na tela;

**Classe Enemy2**: herda a classe Enemy, projetada para representar um tipo específico de inimigo. Possui um comportamento de movimentação angular e disparos múltiplos e fragmentados, semelhantes aos de uma shotgun. Não acrescenta novos atributos. Utiliza os métodos definidos em Enemy, além de acrescentar métodos para ser renderizado e atualizado. A classe é utilizada para criar instâncias de inimigos do tipo 2 ao usar arraylist.

Métodos:

updateState: atualiza a posição e estado do enemy, define seu padrão de movimentação, se está enquadrado na tela e se deve disparar.;

render: renderiza o inimigo na tela;

**Classe Enemy3:** herda a classe Enemy, projetada para representar um tipo específico de inimigo. Possui um comportamento de movimentação em zig-zag, ao encostar no limite da tela pela direita passa a mover-se para a esquerda e assim segue invertendo até chegar ao fim da tela em baixo ou ser atingido. Os disparos são verticais. Não acrescenta novos atributos. Utiliza os métodos definidos em Enemy, além de acrescentar métodos para ser renderizado e atualizado. A classe é utilizada para criar instâncias de inimigos do tipo 3 ao usar arraylist.

Métodos:

updateState: atualiza a posição e estado do enemy, define seu padrão de movimentação, se está enquadrado na tela e se deve disparar.;

render: renderiza o inimigo na tela;

**Classe Entidade:** código que serve como base para todas as estruturas que interagem com o usuário, define as estruturas básicas do jogo, como coordenadas, tamanho e estado, além dos métodos para alterar e retornar todos. Classe que serve como uma facilitadora para a grande parte do projeto.

Atributos:

‘x’ e ‘y’: coordenadas;

‘state’: define o estado;

‘radius’: define o raio;

Métodos: possui apenas getters e setters para gerenciar as mudanças e retornos necessários.

**Classe Hp:** código responsável por gerenciar os pontos de vida de uma entidade, neste caso o player. É possível aumentar, reduzir e renderizar a barra de vida na tela através dos métodos da classe, de modo que permita uma fácil visualização do estado de saúde do jogador.

Atributos:

‘hp’: pontos de vida atuais da entidade;

‘inicialHp’: pontos de vida iniciais da entidade, também são o máximo possível de se obter.

Métodos:

reduceHP: reduz a vida do jogador em 1 ponto;

increaseHP: aumenta a vida do jogador em um ponto, respeitando o máximo;

renderHP: renderiza a barra de vida, em que verde é a vida restante e vermelho é a vida perdida.

Possui métodos getters e setters para auxiliar na execução de outros métodos.

**Classe Main:** parte do código que é o núcleo do jogo, gerencia inicializações, executa o loop principal, atualiza o estado de todos os objetos do jogo e renderiza todos os elementos visuais. Esta classe coordena todas as interações entre o player, inimígos, projéteis e outros elementos do jogo, de modo que a lógica do jogo funcione de maneira coesa e fluida.

Atributos:

‘INACTIVE’, ‘ACTIVE’, ‘EXPLODING’: Constantes que representam os estados das entidades no jogo;

‘running’: bool que indica se o jogo está em execução;

‘delta’: tempo entre frames;

‘currentTime’: tempo atual;

‘nextE1’, ‘nextE2’ e ‘nextE3’: tempo do próximo Spawn dos inimigos do tipo 1, 2 e 3;

‘e2Count’ e ‘e3Count’: Contadores para os Spawns dos inimigos 2 e 3;

‘e2SpawnX’ e ‘e3SpawnX’: Coordenadas de Spawn dos inimigos 2 e 3;

‘player’: instância do jogador;

‘background1’ e ‘background2’: instâncias das camadas de fundo do jogo;

‘powerup’: instância do power-up;

‘nextPowerupTime’: tempo do próximo Spawn de power-up;

‘projectiles’, ‘eprojectiles1’, ‘eprojectiles2’ e ‘eprojectiles3’: listas de projéteis do jogador e dos inimigos;

‘enemies1’, ‘enemies2’ e ‘enemies3’: listas de inimigos do tipo 1, 2 e 3.

Construtor: o construtor da classe Main inicializa todos os elementos do jogo, o que inclui o player, inimigos, projéteis, powerup e fundos.

Métodos:

updatePowerup: atualiza o estado do powerup, verifica se é hora de criar um novo ou se já acabou o efeito do atual;

launchNewEnemies: lança novos inimigos de acordo com os tempos definidos para cada inimigo;

busyWait: mantém a taxa de quadros constante;

gameLoop: loop principal do jogo, que gerencia a atualização de estados, verificação de colisçoes, renderizações e entradas do usuário;

render: renderiza todos os elementos do jogo na tela;

processInput: processa a entrada do usuário através do teclado e atualiza a posição e ação do player;

findFreeIndex: encontra um índice livre em uma lista de entidades;

main: ponto de entrada do programa e inicia o loop principal do jogo.

**Classe Player:** Representa o jogador no jogo, gerencia a posição, estado, vida, movimentos, disparos e outras interações do jogador com o ambiente de jogo. Herda a classe Entidade, aproveitando suas propriedades e métodos, além de adicionar comportamentos específicos para o jogador.

Atributos:

‘VX’ e ‘VY’: velocidades horizontal e vertical do jogador;

‘shoot’: próximo disparo;

‘exStart’: início da explosão;

‘exEnd’: término da explosão;

‘powerupEnabled’: indica se o power-up está ativo;

‘lastPowerupStartTime’: tempo de início do último power-up;

‘isFlashing’: indica se a animação de dano está ativa;

‘flashEndTime’: fim da animação de dano;

Construtor: inicializa os atributos do jogador.

Métodos: updateState: atualiza o estado do jogador, verificando se ele está explodindo ou piscando devido ao dano;

‘resetLastPowerupStartTime’: reseta o tempo do último power-up para o tempo atual;

‘Flash’: inicia o efeito de dano;

‘checkCollisions’: verifica colisões do player com projéteis, inimigos e power-up, ajustando o estado do jogador de acordo com a colisão;

‘render’: renderiza o jogador e seus efeitos na tela;

Também possui métodos getters e setters para operações com atributos relacionados ao power-up e elementos do jogo.

Classe Powerup: código responsável pelo sistema de power-up no jogo, item que o jogador coleta e recebe um bônus de vida e disparos por um determinado período de tempo, esta classe gerencia a posição, estado e renderização do power-up na tela.

Atributos:

‘x’ e ‘y’: coordenadas;

‘v’: velocidade;

‘state’: estado;

‘radius’: raio do power-up;

‘spawnTime’: tempo em que o power-up foi criado;

‘activeTime’: tempo em que o power-up fica ativo na tela.

Métodos:

Place: posiciona o power-up na tela e define seu estado como ativo;

updateState: atualiza o power-up;

render: renderiza o power-up;

Métodos setter e getters para facilitar operações.

Classe Projectile: código que representa projéteis no jogo, que são disparados pelo player e inimigos. Gerencia a movimentação e interação dos projéteis com os outros elementos do jogo

Atributos:

‘vx’ e ‘vy’: velocidades horizontal e vertical dos projéteis.

Métodos:

updateState: atualiza a posição do projétil com base no tempo decorrido;

renderP: renderiza o projétil do player na tela e muda o comportamento de disparo caso o power-up esteja ativo;

renderE: renderiza o projétil dos inimigos;

Também possui métodos getters e setters para facilitar operações.