UNIVERSIDADE DE SÃO PAULO
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

GUSTAVO POMPERMAYER FULANETTI SILVA - T94 - 14760280
JOÃO GABRIEL DE SENNA LAMOLHA - T94 - 14777879
KAUÊ PATRICK DE OLIVEIRA - T04 - 14586261
LUCAS STAMPINI YOKOYAMA - T04 - 14591740

UMA ABORDAGEM ORIENTADA A OBJETOS PARA UM JOGO SHOOT 'EM UP

SÃO PAULO
2024

SUMÁRIO

1 INTRODUÇÃO

A programação orientada a objetos (POO) tem se consolidado como uma metodologia eficaz para o desenvolvimento de sistemas complexos, devido à sua capacidade de organizar e estruturar o código de maneira modular e reutilizável. A POO permite a criação de objetos que encapsulam dados e comportamentos, promovendo a reutilização de código e facilitando a manutenção e a expansão dos sistemas. Este trabalho tem como objetivo apresentar uma abordagem orientada a objetos para a criação de um jogo do tipo shoot 'em up, demonstrando a transformação de um código procedural para um código orientado a objetos.

Inicialmente, o jogo foi desenvolvido utilizando uma abordagem procedural, onde as funções e dados são organizados de maneira linear e segmentada. Embora essa abordagem seja mais simples e direta, ela apresenta diversas limitações em termos de modularidade, reutilização de código e facilidade de manutenção. À medida que o projeto se torna mais complexo, essas limitações se tornam mais evidentes, tornando difícil adicionar novas funcionalidades ou modificar as existentes sem introduzir erros ou complicações desnecessárias.

A POO oferece uma maneira mais estruturada e eficiente de lidar com a complexidade dos sistemas de software. Neste contexto, a refatoração do código procedural para uma abordagem orientada a objetos foi realizada, com o objetivo de melhorar a organização, a legibilidade e a flexibilidade do código.

Este trabalho detalha o processo de transformação do código, destacando as principais diferenças entre as abordagens procedural e orientada a objetos, bem como os benefícios e desafios encontrados ao longo do processo.

2 CRÍTICAS AO CÓDIGO ORIGINAL

Para compreender as motivações por trás da transição de um código procedural para uma abordagem orientada a objetos, é fundamental analisar as limitações e desafios do código procedural original. A seguir, são apresentadas críticas detalhadas, destacando os principais problemas enfrentados com a estrutura procedural, que justificam a necessidade de uma refatoração para um código orientado a objetos.

2.1 Baixa Modularidade

A abordagem procedural organiza o código de forma linear, onde as funções são chamadas de maneira sequencial e os dados são muitas vezes manipulados de forma global. Essa estrutura resulta em baixa modularidade, dificultando a segmentação do código em unidades lógicas e independentes. Isso pode significar que alterações em um componente podem afetar outras partes do código, devido à falta de encapsulamento e separação de responsabilidades.

2.2 Reutilização de Código Limitada

A reutilização de código é um dos maiores desafios em uma abordagem procedural. Funções e procedimentos são frequentemente escritos para atender a um caso específico, o que impede seu uso em diferentes contextos sem modificações significativas. Isso pode levar à duplicação de código, onde lógicas semelhantes precisam ser reescritas para cada novo cenário, resultando em um aumento no tamanho do código e na dificuldade de manutenção.

2.3 Dificuldade na Manutenção

A manutenção do código procedural pode se tornar complexa e propensa a erros. A ausência de encapsulamento e a dependência de variáveis globais fazem com que mudanças em uma parte do código possam ter efeitos colaterais inesperados em outras partes. À medida que novas funcionalidades e melhorias são frequentemente adicionadas, essa interdependência pode levar a bugs difíceis de rastrear e corrigir, aumentando o tempo e o esforço necessário para a manutenção.

2.4 Escalabilidade Limitada

À medida que o software cresce em complexidade, o código procedural tende a se tornar menos escalável. A adição de novas funcionalidades pode exigir alterações significativas na estrutura existente, aumentando a complexidade e o risco de introdução de novos erros. Isso significa que a adição de novos recursos ou a adaptação a novos requisitos pode se tornar uma tarefa árdua, impedindo a evolução natural do software.

2.5 Escalabilidade Limitada

A legibilidade do código é essencial para que outros desenvolvedores possam entender e contribuir efetivamente para o projeto. No entanto, o código procedural, com sua estrutura linear e uso extensivo de variáveis e funções globais, pode ser difícil de ler e entender. Isso pode significar que novos desenvolvedores ou mesmo os desenvolvedores originais, após um período de afastamento, podem ter dificuldade em compreender o fluxo do código, os pontos de interação e a lógica por trás de determinadas implementações.

2.6 Possibilidade de Testes Reduzidos

O código procedural muitas vezes dificulta a criação de testes automatizados eficientes. A falta de modularidade e o uso de variáveis globais significam que testar uma função isoladamente pode ser complicado, pois ela pode depender do estado global do programa. Isso significa que a validação de novas funcionalidades ou a detecção de bugs pode se tornar um processo manual e demorado, dificultando a garantia de qualidade do software.

2.7 Flexibilidade Limitada

A flexibilidade do código é crucial para permitir adaptações e mudanças rápidas em resposta às novas demandas ou ideias criativas. No código procedural, a rigidez da estrutura linear e a interdependência das funções podem limitar a capacidade de adaptação rápida. Isso pode significar que a implementação de novas funcionalidades, ajustes no balanceamento ou alterações na interface do usuário podem exigir refatorações significativas, reduzindo a agilidade do desenvolvimento.

3 EXPLICANDO AS MUDANÇAS IMPLEMENTADAS PARA DESENVOLVER A ABORDAGEM ORIENTADA A OBJETOS

Esta seção detalha as mudanças implementadas, incluindo a introdução de interfaces, a utilização de composição para gerenciamento de atributos, a aplicação de herança para especialização de classes, e a escolha adequada de coleções Java, com o objetivo de melhorar a modularidade, reutilização, manutenção e escalabilidade do sistema.

3.1 Nova Estrutura de Classes e Interfaces

Para otimizar a organização e a manutenção do código, foi implementada uma nova estrutura de classes e interfaces baseada em princípios da programação orientada a objetos. Esta abordagem visa resolver os problemas identificados no código procedural original, promovendo modularidade, reutilização e manutenção eficiente. A seguir, detalhamos as justificativas e as implementações específicas adotadas.

3.1.1 Interfaces para Personagens e Projéteis

No contexto do jogo, a utilização de interfaces facilita a padronização e extensibilidade das funcionalidades dos personagens e projéteis, permitindo que diferentes tipos de entidades possam implementar comportamentos específicos de maneira independente. A interface “Personagem” é implementada pelas classes “Jogador” e “Inimigo”, enquanto a interface “Projetil” é implementada pelas classes “ProjetilJogador” e “ProjetilInimigo”.

A interface “Personagem” define métodos essenciais como “getX()” e “getY()”, que retornam as coordenadas X e Y do personagem, respectivamente. Além disso, métodos como “verificaFimExplosao()” e “verificaEstado()” permitem verificar o estado atual do personagem, como ativo, inativo ou explodindo, enquanto “movimenta()” e “desenha()” são responsáveis pelo movimento e renderização do personagem na tela.

Para os projéteis, a interface “Projetil” estabelece métodos similares como “getX()” e “getY()” para obter as coordenadas X e Y do projétil. Métodos como “verificaEstado()” verificam o estado do projétil, como ativo ou inativo, “movimenta()” define o comportamento de deslocamento do projétil, e “desenha()” cuida da renderização do projétil na tela.

Essas interfaces permitem que tanto personagens quanto projéteis implementem comportamentos específicos de movimento, verificação de estado e renderização de forma flexível e semelhante. Por exemplo, no caso dos inimigos, as classes “InimigoSimples” e “InimigoComposto” implementam “Personagem”,

enquanto “ProjetilJogador” e “ProjetilInimigo” implementam “Projetil”, garantindo uma estrutura modular e de fácil manutenção para o jogo.

3.1.2 Composição para Posição e Forma

Todas as classes que representam entidades (jogador, inimigos e projéteis) utilizam composição com as classes “Ponto2D” e “Forma”.

Desta forma, a utilização de composição para representar a posição e a forma das entidades no jogo é fundamental para o desenvolvimento de funcionalidades flexíveis e modulares. No código fornecido, a classe “Ponto2D” encapsula as coordenadas (x, y) e as velocidades (vX, vY) de uma entidade. Isso permite que qualquer objeto no jogo possua uma posição precisa e a capacidade de se mover de acordo com suas velocidades nos eixos X e Y.

A classe “Forma”, por sua vez, encapsula o raio, a cor e o formato (como círculo, jogador, diamante ou linha) de uma entidade. Utilizando composição, uma entidade pode ter sua aparência visual detalhadamente configurada sem necessidade de duplicação de código. Por exemplo, um inimigo pode ser representado por um círculo azul enquanto um projétil do inimigo pode ser um círculo vermelho, tudo isso definido através dos objetos da classe “Forma”.

Essa abordagem não só promove a reutilização de código, mas também facilita a manutenção e a expansão do jogo. Alterações na aparência visual de uma entidade podem ser feitas modificando apenas os parâmetros do objeto “Forma” associado, sem afetar outras funcionalidades do jogo.

Ao desenhar uma entidade na tela, o método “desenha()” da classe “Forma” utiliza as informações de cor e formato para renderizar a entidade usando a biblioteca gráfica “GameLib”. Isso garante consistência visual e permite que diferentes tipos de entidades sejam facilmente distinguíveis e reconhecíveis pelos jogadores.

Portanto, a combinação das classes “Ponto2D” e “Forma” através da composição oferece uma base sólida para a representação e manipulação de entidades no jogo, suportando tanto a movimentação quanto a renderização de forma eficiente e organizada.

3.1.3 Herança para Especialização de Inimigos

Na implementação do jogo, a especialização dos inimigos é realizada por meio das classes “InimigoSimples” e “InimigoComposto”, exemplificando o conceito de herança na programação orientada a objetos. A classe “InimigoSimples” representa um inimigo básico, enquanto “InimigoComposto” herda características de “InimigoSimples” e introduz comportamentos adicionais e específicos para um tipo mais complexo de inimigo.

Inicialmente, “InimigoSimples” é definido com uma posição aleatória na parte superior da tela e um formato circular com cor ciano. Ele se move verticalmente em direção ao jogador e dispara projéteis quando alinhado com a posição vertical do jogador. Este comportamento é encapsulado nos métodos “movimenta” e “atira”, onde o movimento é controlado pela alteração da posição baseada no ângulo definido e na velocidade vertical.

Por outro lado, “InimigoComposto” especializa-se ao adicionar movimentos horizontais alternados e capacidade de disparar múltiplos projéteis em diferentes direções. Ele herda o movimento básico de “InimigoSimples” e redefine o método “movimenta” para incluir um deslocamento horizontal entre limites específicos da tela. Além disso, utiliza um método especializado “atira” para disparar três projéteis em ângulos calculados dinamicamente, aumentando a complexidade do desafio para o jogador.

Essa estrutura de herança permite uma implementação eficiente e reutilizável, onde “InimigoComposto” expande e adapta o comportamento de “InimigoSimples” sem duplicação de código, promovendo uma melhor organização e manutenção do sistema de inimigos dentro do jogo. Dessa forma, cada classe de inimigo pode ser ajustada independentemente, facilitando a evolução e a expansão do jogo conforme necessário.

3.1.4 Classe Fundo Separada

A classe “Fundo” foi mantida separada das demais classes devido à sua funcionalidade distinta, focada exclusivamente na criação e manipulação do fundo do jogo. O fundo, composto por estrelas que se movem a uma velocidade constante, não interage diretamente com os personagens ou projéteis, justificando seu isolamento.

Na implementação, a classe “Fundo” utiliza uma “ArrayList” para armazenar as coordenadas X e Y das estrelas. A quantidade de estrelas, bem como suas posições iniciais, é definida aleatoriamente conforme o tamanho da tela, promovendo um visual dinâmico e envolvente. A velocidade de movimento do fundo é ajustável, e a cor das estrelas pode ser configurada, permitindo personalização.

O método “desenha” é responsável por atualizar e renderizar o fundo. Ele seleciona a cor apropriada, atualiza a contagem de acordo com a velocidade e o tempo decorrido, e desenha cada estrela em sua nova posição utilizando a biblioteca gráfica “GameLib”. Essa separação de responsabilidades facilita a manutenção e a expansão do código, permitindo alterações e melhorias no fundo sem impactar outras partes do sistema.

3.1.5 Classe “Jogo” para Gerenciamento

A classe Jogo foi criada para manipular as coleções Java e as demais classes, agrupando os métodos de verificação de estado, colisão, e outras funcionalidades essenciais. A main agora contém apenas o controle de tempo e a chamada desses métodos, mantendo a lógica de jogo separada e organizada.

Desta forma, a classe “Jogo” desempenha um papel central no gerenciamento das entidades e na lógica de jogo, coordenando a inicialização, atualização e renderização das principais componentes do jogo. Ela encapsula a complexidade do sistema por meio da manipulação de coleções de entidades e da delegação de responsabilidades para classes especializadas.

Na inicialização, a classe cria e configura os objetos dos fundos “distante” e “próximo”, cada um com suas próprias características visuais definidas pela

quantidade de estrelas e velocidade de movimento. Além disso, inicializa o jogador e define as coleções de inimigos e projéteis, utilizando estruturas adequadas como “LinkedList” para suportar inserções e remoções eficientes durante a execução do jogo.

A classe “Jogo” implementa métodos específicos para verificar colisões entre entidades, atualizar estados das entidades baseadas no tempo decorrido, e verificar a necessidade de criar novos inimigos e projéteis. Utiliza iteradores para percorrer as coleções de forma eficiente, removendo entidades inativas e permitindo a criação dinâmica de novas entidades conforme o progresso do jogo.

No método “desenhaEntidades”, a classe coordena o processo de renderização, chamando os métodos “desenha” de cada entidade para desenhar fundos, jogador, inimigos e projéteis na tela. Essa separação de responsabilidades facilita a manutenção do código e a extensão do jogo, mantendo a “main” livre de detalhes de implementação e focada apenas no controle de tempo e na chamada dos métodos essenciais do jogo.

Essa estruturação da classe “Jogo” contribui significativamente para a organização e escalabilidade do código, seguindo os princípios da programação orientada a objetos e garantindo um desenvolvimento mais modular e eficiente do jogo.

3.2 Uso de Coleções Java:

3.2.1 Organização das Entidades

No desenvolvimento deste jogo, o uso estratégico de coleções Java desempenha um papel crucial na organização e manipulação eficiente das entidades do jogo. As coleções escolhidas (“LinkedList” e “ArrayList”) são fundamentais para gerenciar dinamicamente entidades como inimigos e projéteis, mantendo uma estrutura flexível que permite fácil adição, remoção e iteração sobre os elementos.

3.2.2 Organização dos Inimigos

As coleções “LinkedList” foram escolhidas para armazenar tanto os inimigos quanto os projéteis. Para os inimigos, temos duas categorias principais: “InimigoSimples” e “InimigoComposto”, cada um representado por uma LinkedList separada (“inimigosTipo1” e “inimigosTipo2”, respectivamente). Essa escolha permite uma gestão eficiente dos inimigos, adicionando novos inimigos ao final da lista quando necessário e removendo-os quando ficam inativos, otimizando assim o processamento durante a verificação de estados e colisões.

3.2.3 Organização dos Projéteis

Os projéteis, tanto do jogador quanto dos inimigos, são gerenciados pelas LinkedLists “projeteisJogador” e “projeteisInimigos”. Essas coleções são ideais para armazenar objetos que são frequentemente adicionados e removidos, como projéteis que saem da tela ou colidem com outros objetos. A utilização de LinkedLists permite iterar sobre os elementos de forma eficiente durante a verificação de colisões e atualização de estados.

3.2.4 Iteração e Remoção de Elementos

Durante a verificação de estados e colisões, são utilizados iteradores para percorrer as “LinkedLists” de inimigos e projéteis. Iteradores são essenciais para percorrer coleções e realizar operações seguras de adição/remoção de elementos durante a iteração. Isso é especialmente importante ao remover inimigos ou projéteis que se tornaram inativos, garantindo a consistência dos dados e evitando exceções de concorrência.

3.3 Funcionalidades Extras Implementadas

3.3.1 Pontos de Vida

A classe “Main” solicita ao usuário que informe a quantidade inicial de pontos de vida do jogador através da entrada padrão. Esses pontos de vida são então passados como parâmetro para o construtor da classe “Jogador”, onde são definidos tanto os pontos de vida máximos quanto os atuais do jogador.

Durante o jogo, sempre que ocorre uma colisão entre o jogador e um projétil inimigo, é subtraído um ponto de vida do jogador. Além disso, o estado do jogador é alterado para "atingido", o que geralmente é usado para modificar visualmente o jogador (como mudar a cor temporariamente) para indicar que ele sofreu dano. Para evitar danos consecutivos da mesma colisão ou de colisões próximas que poderiam prejudicar a experiência de jogo, o jogador se torna invulnerável por um curto período após ser atingido.

Após reduzir os pontos de vida, verifica-se se a vida atual do jogador não foi zerada. Se ainda houver pontos de vida restantes, é iniciada a animação de explosão, semelhante ao funcionamento na versão procedural do jogo. Durante a explosão ou ao fim do período de invulnerabilidade, a cor original do jogador é restaurada e, ao término da explosão, os pontos de vida são reiniciados.

3.3.2 Power UP

Power UP é uma classe do pacote entidade que tem como objetivo melhorar as habilidades do jogador por um curto período de tempo, caso seja tocado.Foi inserido também um Enum em “entidades.enums” chamado **“SOB EFEITO”**, que é responsável por indicar que o personagem principal do mini-game está sob efeito das mudanças que o Power UP pode causar.

Os efeitos escolhidos por nós foram: acréscimo de 0.08 na velocidade vertical e horizontal, mudança de cor para a mesma cor do power up “MAGENTA” e além disso, o jogador também não sofrerá nenhum tipo de dano e os projéteis / inimigos não irão colidir com ele por alguns segundos.

3.3.3 Inimigo Novo

O inimigo tipo 3 tem seu formato em círculo e coloração amarela, é uma classe que herda funcionalidades do “InimigoSimples” e dispara em modo “Burst” com 3 projéteis por vez. Além disso, seu movimento é vertical e horizontal: movimenta-se verticalmente até certo ponto definido por nós “um pouco acima da

metade da tela” e inicia assim um movimento horizontal de bate e volta, até ser atingido pelo projétil do player.