Relatório - Projeto de Computação Orientada à Objetos

Higor Freitas
Sistemas de Informação
USP EACH
São Paulo, Brasil
NUSP 15575879

I. Introdução

Este documento apresenta o desenvolvimento de um Exercício-Programa (EP) da disciplina de **Computação Orientada a Objetos (COO)**, ministrada pelo professor **Flávio Luiz Coutinho**, no curso de Sistemas de Informação da EACH-USP. O objetivo do trabalho é aplicar, na prática, os conceitos de orientação a objetos por meio da refatoração e extensão de um jogo inicialmente estruturado de forma procedural.

Embora o projeto fosse proposto para grupos de até quatro pessoas, este trabalho foi desenvolvido por mim individualmente, já que não fui capaz de encontrar grupos disponíveis.

A nova implementação utiliza herança, polimorfismo e encapsulamento para organizar o código em pacotes e classes com responsabilidades claras. Com isso, foi possível implementar chefes (*bosses*), fases baseadas em arquivos, power-ups e um sistema de vidas, de forma modular e extensível, demonstrando os benefícios da orientação a objetos no desenvolvimento de software.

II. Críticas ao código original do jogo

O código fornecido representa um projeto monolítico e procedural que desrespeita princípios fundamentais de engenharia de software aprendidos em aula, especialmente ao utilizar Java, uma linguagem que sabemos ser fortemente orientada a objetos. Abaixo discutimos os principais problemas de design, com foco na ausência de abstração, encapsulamento e modularidade.

a) Ausência de Encapsulamento e Coesão: Todas as entidades do jogo são representadas por vetores paralelos: posições, velocidades, estados, ângulos, etc. Isso viola completamente o princípio de coesão, pois cada inimigo, projétil ou jogador deveria ser um objeto com atributos e métodos próprios. No estado atual, o código é altamente acoplado à estrutura de arrays e exige que todos os aspectos de uma entidade estejam sincronizados manualmente, uma fonte comum de bugs.

b) Dificuldade de Incremento e Evolução: Adicionar um novo tipo de inimigo, projétil ou efeito exige a criação de múltiplos arrays, além de modificações em diversas seções do código: colisões, atualizações, renderização e lógica de spawn. Isso quebra o princípio da abertura/fechamento (Open/Closed Principle), já que cada extensão exige modificar o código existente em vez de apenas adicionar novas classes.

c) Repetição de Código e Falta de Polimorfismo: O código contém trechos duplicados para diferentes tipos de inimigos e projéteis, mudando apenas os nomes das variáveis. Com orientação a objetos, isso seria abstraído com herança ou interfaces, permitindo o uso de polimorfismo para generalizar comportamentos comuns em métodos como `atualizar()` e `desenhar()`. A ausência dessa abstração aumenta o tamanho do código, sua complexidade e dificulta testes.

d) Violação do Princípio da Responsabilidade Única: O método `main` realiza todas as tarefas do jogo: entrada do usuário, física, detecção de colisões, lógica de spawn, renderização e controle de tempo. Isso torna o código ilegível e difícil de manter. Uma arquitetura orientada a objetos permitiria distribuir essas responsabilidades entre múltiplas classes especializadas (por exemplo, `ControladorDeJogo`, `GerenciadorDeColisao`, `Entidade`, etc.).

e) Design Frágil e Não Escalável: O modelo atual não escala bem. A simples adição de um power-up temporário exigiria duplicar a lógica de gerenciamento de estado e colisão. O código é extremamente rígido, e mudanças pequenas podem introduzir efeitos colaterais indesejados. Um sistema baseado em objetos permitiria reutilização de lógica comum e isolar mudanças específicas.

Ou seja, embora o código funcione como um exercício introdutório, sua estrutura procedural limita seriamente qualquer tentativa de evolução, modularidade ou manutenção a longo prazo. Refatorá-lo para um modelo orientado a objetos com classes como `Player`, `Inimigo`, `Projetil` e `Fase` é essencial para garantir facilidade na manutenção, clareza e possibilidade real de expansão. A arquitetura atual, baseada em arrays paralelos e lógica duplicada, é um contra exemplo para qualquer projeto que deseje crescer além de um protótipo básico.

III. Justificativa para a nova estrutura de classes/interfaces adotada.

A nova estrutura do projeto foi redesenhada com base em princípios de orientação a objetos,com modularidade e reutilização. A arquitetura foi dividida em pacotes lógicos e classes especializadas, com uma hierarquia de herança de classes abstratas que organiza entidades com comportamentos similares facilita reutilização do código. A figura a na página seguinte ilustra essa estrutura.

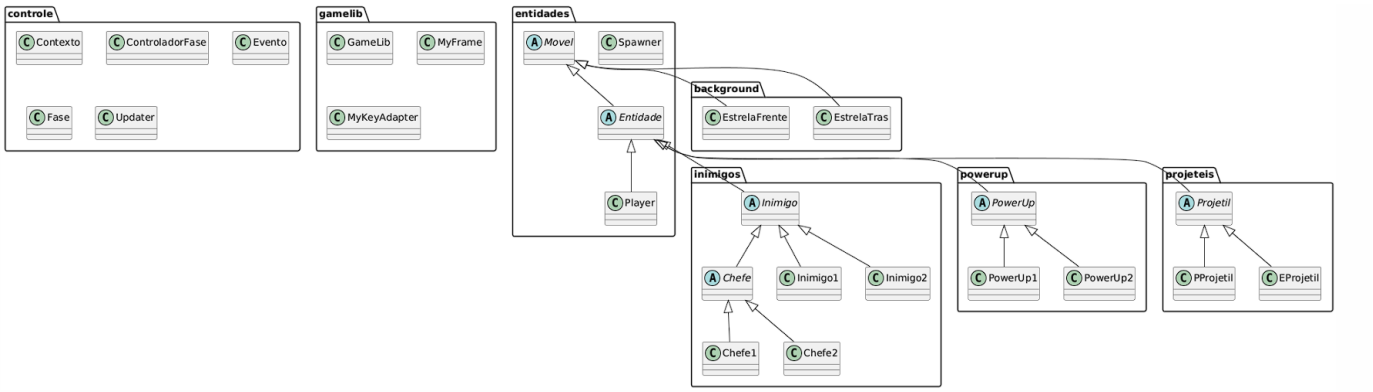

Fig. 1. Diagrama de classes e pacotes do projeto

a) Hierarquia Baseada em `Movel`: No topo da hierarquia está a classe abstrata `Movel`, que encapsula atributos comuns como posição, velocidade e métodos de movimentação. A partir dela, derivam todas as entidades do jogo com comportamento dinâmico. A classe `Entidade` estende `Movel` e acrescenta funcionalidades de colisão, raio de impacto e tempo de vida, isso forma a base para objetos com interação na janela do jogo.

b) Especializações de Entidade: A partir de `Entidade`, o jogo define diferentes tipos de objetos:

- **Player:** o jogador controlável, com lógica de entrada, disparo, vida etc...
- **Inimigo:** entidade hostil com métodos de movimento e ataque autônomos.
- **Projetil:** representa tiros no jogo, com subclasses `PProjetil` (do jogador) e `EProjetil` (dos inimigos).
- **PowerUp:** encapsula bônus temporários que interagem com o jogador.

Especializações de Movel:

- **Background:** as estrelas (`EstrelaFrente`, `EstrelaTras`) herdam diretamente de `Movel`, já que elas só precisam de posicionamento e velocidade.

c) Especializações Concretas e Reuso por Herança: As classes com sufixos numéricos (como `Inimigo1`, `Inimigo2`, `PowerUp1`, `PowerUp2`) são variações específicas das suas superclasses. O mesmo vale para `Chefe1` e `Chefe2`, que herdam de `Chefe`, que por sua vez herda de `Player`. Esse padrão permite reutilizar lógicas comuns e apenas sobrescrever o necessário, promovendo extensão com mínimo acoplamento.

d) Organização por Pacotes:

- `controle`: gerencia estado global (`Contexto`), fases (`Fase`, `Evento`), a lógica de execução (`ControladorFase`) e as atualizações (`Updater`).
- `entidades`: define a base do sistema de objetos móveis e sua hierarquia.
- `inimigos`, `powerup`, `projeteis`: agrupam as especializações de cada categoria de entidade, mantendo clareza e modularidade.
- `background`: isola o comportamento visual de fundo, mantendo separação de responsabilidades.
- `gamelib`: armazena a biblioteca gráfica auxiliar e suas classes de suporte.

Essa estrutura orientada a objetos favorece a reutilização, legibilidade e facilidade de expansão. A separação clara de responsabilidades, aliada ao uso de herança e polimorfismo, permite adicionar novos comportamentos ou entidades sem alterar o código existente, apenas estendendo classes e sobrescrevendo métodos. Além disso, a organização por pacotes facilitaria a navegação e manutenção em projetos de médio e grande porte. Nesse caso, seria extremamente fácil adicionar novos Inimigos, Fases, Bosses, Projéteis.

IV. Utilização das Coleções Java para substituir os Arrays

Para melhorar a arquitetura do jogo e substituir o uso extensivo de arrays paralelos, foi introduzida a classe `Contexto`, que funciona como um repositório e gerenciador centralizado de entidades do jogo. Essa reformulação aproveita as coleções da linguagem Java, como `ArrayList`, para armazenar e manipular dinamicamente objetos de forma segura, extensível e orientada a objetos.

a) Uso de `ArrayList` para Entidades: As coleções foram utilizadas para representar grupos de objetos como projéteis, inimigos e efeitos visuais. Por exemplo, ao invés de manter múltiplos arrays paralelos (`enemy1_X`, `enemy1_Y`, `enemy1_states`), agora existe uma lista de instâncias da classe `Inimigo`, cada uma contendo seus próprios atributos de posição, estado e comportamento encapsulados. Isso melhora drasticamente a legibilidade e evita problemas de sincronização de índice.

b) Encapsulamento e Tipagem Forte: O uso de listas fortemente tipadas (como `List<Projetil>`) garante segurança em tempo de compilação, facilita refatorações e melhora a navegabilidade do código. Além disso, o encapsulamento permite que cada objeto controle sua própria lógica de atualização, colisão e renderização, promovendo a separação de responsabilidades.

c) Flexibilidade e Expansibilidade: Ao usar coleções dinâmicas, o jogo não está mais limitado por um número fixo de entidades (como `new int[10]`). Isso facilita a introdução de novas entidades em tempo de execução, suporte a power-ups temporários, sistemas de partículas e múltiplos inimigos sem a necessidade de redimensionamento manual.

d) Centralização e Encapsulamento com a classe `Contexto`: A classe `Contexto` age como um container de todas as coleções e do estado global do jogo, permitindo que diferentes partes do sistema (como controladores de fase ou renderizadores do GameLib) tenham acesso consistente e seguro aos elementos do jogo. Além disso, a classe oferece métodos como `addProjetil()`, `addInimigo()`, e `clear()` que abstraem a manipulação direta das listas, encapsulando as regras de inserção e limpeza (remoção de objetos inativos). Isso reduz o acoplamento e melhora a modularidade do sistema.

No geral, a transição do uso de arrays para coleções genéricas foi essencial para transformar o código em um projeto modular e escalável. Essa mudança reduziu a complexidade estrutural, melhorou a manutenção e abriu caminho para futuras extensões do jogo. A utilização adequada de coleções Java, em conjunto com princípios de orientação a objetos e encapsulamento via métodos utilitários na classe `Contexto`, resultou em uma base de código mais limpa, flexível e preparada para evolução.

V. Como a orientação à objetos facilitou as novas Implementações

a) Sistema de Fases Baseado em Arquivos: O sistema de fases foi projetado para permitir que a progressão do jogo fosse definida externamente, sem necessidade de modificar o código-fonte. Para isso, foi criada a classe `ControladorFase`, que lê e interpreta eventos definidos nos arquivos `config.txt`, `fase1.txt`, `fase2.txt`. Esses arquivos definem os momentos de entrada dos inimigos, suas posições, tipos e comportamentos. Cada evento é instanciado como um objeto da classe `Evento`, encapsulando os dados da ação a ser executada. A lógica de leitura sequencial e disparo de eventos está encapsulada na classe `Fase`. Ou seja, o `ControladorFase` possui um Array `Fase`, que por sua vez possuem vários `Eventos`. Isso ajudou o código a ficar limpo e desacoplado da lógica principal.

b) Implementação de Chefes: Os chefes foram implementados como subclasses específicas da classe abstrata `Chefe`, que por sua vez estende `Inimigo`. Essa classe intermediária define comportamentos e estruturas comuns a todos os chefes, como padrões de vários disparos, a barra de vida, um movimento específico que não sai da tela etc... Através da herança, classes como `Chefe1` e `Chefe2` herdam essa lógica básica e podem sobrescrevem métodos como `update()` e `atirar()` para implementar seus próprios desenhos e ataques específicos. Essa abordagem modular e extensível permitiu adicionar comportamentos complexos sem afetar a estrutura de inimigos comuns nem duplicar código.

c) Sistema de Power-Ups: Os power-ups foram estruturados como objetos derivados da classe `PowerUp`, que também herda de `Entidade`. Cada tipo de power-up, como `PowerUp1` ou `PowerUp2`, possuem um método `aplicar()` e `remover()` com comportamento específico que manipula o objeto do `Player`, alterando os atributos dele. O `PowerUp1` dobra a velocidade do jogador, que facilita desviar de projéteis e o `PowerUp2` dobra a cadência de tiros. A orientação a objetos permitiu que o jogador pudesse verificar colisões com qualquer power-up e simplesmente invocar o método genérico deles, confiando no polimorfismo para executar o efeito correto, sem condicional extra no jogador.

d) Sistema de Vida e Gerenciamento de Estado: Também com relação ao jogador, ele agora possui múltiplas vidas, implementadas como um contador encapsulado dentro da classe `Player`. Quando ocorre uma colisão com inimigos ou projéteis, o jogador entra em estado de explosão, e uma vida é subtraída; em seguida ele renasce com um tempo de invencibilidade. A lógica de renascimento, exibição visual e término de jogo está separada da lógica de movimentação, mantendo o código limpo e coeso. O encapsulamento da vida permitiria futuras melhorias como barras de HP difemretes ou power-ups de cura sem afetar o restante do código.

VI. Conclusão geral

Em resumo, a arquitetura orientada a objetos foi essencial para o sucesso da implementação incremental dessas funcionalidades. Ao invés de criar blocos de código duplicados ou inserir verificações por tipo espalhadas pelo sistema, bastou criar novas classes que estendem comportamentos existentes. A separação de responsabilidades entre classes como `Contexto`, `Fase`, `Player`, `Inimigo` e `PowerUp` permitiu que as novas funcionalidades fossem adicionadas de forma localizada, com impacto mínimo nas classes já existentes. O uso de herança, encapsulamento e polimorfismo permitiu que o jogo evoluísse naturalmente com novas funcionalidades sem comprometer sua base. O design orientado a objetos proporcionou um ambiente em que cada componente tem seu papel claro e pode ser estendido ou modificado com facilidade, o que seria praticamente inviável em um projeto procedural como antes.