UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES, CIÊNCIAS e HUMANIDADES

GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Lucas Gurgel do Amaral - 14760234

Pedro Henrique dos Santos Matos – 13684915

Aaron Ferraz do Amaral Martins da Silva - 14554474

Paulo Ubiratan Muniz Rodrigues - 14748678

Relatório do EP da matéria de Computação Orientada a Objetos

São Paulo

2024

SUMÁRIO

1 INTRODUÇÃO

Este relatório foi desenvolvido como parte do projeto da disciplina de Computação Orientada a Objetos, oferecida pelo curso de Graduação em Sistemas de Informação da Escola de Artes, Ciências e Humanidades da Universidade de São Paulo. O trabalho foi realizado pelo grupo formado por Lucas Gurgel do Amaral, Pedro Henrique dos Santos Matos, Aaron Ferraz do Amaral Martins da Silva e Paulo Ubiratan Muniz Rodrigues.

O objetivo deste relatório é analisar criticamente o código-fonte original de um jogo desenvolvido em Java, identificar suas deficiências e propor melhorias para aprimorar a qualidade, eficiência e manutenção do código. Além disso, detalhamos as mudanças realizadas na estrutura do código para torná-lo mais modular e orientado a objetos, apresentando uma justificativa para cada alteração e os benefícios obtidos com essas modificações.

2 RESUMO

O relatório começa com uma análise crítica do código original do jogo, identificando problemas como a estrutura monolítica, o uso de arrays para representar entidades, a falta de encapsulamento, a extensão excessiva do método main, o uso de constantes numéricas "mágicas", a função busyWait ineficiente e a lógica de processamento de entrada do usuário misturada no loop principal. Para cada problema identificado, são sugeridas soluções que promovem a modularidade, a reutilização de código e a clareza.

Em seguida, o relatório descreve a nova estrutura de classes e interfaces adotada, que visa a modularidade e a flexibilidade. A nova estrutura utiliza interfaces e classes base para encapsular comportamentos comuns e permite a extensão fácil de novas funcionalidades. Também são destacadas as melhorias na gestão de estados e propriedades das entidades do jogo, substituindo o uso de arrays por coleções Java como List e ArrayList, proporcionando maior flexibilidade e facilidade de manutenção.

O relatório detalha como as novas funcionalidades foram implementadas e como a orientação a objetos ajudou nesse processo. São apresentados exemplos de código que ilustram a criação e gestão de itens de bônus, a renderização do fundo do jogo e a verificação de colisões, demonstrando as vantagens do encapsulamento, herança e polimorfismo.

Por fim, a conclusão reforça os benefícios da reestruturação do código, que resultou em maior flexibilidade, facilidade de manutenção e possibilidade de expansão futura. A utilização de coleções Java e a adoção de práticas de programação orientada a objetos foram fundamentais para alcançar esses resultados, permitindo uma gestão mais eficiente e dinâmica dos objetos do jogo e facilitando a implementação de novas funcionalidades.

3 CRÍTICAS AO CÓDIGO ORIGINAL DO JOGO

3.1 VISÃO GERAL

O arquivo **Main.java** contém o código-fonte principal de um jogo desenvolvido em Java. Este relatório examina detalhadamente o código, identifica suas deficiências e sugere melhorias para aprimorar a qualidade, a eficiência e a manutenção do código.

3.1.1 Estrutura Monolítica

Problema: O código é altamente monolítico, com toda a lógica do jogo, incluindo inicialização, atualização de estados e verificação de colisões, implementada dentro de um único arquivo e função main.

Impacto:

- Dificuldade na manutenção do código.
- Dificuldade na leitura e compreensão do código.
- Impossibilidade de reutilização de componentes.

Solução Sugerida:

- Modularizar o código separando as responsabilidades em diferentes classes e métodos. Por exemplo, criar classes específicas para Player, Projectile, Enemy, GameLoop, etc.

3.1.2 Uso de Arrays para Representar Entidades do Jogo

Problema: O código utiliza arrays para representar estados e propriedades de diferentes entidades do jogo (jogador, projéteis, inimigos, etc.).

Impacto:

- Dificulta a extensão do jogo com novos tipos de entidades.
- Torna o código mais suscetível a erros de índice fora do limite.
- Reduz a clareza e a organização do código.

Solução Sugerida:

- Utilizar classes para encapsular o estado e o comportamento das entidades.
- Utilizar coleções como **ArrayList** para gerenciar entidades de forma dinâmica.

3.1.3 Falta de Encapsulamento

Problema: Todas as variáveis de estado do jogo são definidas como variáveis globais dentro do método main.

Impacto:

- Quebra do princípio de encapsulamento.
- Aumento da complexidade ao rastrear o estado do jogo.
- Dificuldade em isolar e corrigir bugs.

Solução Sugerida:

- Encapsular variáveis de estado dentro de classes apropriadas.
- Utilizar métodos get e set para acessar e modificar estados.

3.1.4 Método main Extensivamente Longo

Problema: O método main é extremamente longo e complexo, com várias responsabilidades misturadas.

Impacto:

- Código difícil de entender e manter.
- Dificuldade em testar partes específicas do código.

Solução Sugerida:

- Dividir o método main em métodos menores e mais coesos.
- Criar um método separado para inicialização, atualização, verificação de colisões e renderização.

3.1.5 Uso de Constantes Numéricas

Problema: O código utiliza muitas constantes numéricas "mágicas" diretamente (por exemplo, 0.25, 0.8, 500).

Impacto:

- Reduz a legibilidade e a compreensão do código.

- Dificulta a manutenção e atualização de valores constantes.

Solução Sugerida:

- Definir constantes nomeadas para substituir valores numéricos mágicos.
- Utilizar enum para representar estados de entidades.

3.1.6 Função busyWait

Problema: A função busyWait é utilizada para esperar um determinado período de tempo.

Impacto:

- Ineficiente, pois consome recursos do processador enquanto espera.
- Pode causar problemas de desempenho, especialmente em sistemas multitarefa.

Solução Sugerida:

- Utilizar métodos mais eficientes para pausas, como Thread.sleep.

3.1.7 Processamento de Entrada do Usuário

Problema: A lógica de processamento de entrada do usuário está misturada com outras lógicas dentro do loop principal.

Impacto:

- Dificulta a compreensão e a modificação da lógica de entrada do usuário.
- Torna o loop principal mais complexo.

Solução Sugerida:

- Extrair a lógica de processamento de entrada do usuário para um método separado.

4 DESCRIÇÃO E JUSTIFICATIVA PARA A NOVA ESTRUTURA DE CLASSES/INTERFACES ADOTADA

4.1 ESTRUTURA ORIGINAL

O código original em **Main.java** é monolítico e utiliza arrays para gerenciar o estado dos objetos do jogo (player, projéteis, inimigos, etc.). Este tipo de estrutura apresenta dificuldades na manutenção, expansão e reutilização do código devido à sua baixa modularidade e alta interdependência entre as partes.

4.2 ESTRUTURA REESTRUTURADA

A nova estrutura de classes e interfaces foi organizada para promover a modularidade, reuso e flexibilidade. As principais alterações incluem a criação de classes específicas para cada tipo de entidade do jogo e a utilização de coleções Java para gerenciar grupos de objetos, substituindo o uso de arrays. A nova estrutura é a seguinte:

1. GameItem: Interface base para todos os itens do jogo. Ela obriga que todos os itens do jogo tenham os métodos render() e update() para o funcionamento do jogo.

1.1. Entity: Classe base para todas as entidades do jogo, contendo propriedades comuns como posição e estado, além da implementação padrão para os métodos principais (update() e render()).

1.1.1. BackgroundStar: Estrela do fundo.

1.1.2. BonusItem: Classe abstrata base para o Item bonus, responsável por oferecer um bonus ao jogador ao ser colidida. Não tem renderização.

1.1.2.1. BonusItemShield: Classe para gerar um item bonus com o efeito INVULNERABLE e com renderização particular.

1.1.3. CollidableEntity: Classe abstrata para fornecer atributos e métodos para entidades colidíveis, que tem pontos de vida, podem receber dano e explodir.

1.1.3.1. Enemy: Classe base para todos os inimigos, permite ajustar parametros de velocidade linear ou angular, além de permitir que os inimigos atirem projéteis.

1.1.3.1.1. BossEnemy: Inimigo mais poderoso e com mais pontos de vida, com renderização de barra de vida no topo da tela.

1.1.3.1.1.1. BossEnemyState: Enum para armazenar os estados do Boss, como movimentação inicial, momento para atirar, e saindo da tela.

1.1.3.1.2. DefaultEnemy: Inimigo padrão, movimentação apenas no eixo Y.

1.1.3.1.3. StrongerEnemy: Inimigo forte, com movimentação tanto no eixo X quanto no eixo Y.

1.1.3.2. Player: Classe do Player, permite que ele atire projéteis, além de se movimentar através das teclas do teclado e receber efeitos bonus (como o estado invulnerável).

1.1.4. Projectile: Projétil que pode ser lançado pelas entidades.

1.2. Background: Responsável por renderizar o fundo e as estrelas.

1.3. Timer: Responsável por renderizar um temporizador.

2. EntityState: Enum para os diferentes tipos de estado das entidades (exemplo: ACTIVE, INACTIVE).

3. BonusType: Enum para os diferentes tipos de bonus do jogador (exemplo: INVULNERABLE).

4. BonusEffect: Classe modelo para agrupar os atributos do efeito bonus.

5. Constants: Classe para armazenar todas as constantes do projeto, podendo ser alteradas para alterar o comportamento do jogo.

6. EnemiesManager: Classe para retornar os inimigos do jogo com base nos temporizadores de spawn definidos.

7. BonusItemsManager: Classe para retornar os items bonus do jogo com base nos temporizadores de spawn definidos.

8. ShootingCallback: Interface implementada pela classe principal do jogo para possibilitar que inimigos e players possam atirar e serem ouvidos através do método shoot().

9. GameManager: Classe principal e responsável por criar e armazenar todos os itens do jogo, fazer os updates, renderizá-los, iniciar/pausar o loop do jogo, etc.

10. GameLib: Biblioteca de renderização do jogo, que foi atualizada para possibilitar outras formas e visualizações, como a tela de Game Over, barra de vida do jogador e do Boss, etc.

4.3 JUSTIFICATIVA

A nova estrutura facilita a manutenção e a adição de novas funcionalidades, pois cada classe tem responsabilidades bem definidas. Isso também torna o código mais legível e testável. A utilização de herança e composição permite reaproveitar código e adicionar comportamentos específicos sem duplicação.

5 DESCRIÇÃO DE COMO AS COLEÇÕES JAVA FORAM UTILIZADAS PARA SUBSTITUIR O USO DE ARRAYS

5.1 UTILIZAÇÃO DE COLEÇÕES JAVA PARA SUBSTITUIR ARRAYS

Na estrutura original, arrays eram usados extensivamente para gerenciar estados e posições de múltiplos objetos, o que torna o código propenso a erros e de difícil manutenção. Na nova estrutura, coleções Java, como **List**, foram utilizadas para substituir arrays, proporcionando maior flexibilidade e métodos convenientes para manipulação de dados.

Exemplos de Substituição:

Gestão de Itens de Bônus:

1. Código:

```
public List<BonusItem> getBonusItemsToSpawn(long currentTime){

List<BonusItem> bonusItemsToSpawn = new ArrayList<>();

if (currentTime > nextBonusItemInterval) {

    bonusItemsToSpawn.add(getBonusItemShield());

    this.nextBonusItemInterval = currentTime +
    Constants.BONUS_ITEM_SHIELD_SPAWN_INTERVAL;
```

```
    } return bonusItemsToSpawn;
}
```

A utilização de **List** permite adicionar e remover itens dinamicamente, o que seria mais complexo com arrays.

Gestão de Estrelas de Fundo:

1. Código:

```
private List<BackgroundStar> stars = new ArrayList<>(); private void initializeStars() {

    for (int i = 0; i < this.starsCount; i++){

        double x = Math.random() * Constants.GAME_WIDTH;
        double y = Math.random() * Constants.GAME_HEIGHT;
        double radius = Math.sqrt(this.starsWidth * this.starsWidth + this.starsHeight * this.starsHeight) / 2;
        this.stars.add(new BackgroundStar(x, y, radius, this.starsColor, this.starsSpeed, this.starsWidth, this.starsHeight));

    }
}
```

A utilização de **List** facilita a inicialização e manipulação de grandes quantidades de objetos de fundo.

Gestão de Entidades do Jogo:

1. Código:

```
private List<GameItem> gameItems = new ArrayList<>();
```

A utilização de **List** permite adicionar, remover e iterar sobre entidades do jogo de forma dinâmica e eficiente.

Gestão de Inimigos:

1. Código:

```
public List<Enemy> getEnemiesToSpawn(long currentTime) {
 List<Enemy> enemiesToSpawn = new ArrayList<>();
 // Lógica para adicionar inimigos à lista
return enemiesToSpawn;
}
```

Isso facilita o gerenciamento dinâmico de inimigos, como o spawn de novos inimigos e a remoção dos inativos.

6 DESCRIÇÃO DE COMO AS NOVAS FUNCIONALIDADES FORAM IMPLEMENTADAS E COMO O CÓDIGO ORIENTADO A OBJETOS AJUDOU NESTE SENTIDO

Gerenciamento de Itens de Bônus:

1. A classe ***BonusItemsManager*** gerencia a criação e distribuição de itens de bônus.
2. Exemplo: Criação de um item de bônus escudo (BonusItemShield), que dá invulnerabilidade temporária ao jogador
3. Código:

```
public class BonusItemShield extends BonusItem {

    public BonusItemShield(double x, double y, double radius, Color color) {

        super(x, y, radius, color, new BonusEffect(System.currentTimeMillis(),
    Constants.BONUS_ITEM_SHIELD_DURATION,BonusType.INVULNERABLE ));

    }

    @Override

    protected void renderActiveEntity() {

    GameLib.drawShieldItem(x, y, radius * 2);
```

}}

Gestão de Fundo do Jogo:

1. A classe ***Background*** gerencia estrelas de fundo, melhorando a imersão do jogo.
2. Código:

```
public class Background implements GameItem {

    private List<BackgroundStar> stars;

    public Background(int starsCount, Color starsColor, int starsWidth, int starsHeight, double starsSpeed) {

        this.stars = new ArrayList<>();

        this.starsColor = starsColor;

        this.starsWidth = starsWidth;

        this.starsHeight = starsHeight;

        this.starsSpeed = starsSpeed;

        this.starsCount = starsCount;

        this.initializeStars();

    } // Métodos de inicialização, atualização e renderização

}
```

Gestão de Entidades:

1. A classe ***GameManager*** gerencia o estado geral do jogo, incluindo a criação e atualização de todas as entidades.
2. Exemplo: Método spawnEnemies para criar novos inimigos.
3. Código:

```
private void spawnEnemies() {
        List<Enemy>enemiesToSpawn=enemiesManager.getEnemiesToSpawn(currentTime);
        gameItems.addAll(enemiesToSpawn);

}
```

Gestão de Fundo Dinâmico:

1. A classe ***Background*** gerencia as estrelas de fundo, melhorando a imersão do jog
2. Código:

```
Private void initializeBackground() {

Backgroundforeground=newBackground(Constants.NUM_FOREGROUND_STARS,Constan
ts.COLOR_FOREGROUND_STARS,Constants.STAR_WIDTH,Constants.STAR_HEIGHT,
Constants.SPEED_FOREGROUND_STARS);

Backgroundbackground=newBackground(Constants.NUM_BACKGROUND_STARS,
Constants.COLOR_BACKGROUND_STARS,Constants.STAR_WIDTH,Constants.STAR_
HEIGHT,Constants.SPEED_BACKGROUND_STARS);          gameItems.add(foreground);
gameItems.add(background);

}
```

Gestão de Colisões:

1. A nova estrutura permite uma verificação de colisões mais organizada e modular.
2. Código:

```
private void checkCollisions() {

        for (GameItem itemA : gameItems) {

                for (GameItem itemB : gameItems) {

                        if (itemA == itemB) continue;

                        if (!(itemA instanceof Entity) || !(itemB instanceof Entity)) continue;

                         // Lógica de verificação de colisões
```

```
            }
        }
}
```

6.1 BENEFÍCIOS DA ORIENTAÇÃO A OBJETOS

1. **Encapsulamento**: Cada entidade do jogo (player, inimigo, item de bônus, estrela) está encapsulada em sua própria classe, isolando estados e comportamentos.
2. **Herança**: Reuso de código comum através de hierarquias de classes (e.g., CollidableEntity estendendo Entity).
3. **Polimorfismo**: Capacidade de tratar diferentes tipos de entidades de jogo de maneira uniforme através de interfaces e classes base, permitindo expansibilidade.

7 CONCLUSÃO

A reestruturação do código do jogo, passando de uma abordagem monolítica e baseada em arrays para uma abordagem modular e orientada a objetos, trouxe diversos benefícios, incluindo maior flexibilidade, facilidade de manutenção e possibilidade de expansão futura. A utilização de coleções Java no lugar de arrays permitiu uma gestão mais eficiente e dinâmica dos objetos do jogo. As novas funcionalidades implementadas, como a gestão de itens de bônus e a criação de um fundo dinâmico, demonstram claramente as vantagens da programação orientada a objetos.