Nomes:	**NUSP:**
Bruno Tenorio Park	15635566
Lucas Giovani Santos Ross	15471693
Tiago Almeida Silva	15490509
Vinicius Chirnev Panhoca	15580531

1. Críticas ao código original

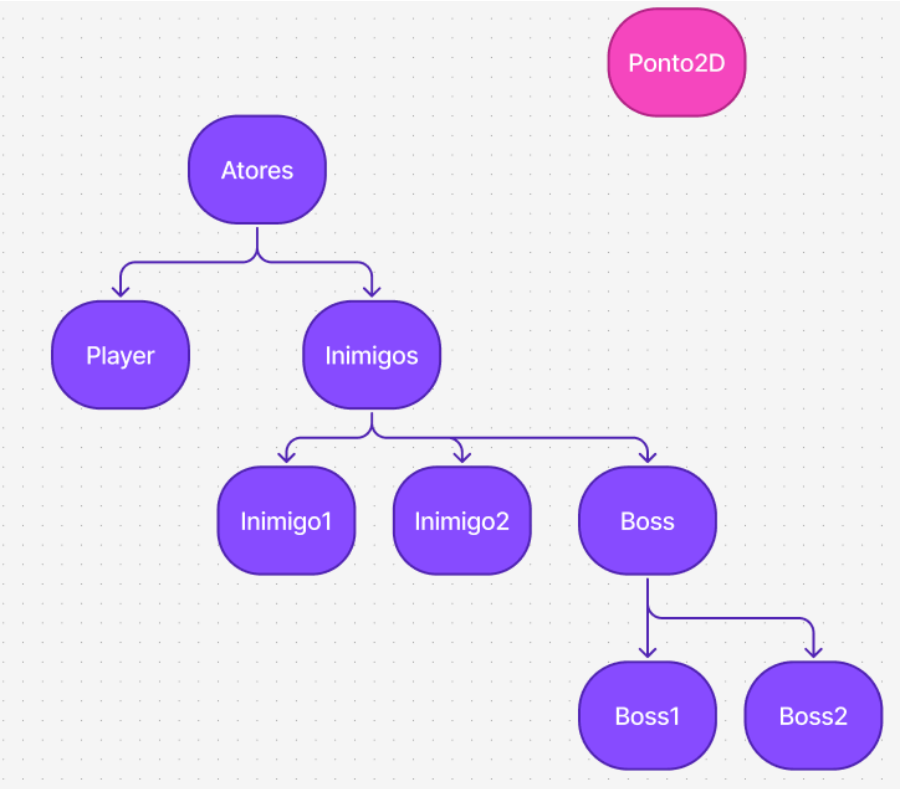
/* completar */

2. Estrutura das classes

2.1 Ponto2D

É uma classe pública que possui os atributos Coordenadas X e Y e suas respectivas velocidades, além de possuir métodos Setters e Getters, por si só não é muito útil, porém ela será incorporada em outras classes por meio da composição.

2.2 Atores e seus derivados



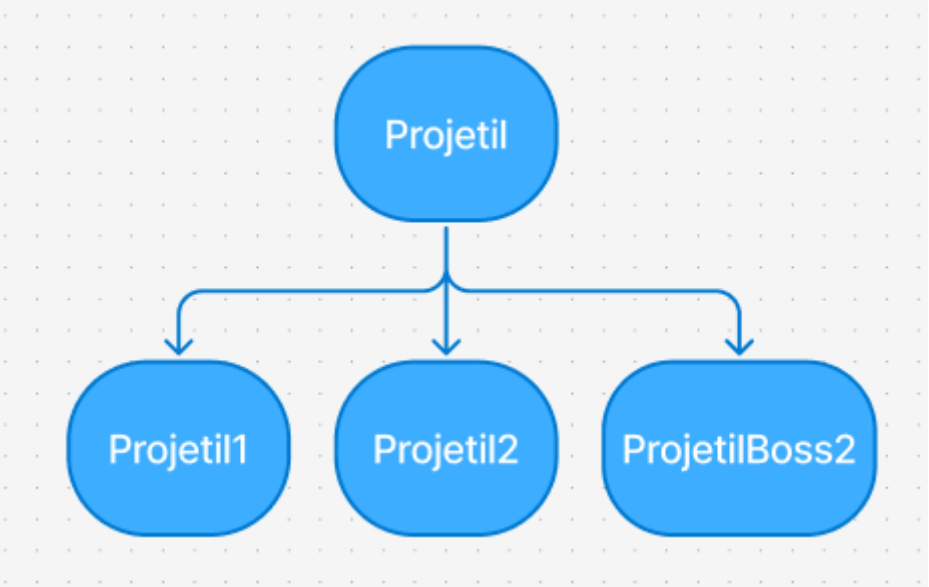
Atores: Classe pública e abstrata que define as características essenciais do player e inimigos, tendo como atributos:

- Ponto2D ponto: posição e velocidade do ator.
- LinkedList<Projetil> listaProjeteis: lista onde são armazenados os projéteis que ele dispara.
- boolean explodindo: indica se o ator está no estado de explosão.
- double inicioExplosao e double fimExplosao: tempos que controlam o início e fim da explosão.
- double raio: raio do ator.
- long proxTiro: tempo mínimo entre tiros.

Está classe também possui os métodos: colision e o abstract dispara() que são responsáveis por implementar as colisões, e cada classe diferente criar a sua própria maneira de disparar,

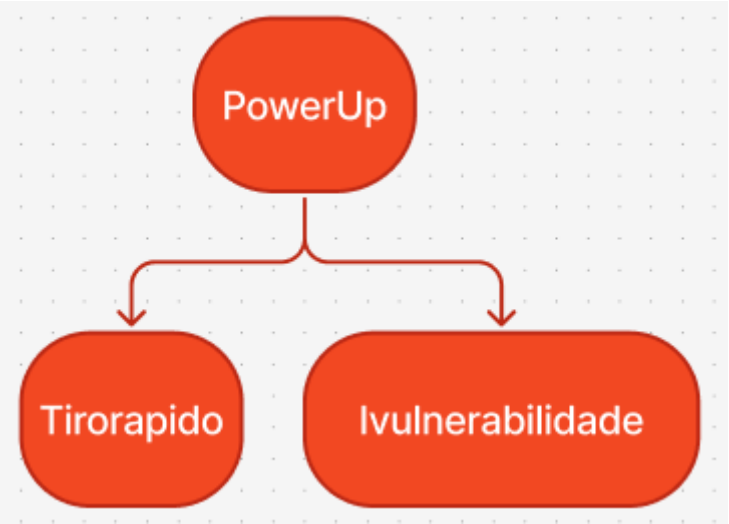
- **Player e Inimigos:** São classes públicas que estendem atores, sendo que inimigos é uma classe abstrata, adicionando os sus próprios atributos.
- **Inimigo 1 e 2:** São classes públicas que estendem Inimigos, implementando a sua própria movimentação e métodos de dispara.
- **Boss:** É uma classe que realiza a extensão de Inimigos, definindo os atributos de vida e vidainicial.
- **Boss 1 e 2** As duas classes são extensões de Boss, mas implementam os seus próprios dispara() e atualizaestado(), fazendo com que os chefes se diferenciem.

2.3 Projetil e seus derivados



- **Projetil**: É uma classe pública e abstrata que serve como base para futuros projéteis de inimigos.
- **Projetil1 Projetil2 e ProjetilBoss2:** São classes públicas utilizadas para criar os projéteis do inimigo1, inimigo2 e o projétil especial do Boss2.

2.4 PowerUp e seus derivados



- **PowerUp:** Classe pública e abstrata que irá definir as principais características dos powerups.
- **Tirorapido e Ivulnerabilidade:** Classes públicas que são extensões de PowerUp e implementam a funcionalidade dos poderes.

3. Player

/* completar */

4. Boss1

/* completar */

5. Boss2

A classe Boss 2 é uma extensão de Boss (que Boss em si é uma extensão de inimigos, e inimigos uma extensão de atores), portanto o Boss2 possui diversos atributos como os de atores que são: “Ponto2D ponto” para representar a sua posição no jogo, uma lista ligada de projéteis chamada “LinkedList<Projetil> listaProjeteis” para criar e remover os tiros da tela, “double inicioExplosão, boolean explodindo e double fimExplosão” que são atributos para controlar a explosão que o Boss irá gerar quando ele morrer. Além de seu “raio” demonstrando o seu tamanho e o “proxTiro” utilizado para criar um intervalo de tempo entre os disparos do chefe. Já seus atributos de inimigos são “double angulo” que representa a direção em que o chefe está olhando, e o “double vr” que é a velocidade de rotação do Boss. Seus atributos

de Boss são “int vida” e “double vidainicial” utilizadas para representar a quantidade de tiros que faltam para o chefe ser morto e a atualização da sua barra de vida. Boss2 não possui nenhum atributo exclusivo da sua classe, apenas métodos.

- public void Boss2 (double x, double y, double vx, double vy, double angulo, double vR, LinkedList<Projetil> listaProjeteis, int vida)

Para chamar o Boss2, basta instancia-lo com seu construtor que irá atribuir os valores passados como parâmetros aos seus atributos por meio do método super, o raio recebe um valor fixo de 27 e o atributo vidainicial recebe o valor da vida que foi passada como parâmetro.

- Public void@Override desenha(long currentTime)

Tem como parâmetro o currentTime que é utilizado para determinar o tempo da explosão ao matar o chefe, esse método é o responsável por gerar todas as imagens relacionadas ao chefe na tela, então ela desenha o formato do Boss2, muda a sua cor ao receber dano para informar ao player que os seus projéteis são efetivos, e chama o método desenhabarra() para aparecer a sua barra de vida na tela, e também desenha a explosão quando o chefe for morrer.

- @Override public void desenhabarra()

É desenhado um retângulo verde representando a vida do Boss2, e a largura do retângulo diminui baseada na vida atual do chefe em comparação com a vida inicial.

- @Override public boolean Colision(Linkedlist<Projetil> projeteis, long currentTime, double c)

Recebe como parâmetros uma lista ligada de projeteis que irá ser os projéteis do player na Main, o currentTime é usado para determinar o tempo das explosões, e o double c é responsável por manipular o tamanho da “hitbox” da colisão por meio de seu valor.

Este método também tem a propriedade de diminuir a vida do chefe quando entrar em contato com o projétil do player inimigo, e caso sua vida chegue em 0, inicia uma explosão retornando o estado explodindo.

- @Override public void dispara(long currentTime, double PlayerY)

É o método que possibilita o chefe atirar os seus próprios projeteis ao detectar que é hora de realizar o próximo tiro, quando o momento chegar, o chefe irá escolher entre dois ataques se o player estiver em uma coordenada Y menor que a do Boss2.

O primeiro ataque tem 90% de chance de ocorrer que dispara 3 projéteis vermelhos semelhantes ao do Inimigo1 em ângulos diferentes, já o segundo tiro que dispara dois projéteis laranjas com uma propriedade especial de “rebater” nos cantos da tela, essa propriedade é gerada pela inversão das suas velocidades X ou Y ao atingir uma borda.

- @Override public boolean atualizaEstado(long DeltaTime, long currentTime, double playerY, LinkedList<Projetil> ProjetilInimigo)

É o método principal que chama os outros métodos além de desenha, além de implementar a movimentação horizontal do Boss, que tem a sua velocidade horizontal invertida toda vez que “bate” em uma das bordas da tela, este método também chama o colision(),se o chefe não estiver com o estado explodindo e também chama o dispara(), caso o Boss seja explodido retorna false para futuramente ser removido do jogo, caso contrário retorna true.

6. Power ups

Os power ups são elementos que aparecem durante a execução do jogo e, ao serem coletados pela nave do jogador, aplicam efeitos temporários que melhoram seu desempenho, como maior frequência de disparo ou invulnerabilidade a danos.

Na implementação original do projeto, não havia um sistema de power-ups integrado. Por isso, foi criada no pacote classes uma classe abstrata chamada PowerUp.java, que encapsula atributos e comportamentos comuns entre os diferentes tipos de power-ups, como coordenadas (x, y), velocidade, raio e a verificação de colisão com o jogador. Essa classe também define a interface comum com os métodos desenha, atualizaEstado e aplicarEfeito.

A partir disso, foram criadas dois PowerUps:

- PowerUpTiroRapido.java: Ao ser coletado, reduz o intervalo de tempo entre os disparos do jogador.
- PowerUpInvulnerabilidade.java: Ao ser coletado, torna o jogador temporariamente invulnerável a danos.

Cada classe implementa seu próprio método de desenho, definindo a cor e formato visual (triângulo com cores distintas), e personaliza o comportamento no método aplicarEfeito.

Além disso, a lógica de aparecimento dos power-ups foi incorporada ao sistema de leitura dos arquivos de configuração das fases. Dessa forma, o aparecimento dos power-ups é controlado por arquivos .txt, seguindo a mesma estrutura usada para definir o surgimento de inimigos e chefes.

Por fim, a manipulação dos power ups foi integrada na Main.java com uma lista do tipo LinkedList<PowerUp> , onde cada instância é atualizada, desenhada e removida se sair da tela ou for coletada. Essa estrutura garante a organização e reaproveitamento de código.

7. Inimigos

Para o controle dos inimigos dentro do jogo criamos uma classe chamada Inimigos, a qual herda de Atores, pois assim, teria um Ponto2D, bem como métodos e atributos para controlar colisões, explosões e disparos. Na classe Inimigos, foram declarados atributos próprios dos inimigos, como sua velocidade angular, chamada de vR, e seu ângulo. Também foi implementado um construtor próprio que definia esse ângulo e velocidade angular, Inimigos foi declarado como uma classe abstrata, isso porque, não deve ser possível instanciar um Inimigo, apenas suas variações, com a classe abstrata declaramos métodos abstratos sendo eles: dispara() - que sobrescreve o disparar de Atores -, desenha() e atualizaEstado(). Isso foi adotado para que fosse possível controlar todos os inimigos em uma única lista ligada de Inimigos dentro da classe Main.

Da classe Inimigos, herdam Inimigo1 e Inimigo2, classes as quais definem de fato as características de cada inimigo e podem ser instanciadas. Dentro destas foram implementados os métodos abstratos declarados em Inimigos e um construtor para cada, onde foi definido o raio fixo para cada tipo de inimigo, sendo 9.0 para o inimigo 1 e 12.0 para inimigo 2. Ambas as classes utilizam o método colission() da classe Atores para verificar sua colisão com os projéteis do player, sem alterar a função em nada.

Na classe Inimigo1 seu método dispara apenas verifica se o tempo para seu próximo tiro é maior do que o tempo atual do jogo e se sua posição y é menor que a do player, indicando que ainda não passou do player, caso ambas as condições sejam verdadeiras adiciona um novo projétil a lista ligada de projéteis inimigos, a

qual é compartilhada por todos os inimigos, e então atualiza o tempo do seu próximo tiro. No método desenha, caso esteja explodindo desenha sua explosão baseada no tempo de início e fim dela, caso contrário apenas desenha o formato do inimigo 1, sendo este uma esfera ciano, a qual é definida na biblioteca GameLib. Já no método atualizaEstado são atualizadas as posições x e y baseado em sua velocidade e ângulo, caso esta instância do inimigo não esteja explodindo chama o método de colisão e caso sua explosão tenha terminado, ou tenha saído da tela, retorna falso, indicando que o inimigo deve ser excluído da lista ligada de Inimigos.

Na classe Inimigo2, utilizamos um atributo booleano a mais chamado ShootNow, que indica o momento no qual o inimigo deve iniciar seus disparos, na função dispara verificamos se esse atributo é verdadeiro e caso seja insere um novo projétil na lista ligada de projéteis inimigos, baseando-se em um array de ângulos para calcular a direção que esse projétil deve seguir. No método desenha, a única diferença é o formato desenhado que ao invés de uma esfera

ciano é um losango magenta. Já na classe atualizaEstado alguns cálculos são feitos para definir a movimentação do inimigo, já que ele se movimenta em uma elipse, por isso utilizamos algumas variáveis adicionais como a previousY e a threshold que servem para verificar as posições verticais dos inimigos e da tela, utilizando dessas variáveis a movimentação do inimigo é feita e o atributo ShootNow é atualizado, depois a função dispara é chamada e são feitas as mesmas verificações de estado explodindo e posição que são feitas em inimigo 1.

8 .Fundo/Background

O fundo é basicamente composto por estrelas, sendo que na implementação anterior/original haviam dois tipo de estrelas: do primeiro plano e de segundo plano, que possuem as mesmas variáveis com a diferença de que a velocidade das estrelas do primeiro plano são mais rápidas do que a do segundo plano. Além disso, no código original, essas duas estrelas estavam declaradas usando arrays.

Então para compactar o código e organizá-lo, implementamos no pacote classes, a classe abstrata Estrela.java que define os atributos comuns às estrelas, como coordenadas (x, y) e velocidade, além dos métodos mover e desenhar.

A partir disso, extendemos a classe abstrata Estrela.java em outras duas classes que seriam EstrelasPlano1.java e EstrelasPlano2.java,onde cada uma define sua própria velocidade e personaliza o método de desenho conforme necessário.

Com isso, na Main.java substituímos os arrays por ArrayList<Estrelas> da coleção para armazenar e manipular estrelas, além disso as listas são inicializadas com a mesma quantidade de estrelas da versão anterior, sendo que agora o código é mais conciso e menos redundante , reduzindo assim a sobrecarga que a função Main havia na implementação anterior.

9. Arquivos e Main

O jogo possuí um sistema de fases, no qual cada fase possuí uma estrutura previamente definida com quais inimigos e bosses serão instanciados em quais posições e momentos específicos, essas fases são definidas em um arquivo de configuração Config.txt encontrado dentro de uma pasta com o nome de arquivos, nesta pasta se encontra dentro do diretório principal do

nosso jogo, o diretório src, dentro da pasta arquivos se encontram os arquivos de configuração específicos para cada fase e o arquivo Config.txt. A leitura desses arquivos bem como a preparação das fases é feita dentro da classe Main e do método main, para tal, utilizamos de uma classe auxiliar chamada Instancia, a qual como o próprio nome diz armazena atributos específicos para cada instância do jogo, esses atributos são os passados nos arquivos de configuração, são eles: nome da instância (CHEFE ou INIMIGO), tipo (1 ou 2), vida, tempo, x e y. Utilizando dessa classe auxiliar, dentro da main criamos um array de listas ligadas de instância, onde cada posição do array representa uma fase do nosso jogo, e cada lista ligada possuí dentro dela todas as instâncias que precisam ser inseridas no jogo.

No método *main*, portanto, definimos uma variável booleana que define o estado do nosso jogo, se for *true*, o jogo está em execução e se for *false* o jogo deve ser finalizado, temos também duas variáveis que armazenam o tempo inicial do jogo e a variação de tempo entre os *loops* principais do jogo. Definido essas variáveis, definimos o caminho para o arquivo de configuração do jogo, que caso o jogo esteja sendo executado pelo diretório src, será “arquivos/Config.txt” e caso não será “src/arquivos/Config.txt”, definido esse caminho, abrimos nosso arquivo de configuração utilizando a classe File da biblioteca java.io e utilizamos um *scanner* para iterar por este arquivo. Definindo a vida do player, a quantidade de fases e criando o *array* de fases baseado nesta quantidade. Então iteramos por cada um dos arquivos de fase e adicionamos na nossa lista ligada para aquela respectiva fase uma instância com cada um dos atributos passados no arquivo.

Ainda na *main*, criamos uma lista ligada para os projéteis do *player*, instanciamos um *player*, criamos a lista ligada de inimigos, de projéteis inimigos, um iterador para iterar pela lista ligada de fases, um *array* para controlar as estrelas do nosso cenário, uma lista ligada de projéteis do *boss*, uma instância de boss que será atualizada em cada fase, uma lista ligada de *power ups* e por fim diversas flags de controle que serão utilizadas para excluir projéteis e inimigos das respectivas listas ligadas.

No método *main*, portanto, definimos uma variável booleana que define o estado do nosso jogo, se for *true*, o jogo está em execução e se for *false* o jogo deve ser finalizado, temos também duas variáveis que armazenam o tempo inicial do jogo e a variação de tempo entre os *loops* principais do jogo. Definido essas variáveis, definimos o caminho para o arquivo de configuração do jogo, que caso o jogo esteja sendo executado pelo diretório src, será “arquivos/Config.txt” e caso não será “src/arquivos/Config.txt”, definido esse caminho, abrimos nosso arquivo de configuração utilizando a classe File da biblioteca java.io e utilizamos um *scanner* para iterar por este arquivo. Definindo a vida do *player*, a quantidade de fases e criando o *array* de fases baseado nesta quantidade. Então iteramos por cada um dos arquivos de fase e

adicionamos na nossa lista ligada para aquela respectiva fase uma instância com cada um dos atributos passados no arquivo.

Ainda na *main*, criamos uma lista ligada para os projéteis do *player*, instanciamos um *player*, criamos a lista ligada de inimigos, de projéteis inimigos, um iterador para iterar pela lista ligada de fases, um *array* para controlar as estrelas do nosso cenário, uma lista ligada de projéteis do *boss*, uma instância de boss que será realizada em cada fase, uma lista ligada de *power ups* e por fim diversas flags de controle que serão utilizadas para excluir projéteis e inimigos das respectivas listas ligadas. No nosso *loop* principal do jogo, ou seja, um *while* que funciona enquanto a variável *running* (que define o estado do jogo) for *true,* nós atualizamos as variáveis do tempo atual do jogo e chamamos as funções atualizaEstado(...) e desenha(...) do *player* além disso, conferimos se a instância atual no nosso iterador das fases é um inimigo ou chefe e instanciamos o respectivo inimigo ou chefe, caso seja um inimigo instanciamos ele na nossa lista ligada de Inimigos e caso seja um boss apenas instanciamos ele e atribuímos seu ponteiro a variável criada anteriormente na main, claro, sempre conferindo qual tipo de inimigo ou chefe estamos instanciando e atualizando o iterador para a próxima instância da lista.

Depois de instanciados chamamos as funções de atualizaEstado e desenha para cada elemento do nosso jogo, ou seja, inimigos, chefes, projéteis de cada um deles, cenário e *powerUps.* Fazemos isso iterando pela lista ligada de cada um dos elementos e sempre conferindo o retorno do método atualizaEstado, para que caso seja falso a instância atual seja deletada da lista. Na verificação do *boss* também implementamos para que caso ele seja destruído, o iterador das fases seja alterado para a próxima fase, ou finalize o jogo caso não existam mais fases, por fim, fazemos as verificações de fim de jogo, ou seja, caso o player não tenha mais vidas restantes ou caso a tecla ESC tenha sido pressionada, o jogo é finalizado.