UNIVERSIDADE DE SÃO PAULO

SISTEMAS DE INFORMAÇÃO

GABRIEL BERNARDINI SCHMIDT – 12873188

JULIANA DE FARIA DUARTE RIBEIRO – 12684969

LUCAS SARTOR CHAUVIN – 11796718

TAMYRIS AYUMY NASCIMENTO ONODA – 12731401

PROJETO DE COO

RELATÓRIO

SÃO PAULO

2022

1. INTRODUÇÃO

Este relatório tem como objetivo descrever quais eram os problemas no código disponibilizado no e-disciplinas e documentar quais foram as melhorias realizadas.

2. CRÍTICAS AO CÓDIGO ORIGINAL DO JOGO

O código original possui uma grande quantidade de *arrays* estáticos, o que significa que é necessário definir um tamanho específico para o vetor, sendo assim menos adaptável e consumindo memória desnecessariamente. Pelo o fato de o *array* ser de tamanho fixo, quando é atingido sua capacidade máxima, é necessário chamar o método *findFreeIndex()*, que retorna o índice que está inativo, e isso causa uma maior complexidade do tempo, já que precisa percorrer o *array* inteiro novamente, e ao final, o índice inativo não é deletado, só reutilizado, o que consome mais memória.

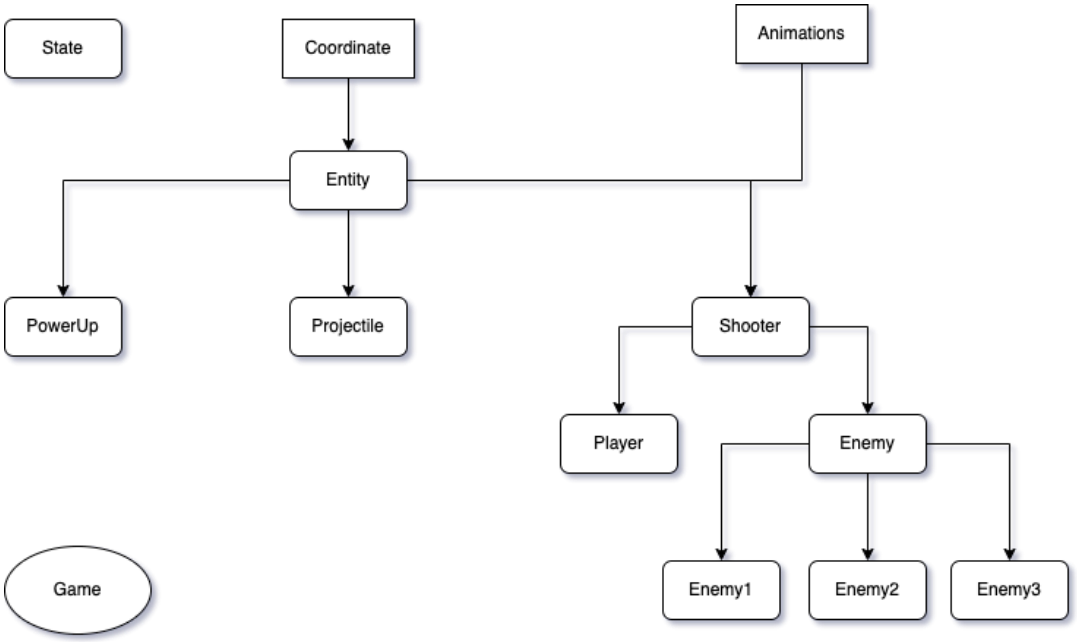
Não usa nada que o Java tem a oferecer, como coleções Java, que poderia ser usada para a criação de listas dinâmicas e não estáticas. Também não utiliza conceitos de orientação a objetos, como por exemplo desrespeita totalmente o princípio do encapsulamento.

Muito código é repetido desnecessariamente, muitas informações que poderiam ser reaproveitadas de maneira mais eficiente. Além disso, possui muitos laços de repetição, fazendo com que a complexidade fique bem maior do que seria necessário.

Nota-se que o código inteiro se concentra no método Main, o que torna a compreensão do programa muito menos legível e, no contexto de orientação a objetos, seria muito melhor se existisse classes e métodos para cada situação, seguindo o Princípio da responsabilidade única, de que cada classe/método tem sua função bem definida.

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

Em busca de solucionar os problemas citados no ultimo tópico, utilizando de uma melhor organização e reaproveitamento de código ao usufruir dos conceitos do paradigma de programação orientada a objetos. Dessa forma, chegamos a uma estrutura de classes e interfaces representada pelo seguinte diagrama:



- Animations: interface que define os métodos das entidades que são desenhadas na tela e que explodem;
- *Coordinate*: interface que define os métodos de qualquer coisa que tem uma posição e anda. Portanto, possui uma coordenada X e Y, assim como a velocidade em ambas coordenadas;
- *State*: classe *Enum* de definição dos estados de entidades (ativo, inativo e explodindo);
- *Entity*: classe que representa uma entidade básica. Implementa a interface *Coordinate* além de possuir atributos de raio e estado;
- *Projectile*: classe de um projetil. É subclasse de *Entity*, uma vez que se move e possui raio e um estado;
- *Power Up:* classe de um *power up*. Tambem é subclasse de *Entity*;

- *Shooter*: subclasse de *Entity*, define os métodos e atributos de uma entidade que atira (player e inimigos);
- *Enemy*: subclasse de *Shooter*, define os métodos e atributos básicos de um inimigo, mas, como cada tipo de inimigo possui atributos únicos, tivemos que separar em outras classes filhas de *Enemy* para cada tipo de inimigo (Enemy1, Enemy2 e Enemy3);
- *Background*: classe que define um objeto que será desenhado no fundo da tela;
- *Game*: classe central do jogo, possui apenas métodos estáticos e concentra toda a lógica do funcionamento jogo.

4. DESCRIÇÃO DE COMO AS COLEÇÕES JAVA FORAM UTILIZADAS PARA SUBSTITUIR O USO DE *ARRAYS*

Conforme citado anteriormente, os *arrays* de tamanho fixo possuem inúmeras desvantagens quando comparados as coleções Java, que são bem mais eficientes e práticas. Por isso, foi utilizado o *ArrayList* da *Collections* do Java para substituir os *arrays* estáticos.

Durante o funcionamento do jogo, todas as entidades se tornam, em algum momento, inativas, isto é: não impactam mais me nada no funcionamento do jogo. No código original do jogo, utilizava-se um método que retornava os índices inativos de cada *array* (*findFreeIndex()*), os quais eram substituídos por valores novos.

Ao substituir para *ArrayList*, isso não é mais necessário, uma vez que, por ser uma estrutura de dados dinâmica, ela cresce indefinidamente. Entretanto, se não tratado da forma correta, isso pode ser um problema já que, ao crescer de forma indefinida, a estrutura pode acabar usando muita memoria e possivelmente quebrando o funcionamento do jogo.

Por conta disso, no novo código, após a rotina de checagem de estados de cada *ArrayList* de entidades, todas as entidades inativas são removidas da lista, mantendo a performance do jogo.

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

Após a fatoração do código original do jogo para as boas praticas do paradigma de programação orientada a objetos, adicionar novas funcionalidades a ele tornou-se um processo muito mais simples.

Isso pois, com toda a lógica do jogo orientada a objetos, ficou possível reaproveitar diversos trechos necessários para o funcionamento das novas funcionalidades.

No caso da implementação do novo inimigo (Enemy3), toda a lógica de entidade e de inimigos já estava pronta. Portanto, apenas foi necessário estender a classe *Enemy* e adicionar os atributos únicos do novo inimigo.

O mesmo aconteceu na implementação do *Power Up*. Nesse caso, como o *Power Up* não anda nem atira, estendeu-se apenas a classe *Entity* e foi necessário adicionar apenas uma nova logica de mudar alguns atributos do Player assim que ele colide com o *Power Up*.