Projeto Final – MC322

Breno Rezende Pinto **RA:** 167688
Marina Lemos dos Santos **RA:** 183955
Claudio dos Santos Junior **RA:** 195727

1. Introdução

O seguinte projeto tem como objetivo a elaboração de uma adaptação do jogo Pac-man. Para tal o programa foi escrito em Java, utilizando os princípios de programação orientado a objetos.

O jogo é desempenhado de forma textual a fim de facilitar a implementação. A movimentação do jogador é feito na entrada do programa, dada pelas teclas W, A, S e D. Sendo W para cima, A para a esquerda, D para a direita e S para baixo (as entradas são lidas com as letras minúsculas). Na saída do jogo, temos o mapa com caracteres representando seus elementos da seguinte forma: H são as paredes, - as pastilhas, @ as pastilhas especiais, G o jogador (pacman), M os fantasmas no modo ativo (enquanto o jogador não come as pastilhas especiais) e W os fantasmas no modo inativo (quando o jogador pode comer os fantasmas).

Escolhemos estruturar o programa em três pacotes: *main*, *game* e *labyrinth*.

2. Estrutura do programa

2.1. Pacote *main*

2.1.1. Classe *Main*

A classe *Main*, possui uma variável chamada *game*, do tipo *GameEngine*. Ela é usada para chamar o método *runGame*, responsável por iniciar o jogo.

2.2. Pacote *game*

Neste pacote temos toda a parte responsável pelo fluxo de execução do programa e suas classes são inicializadas pela *Main*, que dão sequência ao funcionamento do jogo.

2.2.1. Classe *GameEngine*

A classe *GameEngine* é o núcleo do programa. Ela é responsável por inicializar o jogo. Ela possui 3 atributos: *renderManager* (do tipo *TextRenderManager*), *scanner* e *labyrinthMap* (do tipo *LabyrinthMap)*. Essa classe recebe do teclado os comandos do jogador e inicia o loop de execução do jogo pelo método *gameLoop.* Ela também é responsável por escolher o mapa (dependendo da escolha do jogador pode ser um mapa pré definido, ou criado aleatoriamente). Todo o jogo funciona dentro desse loop e o jogo só acaba por ordem deste método - que no caso é quando o jogador perde todas as vidas ou quando ele ganha ao pegar todas as pastilhas do mapa.

2.2.2. Interface *LabyrinthObjectVisitor*

A interface contém os métodos a serem implementados por outra classe. Os métodos por sua vez vão fazer a ligação entre o mapa e os objetos do mapa.

2.2.3. Classe *TextRenderManager*

A classe em questão é responsável por renderizar o mapa do labirinto, isto é, criar a matriz que simboliza o labirinto, limpar as posições e também imprimir na tela o mapa.

2.3. Pacote *labyrinth*

Foi escolhido esse nome para o pacote pois nele está todas as classes relacionadas ao mapa do jogo - que se assemelha ao labirinto por seus corredores estreitos e paredes. Todos os elementos presentes neste labirinto que são parte do jogo são implementados aqui, e serão explicados em seguida.

2.3.1. Classe *LabyrinthMap*

O labirinto possui todos os objetos contidos no mapa, como os fantasmas, o jogador e listas com todos as paredes e as pastilhas. Além disso, o labirinto possui um tamanho (largura e altura). Também optamos em deixar nessa classe a pontuação do jogador e a quantidade de vidas do mesmo, pois acreditamos que isso são características do mapa e não de algum objeto específico do mapa. Por fim, existe o atributo que guarda o tempo de ativação da pastilha especial, ou seja, quantos ciclos precisam ocorrer para que a pastilha especial perca o seu efeito sobre os fantasmas.

Nessa classe, implementamos um método chamado *incrementScore* que calcula a pontuação do jogador recebendo como parâmetro um *boolean*. Caso seja verdadeiro, é uma pastilha especial, caso contrário, é uma pastilha normal. Esse método também aumenta a vida do jogador a cada cinco mil pontos acumulados.

O método *ghostSearch* verifica se o jogador cruzou com algum fantasma, se ele cruzar há a chamada para o método *ghostCross* que recebe o fantasma e testa se o mesmo está ativo ou não. Caso ele esteja ativo, o jogador perde uma vida e todos retornam às posições iniciais. Do contrário, o jogador ganha 300 pontos e o fantasma retorna ao início.

Optamos por mover os fantasmas a cada movimento do jogador, dessa forma o labirinto possui um método *moverFantasmas* que altera a posição dos fantasmas quando o jogador se move. Além disso, caso o jogador conquiste uma pastilha especial, todos os fantasmas mudam de status pois é possível nesse momento eles serem mortos. Então o método *ghostsActivity* altera os status de todos os fantasmas.

O método *updateMap* movimenta o objeto do jogador, verifica se nesse movimento ele cruza com algum fantasma e então move os fantasmas. Além disso também observa se o jogador cruzou com alguma pastilha normal, se sim, aumenta sua pontuação e altera a pastilha como conquistada. Isso também ocorre com as pastilhas especiais, caso o jogador cruze com alguma, os fantasmas são desativados por 50 rodadas.

Por fim, o método *isDone* retorna se o jogo acabou, caso o jogador perca todas as vidas e caso o jogador ainda tiver vidas e todos as pastilhas já tiverem sido conquistadas. Se nenhuma dessas situações ocorrerem, o jogo continua.

2.3.2. Classe *LabyrinthMapLoader*

Para essa classe, buscamos utilizar o padrão singleton, no qual só pode ser criada uma instância do *LabyrinthMapLoader* e somente essa classe pode ser capaz de instanciá-la. Assim, método *getInstance* retorna a única instância. Enquanto o método *loadMapFromFile* realiza a leitura do arquivo de texto e salva o mapa textual nessa instância.

2.3.3. Classe *LabyrinthMapRandom*

Nessa classe também utilizamos o padrão singleton. Ela é responsável por criar o mapa aleatoriamente. No entanto, por se tratar de um mapa de um jogo, e com vários objetos, o mapa foi criado com algumas características fixas e as demais características aleatórias. As características fixas do mapa foram: o tamanho do mapa, as bordas cercadas por paredes e a posição dos fantasmas e do jogador. Foi utilizado porcentagem para atribuir a quantidade de paredes, pastilhas e pastilhas especiais. As pastilhas especiais representam 0,3% do mapa (para que tenha menos de 10 pastilhas especiais no total), 44,7% para as pastilhas comuns, 50% do mapa será composto por paredes e 5% será locais sem nenhum objeto. Essas porcentagens foram atribuídas para que a quantidade de cada tipo de objeto seja próxima das do mapa pré definido.

2.3.4. Classe LabyrinthObject

Essa classe é herdada por todos os objetos que existem no mapa, dessa forma alguns métodos comuns a todos são implementados nela. Como por exemplo um *getter* para a coordenada do objeto e um método para verificar se a coordenada é a mesma.

No seu construtor temos a inicialização do atributo coordenada que recebe como parâmetro dois inteiros x e y. Os métodos accept são um tipo de “resposta” aos métodos visit da interface. É como se o método accept concedesse permissão de visita a quem pede.

2.3.5. Classe *Player*

Essa classe representa o jogador, o método *getDestiny* vai receber a direção digitada na entrada do teclado e retornar a nova coordenada do jogador, esse método é chamado pelo *move* que verifica se não tem parede para poder atualizar a posição atual. Caso tiver uma parede naquela direção, o jogador “perde o movimento”.

2.3.6. Classe *Ghost*

Classe abstrata que define os fantasmas do jogo, possui dois atributos: a direção atual do objeto e seu status, para sabermos se ele está ativo no jogo ou no modo em que pode ser morto pelo jogador.

Ela é estendida da classe *LabyrinthObject*, então no seu construtor que recebe duas coordenadas como parâmetros, há uma chamada para o construtor da classe herdada, inicialização da direção atual como UP e seu status como ativo.

Foram implementados métodos *setters* e *getters* para ambos os atributos, uma vez que é preciso alterá-los e acessá-los em outras partes do código.

O método *containsWall* verifica se em uma determinada coordenada tem uma parede ou não. Enquanto o método *getDestiny* recebe uma direção e retorna uma nova coordenada para o fantasma. E, por fim, temos um método *abstract* que implementa a movimentação do fantasma. Esse método é sobrescrito pelas classes filhas, uma vez que cada fantasma possui uma característica diferente.

2.3.7. Classe *RandomGhost*

O fantasma aleatório escolhe uma direção aleatoriamente para se deslocar. Então o método *randomDirection* é o responsável por implementar a escolha aleatória, enquanto o método *move* movimenta o fantasma de acordo com essa direção, caso não tenha nenhuma parede.

2.3.8. Classe *JumperGhost*

Esse é o fantasma prestigiador que se teletransporta de forma aleatória em uma nova posição no mapa. O método *getRandomCoordinate* escolhe aleatoriamente uma nova coordenada pro jogador que não contenha uma parede e a retorna, ela utiliza do método *lastCoordinate* que obtém o tamanho o mapa. Então o método *move* faz o fantasma se mover alterando sua coordenada para a aleatória obtida no método anterior.

2.3.9. Classe *EvasiveGhost*

O fantasma evasivo tenta se afastar dos outros fantasmas, de forma a escolher uma direção que seja em média mais longe dos outros. É calculada a média simples das coordenadas dos fantasmas (excluindo o evasivo) e então é verificada qual distância é menor entre o fantasma evasivo e o ponto médio calculado. Por exemplo se a distância no eixo x for menor, é porque o fantasma deve se afastar daquele ponto no eixo x. Então é visto se essa distância é negativa ou positiva (para questão de referência se o fantasma deve se deslocar para a direita ou esquerda).

2.3.10. Classe *ChaserGhost*

Nessa classe temos o fantasma perseguidor estendido da classe *Ghost*. Esse fantasma tenta se aproximar o máximo possível à posição atual do jogador. Então o método *calculateDistance* recebe uma coordenada como parâmetro e calcula a distância até ela. Enquanto o método *chasingDirection* retorna a direção que o fantasma vai avançar que seja a menor até o jogador. Para o fantasma se mover, existe o método *move* que recebe uma lista das paredes e o jogador e verifica se o fantasma pode se mover naquela direção, então atualiza suas coordenadas atuais.

Nesse caso encontramos uma dificuldade, já que o método *move* herdado da classe abstrata dos fantasmas possui apenas um parâmetro, que é a lista de paredes, e analisando esse fantasma, precisaríamos passar mais parâmetro para esse método. Então, a nossa solução foi deixar o método *move* herdado vazio e implementar outro dando *overload* para conseguirmos passar os parâmetros necessários.

2.3.11. Enum *Direction*

A seguinte classe possui apenas uma lista enumerada das possíveis direções permitidas no jogo, como *UP, DOWN, LEFT* e *RIGHT*.

2.3.12. Classe *Coordinate*

As coordenadas são muito importantes no jogo, visto que cada objeto possui uma coordenada no mapa. Dessa forma, cada coordenada possui dois inteiros que identificam onde o objeto do labirinto está localizado. Seu construtor recebe esses parâmetros e os inicializa na coordenada.

Também existem métodos para alterar e acessar a coordenada. O método *equals* compara dois objetos do tipo coordenada e retorna se elas são iguais.

2.3.13. Classe *Checkpoint* e *SuperCheckpoint*

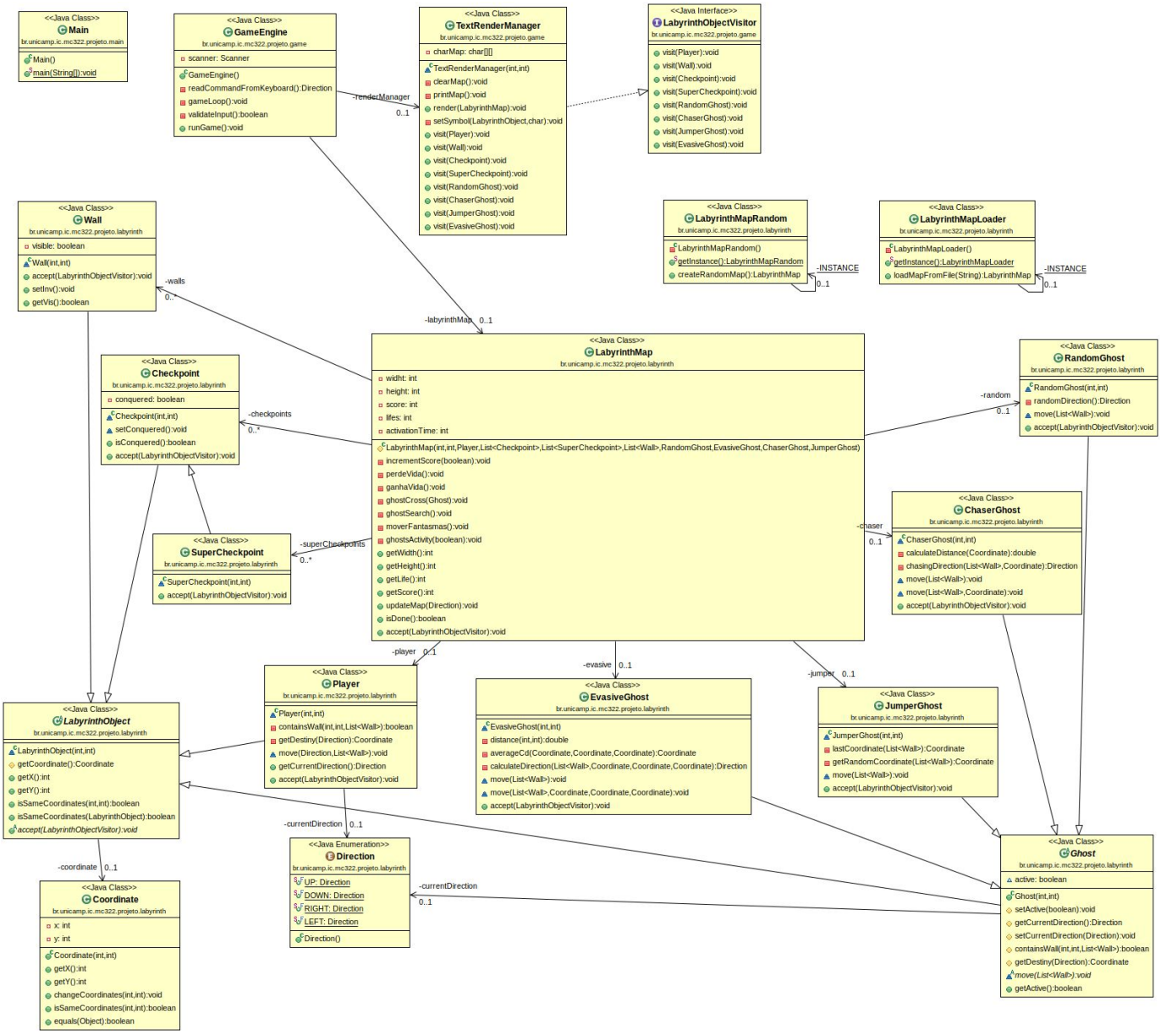
Ambas as classes definem as pastilhas do jogo. A classe *SuperCheckpoint*, que representa as super pastilhas, é subclasse de Checkpoint. Isso acontece pois na renderização do mapa, na classe *TextRenderManager*, ao visitar um objeto é atribuído um símbolo a ele. E como as super pastilhas precisam de um símbolo diferente, ela precisa ser um outro objeto.

A classe Checkpoint possui um atributo protected, para ser visível para as suas filhas, do tipo *boolean* que diz se os mesmos foram conquistados pelo jogador ou ainda não. No construtor esse atributo é iniciado como falso. É implementado também métodos *getters* e *setters*.

2.3.14. Classe *Wall*

A classe parede, além de conter os métodos e atributos da sua classe mãe *LabyrinthObject*, possui um atributo do tipo *boolean* para sua visibilidade e método *getter* e *setter* para sua visibilidade. Essa visibilidade foi adicionada somente para as paredes "invisíveis", que são buracos inacessíveis no mapa. Se esses buracos fossem lidos como passagem, e não paredes, o fantasma que se teletransporta poderia aparecer por lá, o que seria indevido. A questão de torná-las invisíveis e não usar o mesmo caractere igual o resto foi puramente estética.

3. Diagrama UML



4. Dificuldades enfrentadas

O projeto foi feito com base no laboratório 11 da disciplina e, por conta disso, as maiores dificuldades foram na implementação do laboratório e em seguida na adaptação para o projeto, mas nada que com tempo e esforço não tenha sido resolvido.

Nas classes dos fantasmas, o que causava grandes problemas era a implementação dos movimentos dos fantasmas perseguidor e evasivo. No começo pensamos em implementar de forma que o fantasma simularia um trajeto até a coordenada, respeitando as paredes e qual caminho seria o mais curto até o jogador - no caso do perseguidor - ou o mais distante dos outros fantasmas - no caso do evasivo. Porém, como não estávamos conseguindo chegar numa solução boa para essa ideia, decidimos optar por deixar o algoritmo desses fantasmas

menos complexos, calculando puramente a distância entre cada eixo e decidindo o movimento de acordo com essa distância e, caso a melhor distância levasse a uma parede, o fantasma simplesmente não se moveria.

Outra dificuldade encontrada pelo grupo foi na criação do mapa aleatório. Não sabíamos como seria possível criar um mapa aleatório, já que teríamos que construir um mapa transitável, que não quebrasse o programa e com todos os objetos. No final, foi optado por não construir ele totalmente aleatório, fixamos os objetos essenciais para impedir que o programa desse alguma exceção e ficasse deformado. Além disso, como iríamos inserir os objetos através de intervalos estatísticos, não poderíamos saber o número exato de cada objeto, por conta disso, os fantasmas e o jogador foram fixados, já que, só podem existir um de cada objetos.

Dessa forma, decidimos que as posições restantes seriam escolhidas arbitrariamente entre os 4 objetos possíveis: pastilhas, pastilhas especiais, paredes e espaços vazios. Observamos a porcentagem de ocorrência desses objetos no mapa pré definido, e a utilizamos no sorteio dos objetos. Ademais, o algoritmo estatístico foi desenvolvido com base no sorteio aleatório de um número entre 0 e 999, confiamos que o algoritmo random do Java seja aleatório, e dessa forma, cada número tem 0,1% de chance de ser escolhido. E assim, estabelecemos os intervalos para a seleção de cada objeto, de acordo com a porcentagem pré definida de cada um.