Manual de Sistema – Pacman

Diego Fleury Corrêa de Moraes

Nusp: 11800584

Aviso/dica:

Fora testada a possibilidade de rodar o programa através do terminal, a qual (para o sistema operacional Ubuntu 20.04 LTS, o qual fora testado e desenvolvido o jogo) segue o seguinte conjunto de instruções:

Após compilar o projeto Netbeans, um aviso fora dado pela IDE, que permite rodar o arquivo .jar pelo terminal:

java -jar "/home/diego-fleury/NetBeansProjects/PacMan/dist/PacMan.jar"

Para a minha hierarquia de arquivos só fora possível para o projeto localizar os arquivos necessários para a contrução do tabuleiro se a pasta **ArquivosJogo** *estivesse localizada dentro do ambiente de execução do terminal. Por exemplo, caso tenha lançado o terminal dentro de "/home/diego-fleury" a pasta de configurações deve estar contida dentro de "/home/diego-fleury".*

Fora isto aparenta não ter nenhuma problemática a execução, embora não tenha sido testado em qualquer outra distribuição Linux ou nos S.O Windows e Mac, porém, devido à portabilidade de arquivos .jar, espera-se que não tenha-se nenhum imprevisto.

Para melhorar a qualidade da documentação do programa como um todo buscou-se evitar a descrição das regras nesta secção, seja na pontuação, dinâmica dos fantasmas com relação aos estados ou modo de transversar o menu, dado que estes aspectos já estão detalhados na secção de manual de usuário. Com isso, segue a descrição das classes do jogo, em uma ordem que fora vista como a melhor para a compreensão geral do código:

Classes:

PacMan:

Esta classe apenas inicializa a aplicação JavaFX, com a instanciação da classe **Jogo** dentro do método (necessário para a GUI funcionar) start. Além disso, caso deseja-se adicionar novos mapas ao jogo, deve-se colocar o caminho dos arquivos de configuração por mapa (mais detalhes em **Blueprint**) no método start.

Jogo:

Esta classe inicializa o jogo assim que instanciada. Inicialmente realiza-se a extração dos metadados dos mapas passados em **PacMan**, sendo que esta informação é armazenada em várias instâncias da classe **Blueprint** (uma para cada mapa). Após este processo o método MenuInicial é chamado, que essencialmente inicializa uma instância da classe **Menu** (salvando-a) e inicializa-se o procedimento do jogo em SetGame.

Com a chamada deste método várias variáveis cruciais para o funcionamento do jogo são inicializadas, sendo que, no início do método, a instância de **Menu** é chamada algumas vezes para

exibir as janelas gráficas que retornam o nome do jogador e a sua escolha de mapa. A partir deste instante são inicializadas as entidades (**Player** e **Fanstasma**s). Logo após é configurada a instância de **Cena** que faz a exibição do tabuleiro do jogo em si.

Após isso, é chamada a função initGame, que inicializa o conjunto timer-timertask que efetua a sincronização da ativação dos updates no status do jogo. Dentro do método run de timertask temos que é chamada a função GameLoop, que essencialmente faz todo o fluxo da lógica que ocorre em a cada instânte de jogo. Por conta de funcionar em tempo discreto, temos que a velocidade das entidades é simulada através de simplesmente não executar a movimentação de tal entidade até que uma certa quantidade de ciclos tenha passado. Exemplo: para o jogador, temos que esta quantidade é fixa, a cada 4 ciclos deste timer existe apenas a execução da ação de deslocamento do jogador em um deles.

A cada execução do clock é executada a movimentação de **Player**, de cada um dos **Fantasma**s, e depois é chamado o método Juiz, que faz a verificação, atualização, e mediação dos estados do jogo. Inicialmente pensou-se em utilizar o paradigma de máquina de estados (no conceito de *Design Patterns* para POO) para o funcionamento deste método, porém devido em parte à complexidade, em parte pelo tempo não seguiu-se com este plano, e decidiu-se apenas por uma séries de manutenção de variáveis privadas da classe **Jogo** pareadas com uma série de condicionais.

Enfim, caso durante a execução o jogador opte por pausar o jogo (através da tecla ENTER ou do botão de menu na janela gráfica), uma variável é acionada que efetivamente bloqueia a excecução da lógica do jogo durante o tempo que estiver pausado, e a função MenuPausa é chamada, em que a classe **Menu** é encarregada de receber a opção de comando, e o fluxo do programa é redirecionado para quaisquer opções (continuar: pausado = false, sair: termina o programa, mudar de mapa: efetivamente chama SetGame novamente e reinicia os estados do jogo).

Por último, caso detecte-se que o jogador passou de nível, a função ResetGame é chamada, sendo que ela essencialmente faz tudo aquilo que SetGame faz, porém simplesmente preservando os valores do jogo, ao invés de reiniciá-los.

PauseException:

Esta classe essencialmente herda de Exception e é responsável apenas por sinalizar para o programa que deseja-se pausar.

Menu:

O funcionamento pode ser resumido em 3 funções:

→ Armazenar o último mapa Selecionado (index e metadados).
→ Criar janelas gráficas genéricas, com mensagens de prompt e local para armazenamento de input do usuário.
→ Métodos públicos que geram/validam todas as situações em que as janelas/input são necessários.

Blueprint:

Esta classe é responsável pelo preprocessamento, armazenamento e recuperação dos metadados dos mapas disponíveis ao jogo, armazenando as condições iniciais das 3 variáveis essenciais para definir movimentação pelo mapa do jogo (momentum, caminho e offset, mais informações em **Controlador**), além, é claro, da informação de como contruir o mapa em si.

Cada mapa é definido à partir de dois arquivos, em que um determina a informação de conectividade (nós em um grafo, através da classe **NoGrafo**) e o outro é responsável pela informação do conteúdo de cada elemento deste labirinto (arestas em um grafo, através da classe **Caminho**), como os atributos, 'spawn' das **Entidade**s, entre outros. O formato em si que cada um fora armazenado possúi algumas limitações, tanto em poder representativo quanto em facilidade de definição, discutidas na secção limitações mais abaixo.

Arquivo:

Esta classe faz a interface para com as funcionalidades de leitura dos arquivos de configuração, junto da escrita do arquivo de *logging*. Para fins de facilitação da execução do programa no ambiente de terminal (através da execução do .jar) fora adicionada uma impressão do diretório atual na classe, que deve auxiliar na execução se seguida com os passo descritos no começo deste manual.

Cena:

A forma encontrada de realizar a interface para com a aplicação JavaFX fora através da comunicação de **Jogo** com esta classe, através de métodos internos. As variáveis internas à instância de **Cena** armazenam as imagens de cada um dos objetos a serem desenhados (fantasmas, pacman e frutas) e os métodos são em maioria privados (para a construção do mapa através dos metadados em **Blueprint**, com a chamada à Monta, a requisição de atualização daquilo mostrado na tela, através do método Update, o fechamento e retorno do palco de exibição, através de Desmonta, e atualizações assíncronas do estado dos fantasmas, através de SetEstado, sendo os únicos métodos públicos).

O desenho do mapa não se utilizou da ferramenta FXML (pela característica de desejar-se implementar muitos mapas), mas pela geração simples de uma hierarquia:

Root → (possui 3 filhos):

→ Duas instâncias de **HBox,** uma inferior e outra superior, que são responsáveis pela exibição das labels.
→ A instância central do labirinto, que é essencialmente uma instância de **GridPane** que possui como nós 'filhos' várias instâncias de **StackPane**.

StackPane → (possui um número variável de filhos, sendo que depende da configuração inicial contruída na chama à Monta, porém é atualizada dinamicamente pela movimentação das entidades em sua 'superfície').

Cada entidade é representada por uma instância de **ImageView**, que efetivamente é um container para uma instância de **Image**, que contém a imagem desejada, mudada ao longo da partida, seja pela rotação do pacman ou pela mudança dos estados dos fantasmas.

Controlador:

A classe é responsável por toda a lógica de movimentação das **Entidade**s através de **Mapa**, além de fazer a comunicação com os elementos assíncronos de detecção de ações (por *event handling* presente nos aspectos de GUI de JavaFX).

A comunicação com **Cena** é feita através da herança de **EventHandler<KeyEvent>** de JavaFX, junto da linkagem do palco principal (**Stage**) com este handler (implementado dentro da classe, realizando o armazenamento da última tecla pressionada pelo jogador internamente) em específico.

A movimentação fora pensada através do controle das estruturas de dados presentes nas classes **Caminho** e **NoGrafo**, sendo que a principal ideia seria a de restrição de controle e fluidez do movimento: se uma movimentação inválida é detectada a lógica automaticamente segue com a última movimentação válida, através do armazenamento de momentum das **Entidade**s, que indica esta tendência de movimento. A direção de movimento só pode mudar na intersecção de dos caminhos, sendo isto representado através de **NoGrafo**, que facilita a linkagem com os outros 'túneis' (**Caminho**) presentes na redondeza deste. Por fim, para representar a localização dentro de um **Caminho**, existe a noção de um offset, que essencialmente monitora (através da convenção de sinais de eixos definidos como x e y) a localização no interior destes.

Uma das principais razões pela escolha desta representação fora, além da possibilidade de criar novos mapas, o algorítmo de movimentação interna dos **Fantasma**s, que necessita acessar toda a vizinhança de um nó em um grafo (recursivamente) diversas vezes. Fazer isso através de uma matriz seria excessivamente custoso, além de não trazer outros benefícios úteis. Todas as classes que podem se movimentar no tabuleiro devem herdar de **Entidade**, pois somente assim o polimorfismo implementado para a movimentação funcionará.

Mapa:

Responsável pelo armazenamento do conjunto de **Caminho**s e **NoGrafo**s que compoem o layout do labirinto. Realiza a interface com **Jogo** para o controle de alguns aspectos globais (como aparição da fruta), mas primariamente com **Controlador** para a movimentação dos personagens.

Monta o labirinto através da leitura, acesso e manipulação dos dados de uma certa **Blueprint** que define a topologia total.

NoGrafo:

Realiza a interface com a estrutura/classe dual (**Caminho**), sob o controle da comunicação **Mapa:Controlador**, dentro do contexto armazenado em **Entidade**.

Caminho:

Realiza a interface com a estrutura/classe dual (**NoGrafo**), sob o controle da comunicação **Mapa:Controlador**, dentro do contexto armazenado em **Entidade**.

Entidade:

Define uma série de métodos (abstratos ou não) e variáveis que permitem a integração com os aspectos de movimentação (e parcialmente da lógica do restante do jogo, como o nome dos personagens).

Player:

Implementação de métodos de **Entidade** sob a ótica daquilo que é necessário sob o contexto do jogador, com a implementação de métodos e variáveis extras que armazenam informações relevantes para a lógica, como a pontuação, vidas, atributos capturados, nome, entre outros.

Fantasma:

Implementação sob uma base comum (abstrata) de métodos e variáveis comuns a todos os fantasmas, como os 3 estados (FUGA, ATAQUE e MORTO), além de tempo de fuga, interface com as velocidades e nome.

FantasmaAleatorio:

A lógica, vantagens e desvantagens deste tipo estão descritos em maior detalhe no manual de usuário, portanto a implementação basicamente se resume na lógica por trás do descrito.

FantasmaPerseguidor:

A lógica, vantagens e desvantagens deste tipo estão descritos em maior detalhe no manual de usuário, portanto a implementação basicamente se resume na lógica por trás do descrito.

A_Star:

Esta classe realiza a implementação do algorítmo de busca em grafos A*, conceitualmente adaptado para manipular as estruturas de labirinto descritas anteriormente. A métrica/heurística utilizada fora a distância Manhattan, pelo fato de o labirinto apenas permitir movimentos verticais ou horizontais. Os métodos públicos permitem chamar/setar os alvos deste algorítmo, sendo que cada instância de **Fantasma** possui uma instância desta classe, por conta da necessidade de retornar ao spawn quando o for comido pelo **Player** (**FantasmaPerseguidor** utiliza-se desta classe para a fuga e ataque também).

Limitações:

→ O sistema fora feito pensando em Java, e depende profundamente da execução neste ambiente, devido à existência do coletor de lixo, dado que muitos aspectos teriam de ser mudados caso existisse a necessidade de manejamento de memória.

→ O formato em que o mapa fora escrito depende de algumas regras, como a necessidade de existir um nó em (x = 0 , y = 0), o fato de todas as coordenadas terem de ser positivas, o fato de que depende-se da sensibilidade do programador para gerar o arquivo (deve-se pensar em não se ‘contradizer’ em vários pontos do arquivo, pela existência de redundâncias na escrita, como por exemplo garantir que se A→ B existe também a conexão B→ A), entre outros empecilhos.

→ A geração das janelas de jogo dependem da existência de um mapa ‘relativamente grande’, devido ao comprimento fixo para as Labels. As dimensões deve ser maiores que (x = 15, y = 20) para a renderização satisfatória da imagem.

→ A forma com que os fantasmas retornam à origem após mortos, e o modo com que retornam ao estado de ataque, é executada de maneira não ideal, porém isto depende (e fora notado infelizmente tarde demais) da forma com que A* é implementado/integrado, e portanto a solução que maximiza a jogabilidade fora a implementada (se o fantasma está se direcionando para o nó de ‘spawn’ ele retorna ao estado de ataque). Entretando este aspecto negativo é minimizado para mapas maiores (e possivelmente com alguns ajustes no arquivo de configuração, como feito para o mapa 1).