MC322 C – Programação Orientada a Objetos
Laboratório 11 – 1s2019

Leonardo Montecchi (Professor)
leonardo@ic.unicamp.br

Elder Rodrigues (PED)
elderjr1995@gmail.com

21/05/2019

1 Introdução

O principal objetivo deste laboratório é de auxiliar no desenvolvimento do projeto. As estruturas que serão apresentadas aqui poderão servir como base para o projeto. Portanto, para o problema que será detalhado na próxima seção, será apresentado uma possível solução destacando as decisões feitas sobre a arquitetura do sistema.

Outro detalhe é que este laboratório é mais extenso comparado aos demais. Para o próximo laboratório prático (28/05), você pode dar continuidade no desenvolvimento deste laboratório.

2 Especificação do Sistema

Neste laboratório deverá ser desenvolvido um jogo que consiste em um labirinto.

Todo labirinto contém um tamanho especificado por comprimento e altura. O jogador começa em um local específico do labirinto e deve passar por todos os *checkpoints* que estão espalhados no labirinto. Ao passar por todos, o jogo termina automaticamente.

Todos os elementos do labirinto possuem um par de coordenadas x e y que representa sua localização. A seguir cada um dos elementos do labirinto é detalhado:

- **Parede**. As paredes são apenas objetos no mapa para limitar o movimento do jogador.
- **Checkpoint**. Os *checkpoints* são pontos específicos espalhados pelo labirinto. Quando um *checkpoint* é alcançado pelo jogador, ele não é removido do labirinto, mas apenas é sinalizado que o jogador já passou por ele.

- **Jogador**. É o personagem que está no labirinto. O jogador pode movimentar para direita, esquerda, cima ou baixo. Vale lembrar que o movimento do jogador só deve ser executado se o destino final estiver livre ou contém um *checkpoint*. Por exemplo, se o jogador estiver na posição (3,2) e desejar movimentar para a direita, então deve ser verificado se a posição (4,2) contém uma parede antes de realizar o movimento.

Observação. Não é necessário considerar dimensões de tamanho diferente para os objetos, isto é, todos os elementos do labirinto são do mesmo tamanho. Portanto, para tratar efeitos de colisões, basta verificar se os objetos estão nas mesmas coordenadas.

3 Desenvolvimento

3.1 Modelagem dos elementos básicos

A Figura 1 ilustra o diagrama UML das classes básicas do jogo. Cada um dos objetos do labirinto estão modelados como classe. Além disso, a super classe é uma classe abstrata que encapsula às coordenadas dos objetos do labirinto.

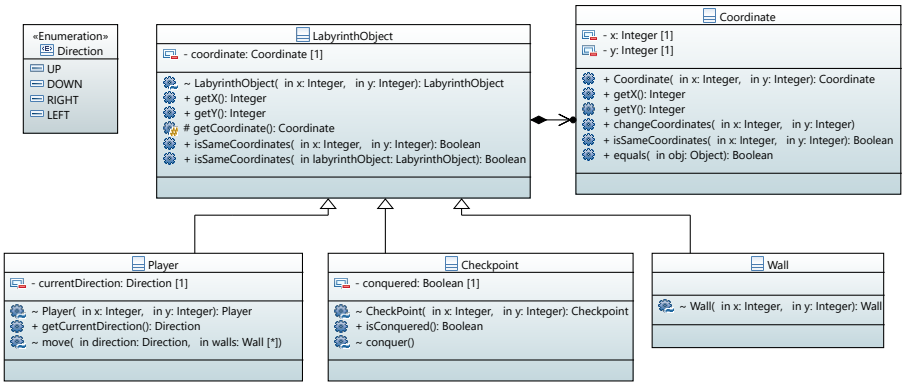

Figura 1: UML dos elementos básicos do labirinto

Classe *Coordinate*. Esta classe apenas representa um par de coordenadas x e y.

Classe *LabyrinthObject*. Esta classe representa um objeto genérico do labirinto. Note que o construtor da classe recebe as coordenadas em forma de inteiros e não como um objeto da classe *Coordinate*, isto é, a criação de um objeto *Coordinate* é feita internamente. Essa foi uma decisão para deixar de controle interno qual classe irá representar as coordenadas. Dessa forma, caso a classe *Coordinate* seja excluída e trocada por outra classe (e.g., *Point* de java.awt), classes externas não serão afetadas por tal mudança.

Outro detalhe importante é que o método *getCoordinate* tem visibilidade *protected*. Existem classes externas que precisam da informação das coordena-

das. No entanto, deixar o método *getCoordinate* público implica no retorno da referência das coordenadas, isto é, classes externas podem alterar as coordenadas do objeto invocando o método público *changeCoordinates*, resultando em uma possível invalidação do modelo (e.g., mover uma parede). Devido a isso, o método *getCoordinate* tem visibilidade protected. Classes externas podem consultar as coordenadas dos objetos invocando os métodos *getX* ou *getY*.

Classe *Wall*. Representa apenas uma parade do labirinto.

Classe *Checkpoint*. Representa um *checkpoint* do labirinto. É possível verificar se o *checkpoint* já foi conquistado ou não utilizando o método *isConquered*. O método *conquer* por sua vez modifica o atributo *conquered* para true, indicando que o *checkpoint* foi conquistado. Porém, note que este último método tem visibilidade de pacote. Apenas classes de gerenciamento interno do pacote podem realizar a invocação.

Classe *Player*. Representa o jogador no mapa. O jogador tem uma direção atual (para onde está olhando) e que pode ser conferida pelo método *getCurrentDirection*. Um jogador também pode movimentar pelo labirinto. Esta ação é implementada pelo método *move* que recebe a direção que o jogador deve movimentar e também uma lista de paredes. A lista de paredes é entregue ao método para que seja verificado se o jogador está se movimentando para uma parede. Quando um jogador consegue realizar o movimento com sucesso, sua direção atual é alterada para a direção do movimento que realizou. Note também que o método *move* tem visibilidade de pacote, isto é, apenas classes do gerenciamento interno do pacote podem invocar tal método.

3.2 Modelagem do labirinto

A Figura 2 ilustra o diagrama UML para a classe *LabyrinthMap*.

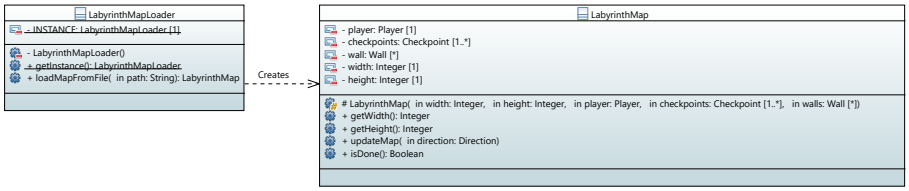

Figura 2: UML do mapa do labirinto

LabyrinthMap. Esta classe representa um mapa do labirinto. Aqui é encapsulado o comprimento (*width*) , altura (*height)* e os elementos que compõe um mapa (paredes, *checkpoints* e um jogador).

O mapa contém apenas quatro métodos públicos: *getWidth*, *getHeight*, *isDone* e *updateMap*. O método *isDone* apenas verifica se o labirinto foi completado. Para isso, basta percorrer a lista de *checkpoints* e verificar se todos *checkpoints* já foram conquistados. Já o método *updateMap* atualiza o mapa de acordo com uma direção recebida. De forma mais detalhada, o método que atualiza o mapa recebe uma direção, move o jogador caso a direção seja diferente

de *null* e verifica se a nova posição do jogador conquistou um dos *checkpoint*.

Além disso, o mapa é modelado como uma classe imútavel, isto é, após criar um mapa, nenhum objeto será removido ou adicionado. Devido a isso, o mapa não permite acesso às suas listas através de métodos *gets*, pois caso o acesso seja liberado, elementos que não pertencem ao mapa podem ser adicionados, podendo invalidar o mapa. Aqui pode surgir a dúvida: então como será possível imprimir ou desenhar os elementos que estão no mapa? Isso será explicado na próxima seção.

MapLoader. Esta classe é responsável por criar instâncias da classe *LabyrinthMap*. Note que todos os construtores até então foram definidos com visibilidade de pacote. Essa decisão foi feita justamente para não permitir que classes externas criem mapas e seus elementos. A única forma externa para se obter um mapa deve ser através da classe *MapLoader*. Além disso, tal classe foi modelada seguindo o padrão de projeto *Singleton*, isto é, apenas uma instância da classe deve existir. O método *loadMapFromFile* obtém um mapa através de um arquivo texto. Para simplificar a implementação, você pode criar também um método *createDefaultMap* para criar um mapa padrão apenas para testar seu código.

Exercício - Parte 1. A partir deste ponto do laboratório, você deve implementar as classes mencionadas de acordo com os diagramas UML apresentados. Lembre-se que todas as classes devem ser implementadas no mesmo pacote para obter o resultado esperado da visibilidade de pacote. Contudo, as próximas classes que serão apresentadas devem ser implementadas em pacotes diferentes.

3.3 Modelagem da GameEngine

O jogo proposto neste laboratório pode ser visualizado de duas formas diferentes: textual ou gráfica. Seguindo os conceitos de POO, você deve estruturar seu código de tal forma que novas visualizações possam ser adicionadas sem impactar nas estruturas já definidas no programa. Visto isso, uma possível solução é apresentada na Figura 3.

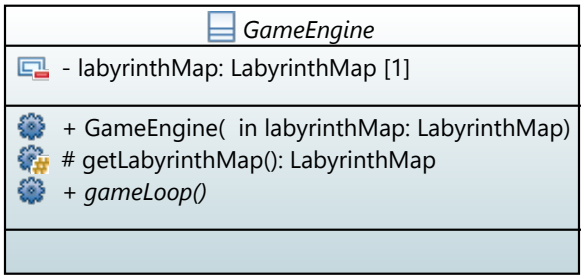

Figura 3: UML da classe *GameEngine*

Classe *GameEngine*. Esta classe tem a proposta de generalizar como o

labirinto será visualizado e gerenciado. Toda classe que estende a *GameEngine* deve prover como o *gameLoop* será executado. Tal método tem a função de estabelecer como o jogo será atualizado e visualizado.

3.3.1 TextGameEngine

Um exemplo de implementação concreta da classe *GameEngine* é ilustrado pelo diagrama UML da Figura 4.

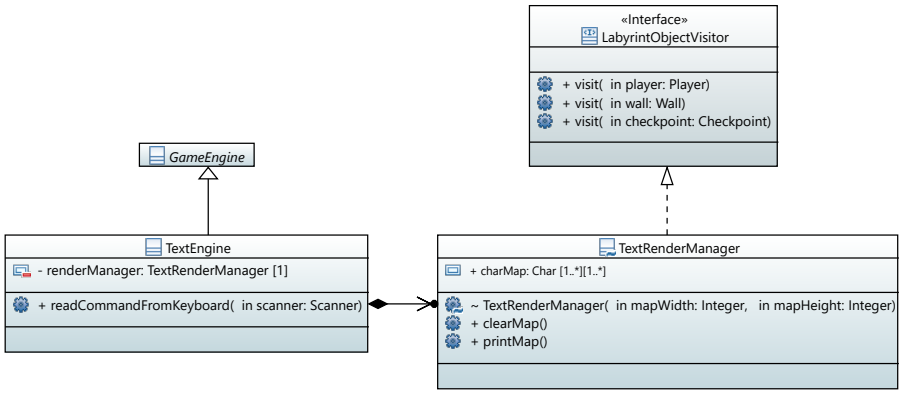

Figura 4: UML da classe *GameEngine*

Classe *TextEngine*. Esta classe estende a classe *GameEngine* com o objetivo de prover uma implementação para gerenciar e visualizar o jogo através do terminal/prompt de comando. O *game loop* desta classe basicamente segue os passos: i) imprime o labirinto na tela; ii) lê o comando do próximo movimento no teclado; iii) atualiza o mapa; e iv) verifica se o labirinto foi completado.

Classe *TextRenderManager*. A responsabilidade de imprimir o labirinto na tela é da classe *TextRenderManager*. Note que a própria classe tem visibilidade *package* (ao declarar a classe ela não é *public class*, mas somente *class*). Isso porque a classe *TextRenderManager* tem o único propósito de auxiliar a classe *TextEngine*, isto é, não há nenhuma necessidade de classes externas do pacote da *TextEngine* saber de sua existência.

Apesar da responsabilidade da classe *TextRenderManager* ser de imprimir o labirinto, como isso deve ser feito sendo que o mapa não disponibiliza as listas dos elementos? Para resolver este problema de forma elegante, é possível utilizar o padrão de projeto *Visitor*. O padrão *Visitor* é principalmente utilizado para separar o algoritmo da estrutura. No contexto deste laboratório, o padrão será utilizado para separar o algoritmo de renderização das classes *Jogador, Parede e Checkpoint* para não criar uma dependência da estrutura com a forma que ela é visualizada. No padrão *Visitor*, existem dois tipos de entidade: o visitante e quem está sendo visitado. O visitante geralmente é estruturado em forma de *interface* (*interfaces* ainda não foram discutidas na disciplina, faça uma breve

pesquisa para saber os conceitos básicos) e deve poder visitar qualquer um dos tipos da hierarquia do objeto que está sendo visitado. A interface para o contexto deste laboratório então seria:

```
1 public interface LabyrinthObjectVisitor {
2
3     public void visit(Jogador jogador);
4     public void visit(Parede parede);
5     public void visit(CheckPoint checkpoint);
6 }
```

A entidade que pode ser visitada por sua vez tem o método abstrato *accept* que recebe um visitante e aceita (ou não) ser visitada. No contexto deste laboratório, quem pode ser visitado é a classe *LabyrinthObject* e suas classes filhas. A implementação do método *accept* pode ser implementada da seguinte forma:

```
1  public class LabyrinthObject {
2    ...
3    public abstract void
         accept(LabyrinthObjectVisitor visitor);
4  }
5
6  public class Player extends LabyrinthObject {
7    ...
8    @Override
9    public void accept(LabyrinthObjectVisitor
         visitor) {
10     visitor.visit(this);
11   }
12 }
13
14 public class Wall extends LabyrinthObject {
15   ...
16   @Override
17   public void accept(LabyrinthObjectVisitor
         visitor) {
18     visitor.visit(this);
19   }
20 }
21
22 public class Checkpoint extends LabyrinthObject {
23   ...
24   @Override
25   public void accept(LabyrinthObjectVisitor
         visitor) {
26     visitor.visit(this);
27   }
28 }
```

Note que o método *accept* é sobrescrito em cada subclasse para invocar o método *visit* de sua respectiva classe. É importante mencionar que mesmo que o

código é exatamente igual em todas as subclasses, cada um tem comportamento diferente. Por exemplo, o método *accept* da classe *Jogador* invoca o método *visit* que recebe um objeto do tipo *Player* como parâmetro, enquanto que na classe *Wall* é invocado o *visit* que recebe um objeto do tipo *Wall*.

Visto como o padrão *Visitor* é aplicado no contexto deste laboratório, a classe *TextRenderManager* implementa a *interface LabyrinthObjectVisitor* e decide como cada objeto do labirinto será renderizado. O código abaixo ilustra como ficou o código da classe.

```
1  class TextRenderManager implements
       LabyrinthObjectVisitor{
2
3   private char charMap[][];
4
5   public TextRenderManager(int mapWidth, int
       mapHeight) {
6    this.charMap = new char[mapHeight][mapWidth];
7   }
8
9   public void clearMap() {
10   for(int i = 0; i < charMap.length; i++) {
11    for(int j = 0; j < charMap[0].length; j++) {
12     charMap[i][j] = ' ';
13    }
14   }
15  }
16
17  public void printMap() {
18   for(int i = 0; i < charMap.length; i++) {
19    for(int j = 0; j < charMap[0].length; j++) {
20     System.out.print(charMap[i][j] + " ");
21    }
22    System.out.println();
23   }
24  }
25
26  private void setSymbol(LabyrinthObject obj, char
       character) {
27   charMap[obj.getY()][obj.getX()] = character;
28  }
29
30  @Override
31  public void visit(Player jogador) {
32   setSymbol(jogador, 'J');
33  }
34
35  @Override
36  public void visit(Wall parede) {
37   setSymbol(parede, 'X');
38  }
39
```

```
@Override
 public void visit(CheckPoint checkpoint) {
  if(checkpoint.isConquered()) {
   setSymbol(checkpoint, 'T');
  }else {
   setSymbol(checkpoint, 'C');
  }
 }
}
```

Finalmente, a classe *LabyrinthMap* também contém o método *accept* que recebe um visitante. Para cada elemento contido no mapa, é invocado então o método *accept* definido na classe *LabyrinthObject*. O código abaixo ilustra como tal gerenciamento é feito. Note que desta forma classes interessadas na lista de elementos do mapa podem realizar suas operações, mas sem acesso a lista.

```
public void accept(LabyrinthObjectVisitor visitor){
 for(Wall wall : walls) {
  wall.accept(visitor);
 }
 for(CheckPoint checkpoint : checkpoints) {
  checkpoint.accept(visitor);
 }
 player.accept(visitor);
}
```

Por último, o *gameLoop* da classe *TextManager* é implementado da seguinte forma:

```
@Override
public void gameLoop() {
 Scanner scanner = new Scanner(System.in);
 LabyrinthMap map = getMap();
 Direction newDirection;
 while (!map.isDone()) {
  renderManager.clearMap(); //limpa o mapa
  map.accept(renderManager); //estrutura o mapa
  renderManager.printMap(); //imprime o mapa na tela
  newDirection = readCommandFromKeyboard(scanner);
  map.updateMap(newDirection);
 }
 System.out.println("Labirinto finalizado!
    Parabens");
 scanner.close();
}
```

Exercício - Parte 2. Faça a implementação da *GameEngine* e *TextEngine* para visualizar o jogo via prompt/terminal.

3.3.2 GraphicalGameEngine (extra)

A classe *GraphicalGameEngine* gerencia o jogo de forma a apresentar uma visualização gráfica. Porém, como as bibliotecas gráficas de Java estão fora do escopo desta disciplina, tal classe não será apresentada neste laboratório. Caso tenha interesse nesta parte, compareça nos atendimentos que os detalhes particulares desta implementação serão explicados.

3.4 Conclusão

Ao definir as estruturas seguindo os conceitos de POO, note que agora é possível estender o código para implementar novas visualizações do labirinto. Além disso, também é possível alternar de visualização de forma fácil. Por exemplo, o código abaixo ilustra a inicialização do jogo. Perceba que basta alterar a *GameEngine* que está sendo utilizada para alterar o modo que o jogo é visualizado/gerenciado.

```
1  public class Main {
2
3   private static void runGame(GameEngine gameEngine){
4    gameEngine.gameLoop();
5   }
6
7   public static void main(String[] args) {
8    Mapa map =
         MapLoader.getInstance().createDefaultMap();
9    runGame(new TextEngine(map)); //executa o jogo de
         forma textual
10   runGame(new GraphicalEngine(map)); //executa o
         jogo de forma grafica
11  }
12 }
```

A arquitetura planejada neste laboratório é uma possível solução para o problema, mas não é a única. Quais decisões você tomaria que sejam diferente das apresentadas? Por exemplo, a hierarquia definida dos elementos básico não está totalmente coesa para adicionar novos elementos no labirinto. O que aconteceria caso fosse desejado adicionar um outro objeto que também limita o movimento do jogador, mas não necessariamente é uma parede? Talvez o interessante então seria ter um atributo "passavel"definido na classe *LabyrinthObject* que indica se o jogador pode passar ou não por tal objeto.