Projecto Guiado – 2ª Iteração
Teste unitário e desenvolvimento guiado por testes.

2.1 Desenvolver uma classe de testes unitários automáticos em JUnit 4, para testar o jogo no modo básico, com um dragão imóvel. A classe deve conter vários métodos de teste para testar as seguintes situações elementares (um método para cada situação):

a) herói move-se uma posição em direção a uma célula livre;
b) herói tenta mover-se sem sucesso contra uma parede;
c) herói move-se para a posição da espada e apanha a espada;
d) herói desarmado move-se para uma posição adjacente ao dragão e é morto por ele (derrota);
e) herói armado move-se para uma posição adjacente ao dragão e mata-o;
f) herói move-se para a saída após apanhar a espada e matar o dragão (vitória);
g) herói tenta mover-se sem sucesso para a saída sem ter apanhado a espada;
h) herói tenta mover-se sem sucesso para a saída armado mas sem ter morto o dragão.

Em cada método, usar um labirinto o mais simples possível, num estado inicial que permita testar a funcionalidade num único passo do herói, como ilustrado abaixo.

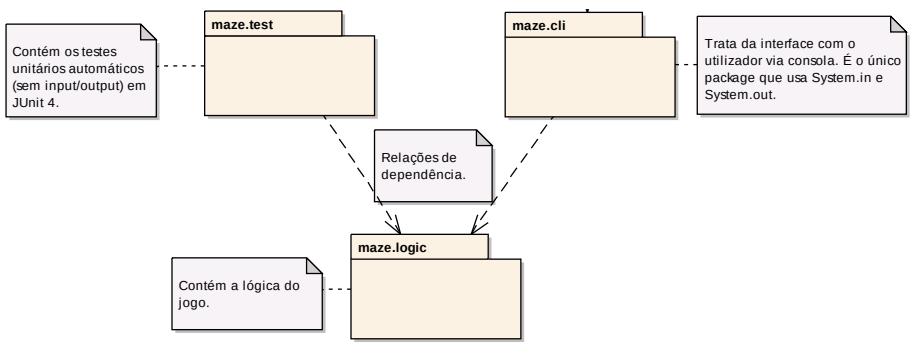
```
package maze.test;
import static org.junit.Assert.*;
import org.junit.Test;
import maze.logic.*;

public class TestMazeWithStaticDragon {
   char [][] m1 = {{'X', 'X', 'X', 'X', 'X'},
                   {'X', ' ', ' ', 'H', 'S'},
                   {'X', ' ', 'X', ' ', 'X'},
                   {'X', 'E', ' ', 'D', 'X'},
                   {'X', 'X', 'X', 'X', 'X'}};

   @Test
   public void testMoveHeroToFreeCell() {
      Maze maze = new Maze(m1);
      assertEquals(new Point(1, 3), maze.getHeroPosition());
      maze.moveHeroLeft();
      assertEquals(new Point(1, 2), maze.getHeroPosition());
   }

   @Test
   public void testHeroDies() {
      Maze maze = new Maze(m1);
      assertEquals(MazeStatus.HeroUnarmed, maze.getStatus());
      maze.moveHeroDown();
      assertEquals(MazeStatus.HeroDied, maze.getStatus());
   }
}
```

O resto do programa deve continuar a funcionar normalmente. No final do exercício, a estrutura do programa deve ser semelhante à ilustrada abaixo.



2.2 Criar uma classe de teste adicional para testar o jogo com os comportamentos mais complexos do dragão (movimentação aleatória, dormir/acordar). Neste caso, para cada movimento do herói (a que se pode seguir automaticamente um movimento do dragão), podem existir vários estados seguintes possíveis do jogo; o código de teste deve verificar se o estado seguinte é um dos admissíveis, falhando se não for um dos admissíveis.

Sugestão (7/3/2016): Organizar o código de teste como ilustrado abaixo.

```
   @Test(timeout=1000)
   public void testSomeRandomBehavior() {
      boolean outcome1 = false, outcome2 = false, ...;
      while (! outcome1 || ! outcome2  ...) {
         someAction;
         if (condition1)
            outcome1 = true;
         else if (condition2)
            outcome2 = true
         ...
         else
            fail("some error message");
   }
}
```

2.3 Com a ferramenta **EclEmma**, analisar a cobertura de instruções e acrescentar mais casos de teste se necessário para garantir que a cobertura do pacote com a lógica do jogo é >= 75%.

2.4 Com a ferramenta **PIT**, analisar a cobertura de mutantes e acrescentar mais casos de teste se necessário para garantir que a cobertura de mutantes no pacote com a lógica do jogo é >= 50%.

2.5 [Para fazer em casa e apresentar na aula seguinte] Gerar aleatoriamente o labirinto (incluindo a disposição inicial de H, D e E), com dimensão N x N, com N indicado pelo utilizador, e executar o programa com esse labirinto.
Contemplar as seguintes restrições:

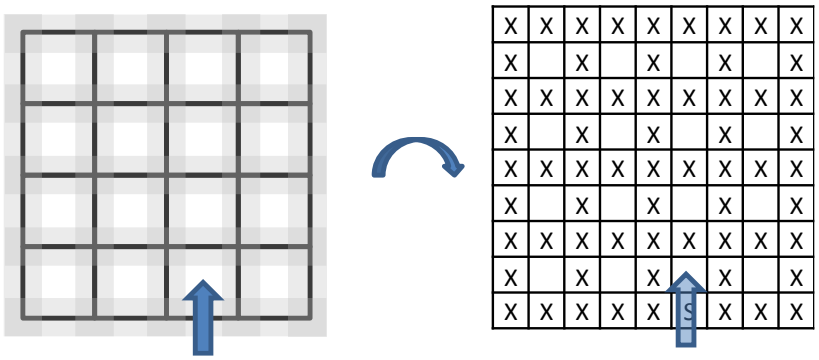
a) a fronteira deve ter exatamente uma saída, e de resto parede;
b) a saída não pode ser num canto;
c) tem de existir um caminho entre qualquer célula em branco e a saída;
d) não podem existir quadrados 2x2 só com espaços em branco;
e) não podem existir quadrados 2x2 com espaços em branco numa diagonal e paredes na outra;
f) não pode conter quadrados 3x3 só com paredes;
g) E, H, D e S devem ser colocados em posições diferentes, com H e D não adjacentes.

O código de teste destas restrições é fornecido na classe **`TestMazeBuilder`** anexa, que assume que a classe geradora de labirintos se chama **`MazeBuilder`** e implementa a interface:

```
package maze.logic;

public interface IMazeBuilder {
   public char[][] buildMaze(int size) throws IllegalArgumentException;
}
```

Como base para o algoritmo a desenvolver, sugere-se a utilização do 1º algoritmo indicado em http://en.wikipedia.org/wiki/Maze_generation_algorithm. Como esse algoritmo considera uma representação em que as paredes não têm espessura, é necessário adaptá-lo tendo em conta o seguinte mapeamento (ilustrado para a situação do labirinto do início do algoritmo):



Nesta abordagem, na representação da direita (sugerida no exercício) o tamanho tem de ser ímpar.

Para indicações mais detalhadas, podem consultar ainda a seguinte página que o estudante Henrique Ferrolho preparou: http://difusal.blogspot.pt/2014/02/maze-generation-algorithm.html

2.6 [Para fazer em casa e apresentar na aula seguinte] Criar a possibilidade de existência de mais do que um dragão. Usar uma coleção apropriada para guardar o conjunto de dragões.