Universidade Federal de Minas Gerais
INSTITUTO DE CIÊNCIAS EXATAS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Modularização de Software

Programação Modular - Atividade 1

Matheus Filipe Sieiro Vargas - 2014144812

1 Relatório sobre o projeto

A proposta de atividade apresentada visa a implementação em linguagem de programação JAVA, de uma versão modificada do conhecido problema Knight's Tour.

1.1 Knight's Tour

Em um jogo de Xadrez, o cavalo (Knight) é uma peça que possui uma movimentação diferenciada das demais peças. Sua movimentação é baseada em descrever trajetórias semelhantes a letra "l" no tabuleiro seguindo as seguintes regras:

- Se move duas casas em relação ao eixo Y de coordenadas cartesianas, no sentido positivo ou negativo, e apenas uma em relação ao eixo X, no sentido positivo ou negativo.
- Se move duas casas em relação ao eixo X de coordenadas cartesianas, no sentido positivo ou negativo e apenas uma em relação ao eixo Y, no sentido positivo ou negativo.

A partir dessa movimentação, o problema proposto consiste em movimentar a peça em todas as 64 possíveis casas do tabuleiro, sem que este, visite a mesma casa mais de uma vez.

O problema Knight's Tour é amplamente discutido devido a complexidade de suas soluções uma vez que exige muitos recursos computacionais.

1.2 Atividade 1

A atividade proposta consiste em uma versão do Knight's Tour onde uma vez que a peça atinja uma casa que não possibilite novas movimentações, deve retroceder a casa anterior e tentar novos caminhos. Portanto a quantidade de vezes que a peça visita a mesma casa não importa. O enfoque do projeto é baseado na implementação dos atributos e métodos que possibilitem a movimentação do cavalo no tabuleiro utilizando conceitos de modularização de código.

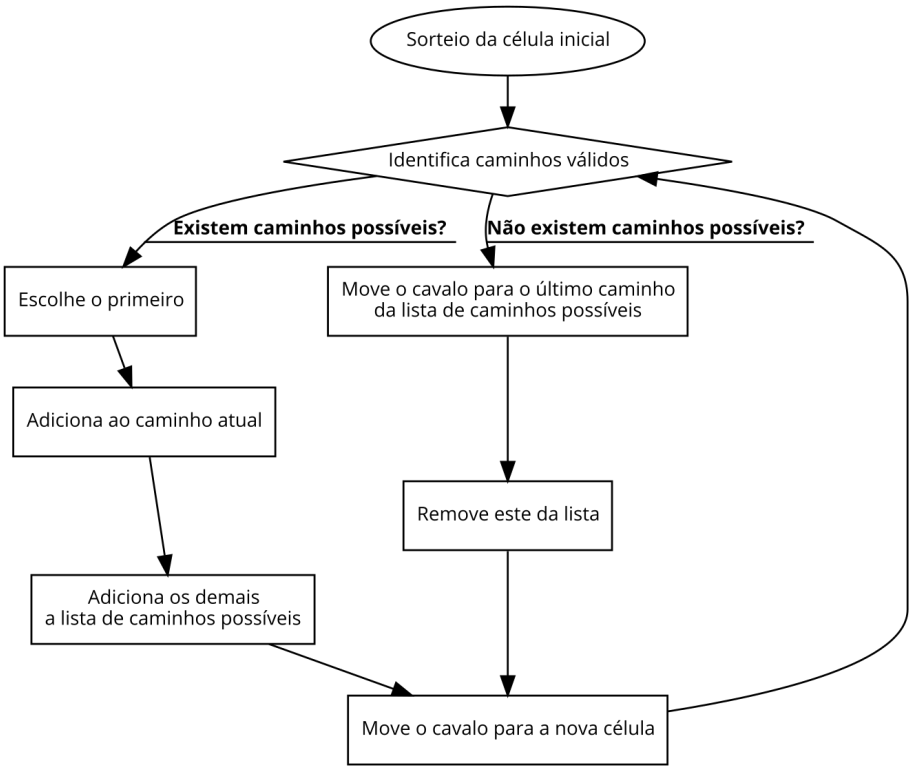

Figure 1: Fluxo de código.

2 Solução

2.1 Descrição

O método adotado para a solução segue a filosofia do **backtracking** auxiliado por uma lista de caminhos possíveis não percorridos.

Ao identificar o fim de um caminho, a lista de caminhos alternativos, ordenada de acordo com a movimentação prévia, é consultada e um novo caminho é definido à partir da posição imediatamente anterior à ultima do caminho atual.

2.2 Módulos

2.2.1 Módulo principal - Main.java

O módulo principal tem a finalidade de sortear a casa inicial do *tour* do cavalo, bem como instanciar as classes responsáveis pelo tratamento das funções relativas aos seus objetos. É o ponto de partida do código como um todo, e chama todas as demais classes para realizar as respectivas operações.

```java
import java.util.Random;
public class Main {

    public static void main(String[] args) {
        for(int i = 0; i < 6; i ++){

            Random rand = new Random();

            //Initial coordinates
            int INITIAL_X = rand.nextInt(7) + 0;
            int INITIAL_Y = rand.nextInt(7) + 0;

            System.out.println("Starting Tour from " + new
    ChessCoordinates().printCoordinates(INITIAL_X,INITIAL_Y));

            //Initializes the coordinates
            ChessCoordinates initialChessCoordinates = new
    ChessCoordinates(INITIAL_X,INITIAL_Y);

            //Iniztializes the board
            ChessBoard board = new ChessBoard();
            //Initializes the Knight
            ChessKnight chessKnight = new ChessKnight(
    initialChessCoordinates);
            long start = System.currentTimeMillis();
            chessKnight.moveKnight(board);
            long end = System.currentTimeMillis();
            System.out.println("It took " + (end - start) + " ms" +
     "\n");
        }

    }
}
```

Listing 1: Class Main.java

2.2.2 Módulo Tabuleiro - ChessBoard.java

Módulo responsável pelo tratamento dos atributos e funções específicas relativas ao tabuleiro.
Funções presentes

- **Construtor:** Instancia a matriz que representa o tabuleiro inicializada com valores arbitrários.
- **Mostrar matriz:** Mostra a configuração da matriz passada como parâmetro.

Esta classe é invocada a cada nova iteração da solução e toda as vezes que é necessário imprimir o estado da matriz que represente o tabuleiro.

```java
public class ChessBoard {

    String board[][];
    private int CHESS_DIMENS = 8;
    private String INITIAL_VALUE = "__";

    public ChessBoard() {
        board = new String[CHESS_DIMENS][CHESS_DIMENS];

        for(int x = 0; x < CHESS_DIMENS; x++){
            for(int y = 0; y < CHESS_DIMENS; y++){
                board[x][y] = INITIAL_VALUE;
            }
        }
    }

    public void printChessBoard() {
        System.out.println("Board final configuration");

        int x,y;
        System.out.println(" -------------------------");
        for(y = 0; y < CHESS_DIMENS; y++){
            System.out.print("| ");
            for(x = 0; x < CHESS_DIMENS; x++){
                System.out.print(board[x][y] + " ");
            }
            System.out.print("|");
            System.out.println();
        }
        System.out.println(" -------------------------");
    }

}
```

Listing 2: Class ChessBoard.java

2.2.3 Módulo Células - ChessCoordinates.java

Módulo responsável pelo tratamento dos atributos e funções específicas relativas as coordenadas das células do tabuleiro.
Funções presentes

- **Construtor (Vazio e Com parâmetros):** Instancia as coordenadas correspondentes a uma célula do tabuleiro.
- **Mostrar coordenadas:** A função mostrar as coordenadas explora o polimorfismo para criar um padrão ao mostrar as coordenadas passando o objeto como parâmetro ou apenas o valor de X e de Y.
- **Igual (Sobrescrito) :** A função **equals** foi sobrescrita para comparar os objetos à partir de seus atributos.

Esta classe é invocada por todos os módulos que utilizam as coordenadas do tabuleiro.

```java
public class ChessCoordinates {

    //X cordinate
    int x;
    // Y cordinate
```

```
6       int y;
7
8       public ChessCoordinates(){
9
10      }
11
12      public ChessCoordinates(int x, int y){
13          this.x = x; this.y = y;
14      }
15
16      public String printCoordinates(int x, int y) {
17          return ("{" + x + ", " + y + "}");
18      }
19
20      public String printCoordinates(ChessCoordinates
        chessCoordinates) {
21          return ("{" + chessCoordinates.x + ", " + chessCoordinates.
        y + "}");
22      }
23
24      @Override
25      public boolean equals(Object obj) {
26          ChessCoordinates chessCoordinates = (ChessCoordinates) obj;
27          if(this.x == chessCoordinates.x && this.y ==
        chessCoordinates.y){
28              return true;
29          }else{
30              return false;
31          }
32      }
33
34  }
```

Listing 3: Chess Coordinates.java

2.2.4 Módulo Cavalo - ChessKnights.java

Módulo responsável pelo tratamento dos atributos e funções específicas relativas ao cavalo.

Funções presentes

- **Construtor:** Instancia a peça com as coordenadas iniciais definidas na main.
- **Calcular possíveis coordenadas:** Função que busca à partir da posição corrente, as possíveis novas coordenadas acessíveis pelo cavalo, respeitando as regras de movimentação.
- **Mostrar possíveis coordenadas:** Mostra a lista de possíveis movimentações. Muito utilizada para debugar o código.
- **É um movimento válido?:** Define se o movimento para uma nova célula é valido de acordo com as regras definidas para o contexto.
- **Já presente no caminho:** Define se a coordenada passada como parâmetro já foi inserida no caminho solução.
- **Movimente o cavalo:** À partir das validações executadas pelos demais métodos, itera sobre as possibilidades para definir a solução para o problema e retorna o resultado.

Esta classe é invocada por todos os módulos que utilizam as coordenadas do tabuleiro.

```
import java.util.ArrayList;
import java.util.List;

public class ChessKnight {

    private ChessCoordinates startCoordinates;
    private ChessCoordinates currentCoordinates;
    private List<ChessCoordinates> currentPath = new ArrayList<
    ChessCoordinates>();
    private List<ChessCoordinates> alternativePaths = new ArrayList
    <ChessCoordinates>();
    private List<ChessCoordinates> alreadyTested = new ArrayList<
    ChessCoordinates>();

    public ChessKnight(ChessCoordinates coordinates) {
        this.startCoordinates = coordinates;
        this.currentCoordinates = coordinates;
    }

    public List<ChessCoordinates> calculatePossibleMovements(
    boolean shouldVerifyInPath) {
        List<ChessCoordinates> possibleMoviments = new ArrayList<
    ChessCoordinates>();
        int X_COORD, Y_CORRD;
        ChessCoordinates possibleMoviment;

        X_COORD = this.currentCoordinates.x - 1;
        Y_CORRD = this.currentCoordinates.y - 2;
        possibleMoviment = new ChessCoordinates(X_COORD, Y_CORRD);
        if(isValidMoviment(possibleMoviment, shouldVerifyInPath)) {
            if(!possibleMoviments.contains(possibleMoviment))
                possibleMoviments.add(possibleMoviment);
        }

        X_COORD = this.currentCoordinates.x - 1;
        Y_CORRD = this.currentCoordinates.y + 2;
        possibleMoviment = new ChessCoordinates(X_COORD, Y_CORRD);
        if(isValidMoviment(possibleMoviment, shouldVerifyInPath)) {
            if(!possibleMoviments.contains(possibleMoviment))
                possibleMoviments.add(possibleMoviment);
        }

        X_COORD = this.currentCoordinates.x + 1;
        Y_CORRD = this.currentCoordinates.y - 2;
        possibleMoviment = new ChessCoordinates(X_COORD, Y_CORRD);
        if(isValidMoviment(possibleMoviment, shouldVerifyInPath)) {
            if(!possibleMoviments.contains(possibleMoviment))
                possibleMoviments.add(possibleMoviment);
        }

        X_COORD = this.currentCoordinates.x + 1;
        Y_CORRD = this.currentCoordinates.y + 2;
        possibleMoviment = new ChessCoordinates(X_COORD, Y_CORRD);
        if(isValidMoviment(possibleMoviment, shouldVerifyInPath)) {
            if(!possibleMoviments.contains(possibleMoviment))
                possibleMoviments.add(possibleMoviment);
        }


        X_COORD = this.currentCoordinates.x - 2;
```

```
56          Y_CORRD = this.currentCoordinates.y - 1;
57          possibleMoviment = new ChessCoordinates(X_COORD, Y_CORRD);
58          if(isValidMoviment(possibleMoviment, shouldVerifyInPath)) {
59              if(!possibleMoviments.contains(possibleMoviment))
60                  possibleMoviments.add(possibleMoviment);
61          }
62
63          X_COORD = this.currentCoordinates.x - 2;
64          Y_CORRD = this.currentCoordinates.y + 1;
65          possibleMoviment = new ChessCoordinates(X_COORD, Y_CORRD);
66          if(isValidMoviment(possibleMoviment, shouldVerifyInPath)) {
67              if(!possibleMoviments.contains(possibleMoviment))
68                  possibleMoviments.add(possibleMoviment);
69          }
70
71          X_COORD = this.currentCoordinates.x + 2;
72          Y_CORRD = this.currentCoordinates.y - 1;
73          possibleMoviment = new ChessCoordinates(X_COORD, Y_CORRD);
74          if(isValidMoviment(possibleMoviment, shouldVerifyInPath)) {
75              if(!possibleMoviments.contains(possibleMoviment))
76                  possibleMoviments.add(possibleMoviment);
77          }
78
79          X_COORD = this.currentCoordinates.x + 2;
80          Y_CORRD = this.currentCoordinates.y + 1;
81          possibleMoviment = new ChessCoordinates(X_COORD, Y_CORRD);
82          if(isValidMoviment(possibleMoviment, shouldVerifyInPath)) {
83              if(!possibleMoviments.contains(possibleMoviment))
84                  possibleMoviments.add(possibleMoviment);
85          }
86
87          return possibleMoviments;
88
89      }
90
91      public void showPossibleMovements(List<ChessCoordinates>
    possibleMoviments) {
92          System.out.print("Possible moviments ");
93          System.out.println("for the current location: " + new
    ChessCoordinates().printCoordinates(this.currentCoordinates));
94          for(ChessCoordinates coord : possibleMoviments){
95              System.out.print(new ChessCoordinates().
    printCoordinates(coord));
96              System.out.println("");
97          }
98      }
99
100     public boolean isValidMoviment(ChessCoordinates
    chessCoordinates, boolean shouldVerifyInPath){
101         if((chessCoordinates.x >= 0 && chessCoordinates.x < 8) &&
102                 (chessCoordinates.y >= 0 && chessCoordinates.y < 8)
     ) {
103
104             if(!shouldVerifyInPath) {
105                 return true;
106             } else {
107                 if(isAlreadyInPath(chessCoordinates)){
108                     return false;
109                 }
110                 return true;
111             }
112         }
```

```
113         return false;
114     }
115 
116     public boolean isAlreadyInPath(ChessCoordinates
    chessCoordinates){
117 
118         if(this.currentPath.contains(chessCoordinates)) {
119             return true;
120         }
121         return false;
122     }
123 
124     public void moveKnight(ChessBoard chessBoard) {
125 
126         int numberOfIterations = 1;
127         int counter = 0;
128         if(!this.currentPath.contains(this.currentCoordinates))
129             this.currentPath.add(this.currentCoordinates);
130 
131         List<ChessCoordinates> possibleMovements =
    calculatePossibleMovements(true);
132         addElementsToAlternativePaths(possibleMovements.subList(1,
    possibleMovements.size()));
133 
134         while(!possibleMovements.isEmpty() && counter <= 63){
135             numberOfIterations ++;
136 
137             if(chessBoard.board[this.currentCoordinates.x][this.
    currentCoordinates.y] == "--") {
138                 chessBoard.board[this.currentCoordinates.x][this.
    currentCoordinates.y] = String.format("%02d", counter  );
139                 counter ++;
140             }
141 
142             this.currentCoordinates = possibleMovements.get(0);
143             if(!this.currentPath.contains(possibleMovements.get(0))
    )
144                 this.currentPath.add(possibleMovements.get(0));
145 
146             addElementsToAlternativePaths(possibleMovements.subList
    (1, possibleMovements.size()));
147 
148             possibleMovements = calculatePossibleMovements(true);
149 
150             if(possibleMovements.isEmpty()) {
151                 while(possibleMovements.isEmpty()) {
152                     if(this.alternativePaths.size() > 0){
153                         this.currentCoordinates = this.
    alternativePaths.get(this.alternativePaths.size() - 1);
154                         this.alternativePaths.remove(this.
    alternativePaths.size() - 1);
155                         possibleMovements =
    calculatePossibleMovements(false);
156                     }else{
157                         break;
158                     }
159                 }
160             }
161         }
162         chessBoard.printChessBoard();
163         System.out.println("Number of iterations " +
    numberOfIterations);
```

```
164     }
165
166     public void addElementsToAlternativePaths(List<ChessCoordinates
        > list) {
167         for(ChessCoordinates coord : list) {
168             if(!this.alternativePaths.contains(coord) && !this.
        alreadyTested.contains(coord)){
169                 this.alternativePaths.add(coord);
170                 this.alreadyTested.add(coord);
171             }
172         }
173     }
174
175 }
```

Listing 4: Chess Knight.java

2.3 Resultados

```
1  Starting Tour from {4, 0}
2  Board final configuration
3   -------------------------
4  | 04 09 02 27 00 31 56 62 |
5  | 07 20 05 30 47 28 49 40 |
6  | 10 03 08 01 26 41 32 55 |
7  | 21 06 19 46 29 48 39 50 |
8  | 18 11 16 25 42 33 54 59 |
9  | 15 22 13 36 45 38 51 60 |
10 | 12 17 24 43 34 53 61 57 |
11 | 23 14 35 63 37 44 58 52 |
12  -------------------------
13 Number of iterations 92
14 It took 53 ms
15
16 Starting Tour from {2, 3}
17 Board final configuration
18  -------------------------
19 | 27 08 25 10 39 16 45 60 |
20 | 24 01 28 17 30 51 38 58 |
21 | 07 26 09 40 11 44 15 46 |
22 | 02 23 00 29 18 31 50 37 |
23 | 63 06 41 12 43 14 47 59 |
24 | 62 03 22 19 32 54 36 49 |
25 | 21 34 05 42 13 48 56 53 |
26 | 04 57 20 33 55 35 61 52 |
27  -------------------------
28 Number of iterations 123
29 It took 12 ms
30
31 Starting Tour from {5, 6}
32 Board final configuration
33  -------------------------
34 | 05 10 03 28 43 32 51 59 |
35 | 08 21 06 31 46 29 44 41 |
36 | 11 04 09 02 27 42 33 52 |
37 | 22 07 20 47 30 45 40 56 |
38 | 19 12 17 26 01 34 53 58 |
39 | 16 23 14 37 48 39 50 62 |
40 | 13 18 25 60 35 00 57 54 |
41 | 24 15 36 63 38 49 55 61 |
42  -------------------------
```

```
Number of iterations 126
It took 12 ms

Starting Tour from {4, 6}
Board final configuration
 -------------------------
| 12 03 10 44 32 42 61 47 |
| 09 22 13 41 15 45 55 56 |
| 04 11 02 31 43 33 48 37 |
| 23 08 21 14 40 16 46 54 |
| 20 05 30 01 34 49 36 60 |
| 27 24 07 57 17 39 52 53 |
| 06 19 26 29 00 35 50 38 |
| 25 28 63 18 58 59 62 51 |
 -------------------------
Number of iterations 122
It took 14 ms

Starting Tour from {0, 5}
Board final configuration
 -------------------------
| 05 22 03 24 38 30 40 57 |
| 02 15 06 31 63 54 59 52 |
| 21 04 23 37 25 39 29 41 |
| 16 01 14 07 32 45 53 58 |
| 13 20 11 26 36 28 42 56 |
| 00 17 08 33 44 47 51 55 |
| 62 12 19 10 27 35 49 46 |
| 18 09 34 43 48 61 60 50 |
 -------------------------
Number of iterations 113
It took 10 ms

Starting Tour from {1, 4}
Board final configuration
 -------------------------
| 13 02 11 04 39 30 37 58 |
| 10 23 14 29 36 45 62 59 |
| 01 12 03 40 05 38 31 44 |
| 24 09 22 15 28 35 46 54 |
| 21 00 19 06 41 32 43 57 |
| 08 25 16 27 52 47 34 56 |
| 63 20 07 18 33 42 49 50 |
| 61 17 26 60 48 51 55 53 |
 -------------------------
Number of iterations 127
It took 11 ms
```

3 Parte 2 - Pesquisa

3.1 Coesão Lógica

No conceito de coesão lógica, módulos são agrupados quando desempenham mesma função mesmo que para objetos de natureza distintas.

3.2 Coesão Temporal

A coesão temporal se limita ao fluxo específico de um módulo, uma vez que depende somente do contexto em que determinada função é chamada e a ordem com que invoca as funções sucessivas. Consiste no agrupamento do fluxo de determinada chamada dentro de determinada e específico caso de uso.

3.3 Coesão Processual

Agrupamento de módulos de uma determinada solução a partir de fluxos comuns de execução.

3.4 Coesão Sequencial

Agrupamento de módulos a partir da forma e da ordem de como os dados de entrada e saída são apresentados.

3.5 Coesão Funcional

Partes de um módulo ou classes são agrupados porque todos contribuem a uma única tarefa definida do módulo.

3.6 Acoplamento de conteúdo

Dois módulos apresentam acoplamento de conteúdo se um dos módulos faz referência ao conteúdo de outro módulo.

3.7 Acoplamento comum

Acoplamento comum é definido aos módulos se estes fazer referência a mesma área de dados.

3.8 Acoplamento de controle

Uma vez que um módulo é necessário para fornecer a outro os parâmetros necessários para controlar o fluxo neste, este acoplamento é definido como acoplamento de controle.

3.9 Acoplamento de carimbo

Assim como o acoplamento de dados, o acoplamento de carimbo é definido pela comunicação entre módulos através de dados passados por parâmetros, porém quando um deles é um tipo de dados específico.

3.10 Acoplamento de dados

Quando módulos se comunicam através de dados passados como parâmetros, dizemos que existe um acoplamento de dados.

References

[1] Qualidade de Projeto. http://www.ic.unicamp.br/ thelma/inf327/aulas-2008/Slides-Aulas/AULA6-quali.pdf

[2] Projeto de software. http://www.inf.ufpr.br/andrey/ci163/IntroduzProjetoAl.pdf

[3] Coesão e Acoplamento. https://wpjr2.wordpress.com/2008/04/22/coesao-e-acoplamento/