

Trabalho prático: Relatório

PERT/CPM Project

Estruturas de Informação (ESINF)

Docente: Ana Maria Madureira (AMD)

Guilherme Sousa nº1130638

João Oliveira nº1130537

Data: 6/10/2015

# Índice

1

1ªa Parte, 4

B

Bibliografia, 9

C

Complexidade Temporal (Big-Oh), 5

Conclusão, 9

D

Desenvolvimento, 4

Diagrama de Classes, 4

Diagrama de Sequência, 5

I

Introdução, 3

T

Testes e Resultados, 8

# Introdução

Neste semestre, na cadeira de Estruturas de Informação vamos desenvolver um projeto usando a “ferramenta” de PERT (Project Evaluation and Review Technique) em conjunto com um algoritmo de agendamento de um conjunto de atividades do projeto, também conhecido por CPM (Critical Path Method).O objetivo deste trabalho é especificar e modular uma aplicação de gestão de atividades de projeto, assim como providenciar um conjunto particular de funcionalidades típicas destas aplicações. Nesta primeira parte do trabalho vamos tirar partido da programação orientada a objetos e começar por implementar classes simples, que posteriormente vão fazer parte do sistema final.

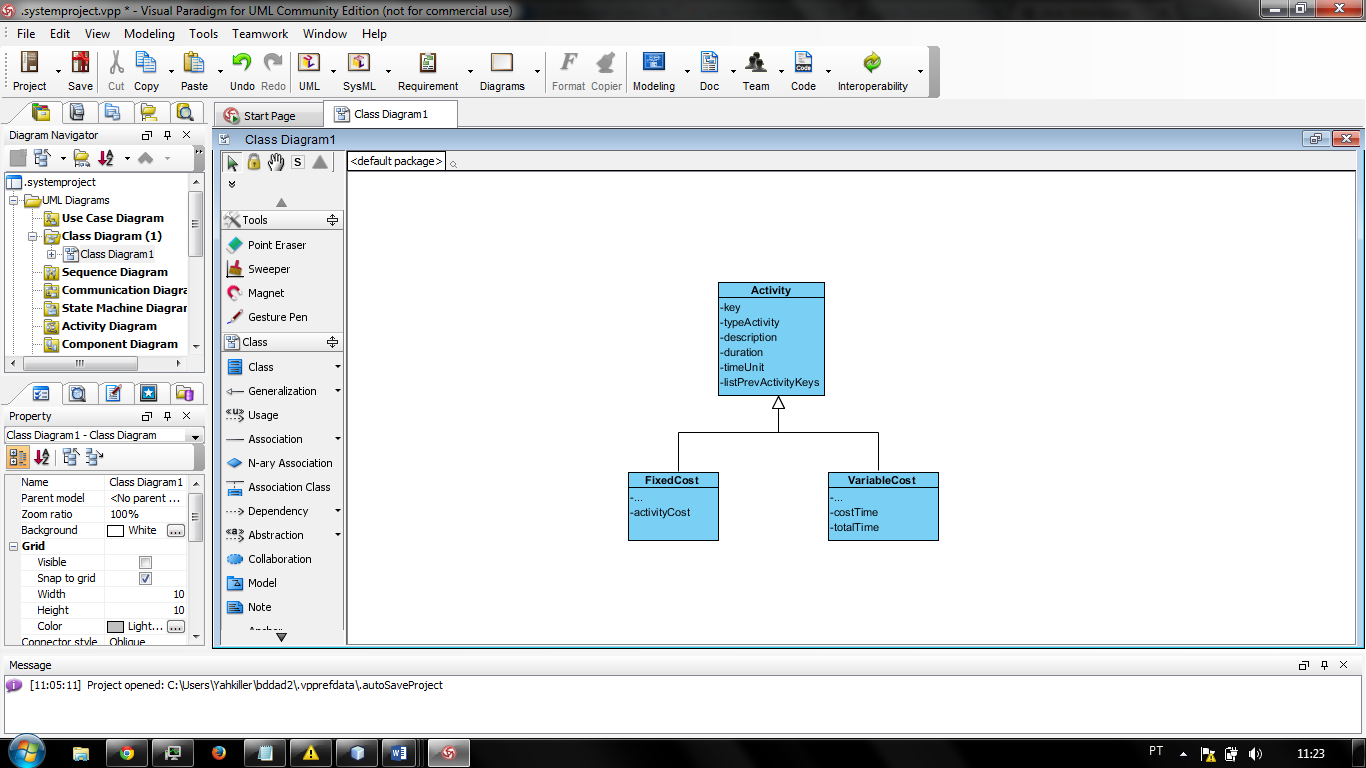
# Desenvolvimento

# 1ªa Parte

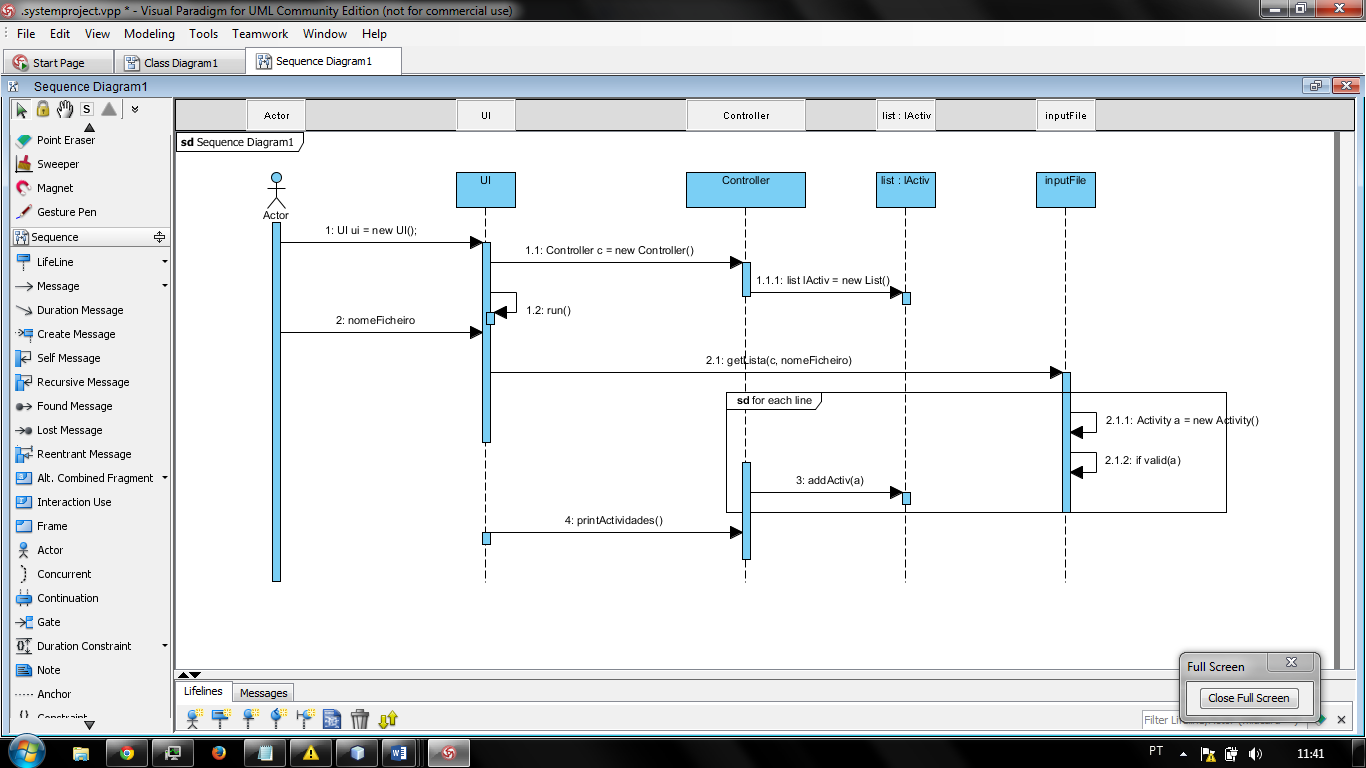
Nesta primeira parte tivemos como objetivo a criação das classes actividade, actividade fixa e actividade variavel. Consideramos uma relação relação de polimorfismo já que actividade fixa e variável partilham alguns atributos e são ambas atividades.

Além disso também tivemos de implementar a leitura de actividades de um ficheiro de texto tendo como base um exemplo dado.

# Diagrama de Classes



# Diagrama de Sequência



# Complexidade Temporal (Big-Oh)

|  |  |
| --- | --- |
| **Método getLista() da classe inputFile** | **O()** |
| int paramFixed = 6; | 1 |
| int paramVar=7; | 1 |
|  |  |
| BufferedReader br = new BufferedReader(new FileReader(localizacao)); | 1 |
| try { |  |
|  |  |
| String line = br.readLine(); | 1 |
|  |  |
| while (line != null) { | **n** |
| String[] parts = line.split(","); | n |
|  |  |
| if (parts[1].equalsIgnoreCase("fca")) { | n |
|  |  |
| if (parts.length > paramFixed) { | n |
| **addActivFixed(paramFixed, parts, controller);** | n**\*n** |
| } else { |  |
| fixedCost fixed = new fixedCost(parts[0], parts[1], parts[2], Integer.parseInt(parts[3]),parts[4], Integer.parseInt(parts[5])); | n |
| **controller.addActiv(fixed);** | **n\*n** |
| } |  |
| } |  |
| if (parts[1].equalsIgnoreCase("vca")) { | n |
| **addActivVar(paramVar, parts, controller);** | **n\*n** |
| }else{ |  |
| variableCost variable = new variableCost(parts[0], parts[1], parts[2], Integer.parseInt(parts[3]), parts[4], Integer.parseInt(parts[5]), Integer.parseInt(parts[6])); | n |
| **controller.addActiv(variable);** | **n\*n** |
| } |  |
| } |  |
| line = br.readLine(); | n |
| } |  |
| } finally { |  |
| br.close(); | 1 |
| } |  |
| **Complexidade** | **n2** |

Este método é determinístico pois independentemente dos valores obtidos vai apresentar sempre a mesma complexidade. E tem como complexidade temporal **n2** pois implementa um while e dentro desse while vão ser executados métodos para adicionar uma variável (addActivVar/addActivFixed ou controller.addActiv()), e esses métdos têm complexidade n. Logo a complexidade é de **n\*n = n2**.

|  |  |
| --- | --- |
| **Método addActivFixed da casse inputFile** | **O()** |
| List<String> listPrevActKeys = new ArrayList(); | 1 |
| for (int i = paramFixed; i < parts.length; i++) { | **1, n+1, n** |
| String prevKey = parts[i]; | n |
| if (controller.hasActivity(prevKey) && (!listPrevActKeys.contains(prevKey))) { | n |
| listPrevActKeys.add(prevKey); | n |
| } else { |  |
| System.out.println("A prevKey '" + prevKey + "' da Actividade '" + parts[0] + "' n existe!"); |  |
| } |  |
| } |  |
|  |  |
| fixedCost fixed = new fixedCost(parts[0], parts[1], parts[2], Integer.parseInt(parts[3]), parts[4], Integer.parseInt(parts[5]), listPrevActKeys); | 1 |
| **controller.addActiv(fixed);** | **n** |
| } |  |
| **Complexidade** | **n** |

Este método é determinístico pois independentemente dos valores obtidos vai apresentar sempre a mesma complexidade. E tem como complexidade temporal **n** pois implementa um “for” que percorre os ‘n’ parametros que contêm prevKeys.

|  |  |
| --- | --- |
| **Método addActivVar da casse inputFile** | **O()** |
| List<String> listPrevActKeys = new ArrayList(); | 1 |
| for (int i = paramVar; i < parts.length; i++) { | **1, n+1, n** |
| String prevKey = parts[i]; | n |
| if (controller.hasActivity(prevKey) && (!listPrevActKeys.contains(prevKey))) { | n |
| listPrevActKeys.add(prevKey); | n |
| } else { |  |
| System.out.println("A prevKey '" + prevKey + "' da Actividade '" + parts[0] + "' n existe!"); |  |
| } |  |
| } |  |
| if (validacaoUnid(parts[4])) { | 1 |
| variableCost variable = new variableCost(parts[0], parts[1], parts[2], Integer.parseInt(parts[3]), parts[4], Integer.parseInt(parts[5]), Integer.parseInt(parts[6]), listPrevActKeys); | 1 |
| **controller.addActiv(variable);** | **n** |
| } |  |
| Complexidade | **n** |

Este método é determinístico pois independentemente dos valores obtidos vai apresentar sempre a mesma complexidade. E tem como complexidade temporal **n** pois implementa um “for” que percorre os ‘n’ parametros que contêm prevKeys.

|  |  |
| --- | --- |
| **Método addActiv da classe Controller** | **O()** |
| if (!hasActivity(a.getKey())) { | **n** |
| lActiv.add(a); | 1 |
| return true; | 1 |
| } else { |  |
| return false | 1 |
| } |  |
| **Complexidade** | **n** |

Este método é determinístico pois independentemente dos valores obtidos vai apresentar sempre a mesma complexidade. E tem como complexidade temporal **n** pois implementa o método hasActivity() descrito a seguir.

|  |  |
| --- | --- |
| **Método hasActivity da classe Controller** | **O()** |
| for (int i = 0; i < lActiv.size(); i++) { | **1, n+1, n** |
| if(key.equalsIgnoreCase(lActiv.get(i).getKey())){ | **n** |
| return true; | 1 |
| } |  |
| } |  |
| return false | 1 |
| **Complexidade** | **n** |

Este método é determinístico pois independentemente dos valores obtidos vai apresentar sempre a mesma complexidade. E tem como complexidade temporal **n** pois implementa um “for” que percorre as ‘n’ actividades existentes na lista.

# Testes e Resultados

Foram realizados testes a todas as classes implementadas, de maneira a verificar a eficácia da nossa ainda simples aplicação. Como se pode verificar ao testar todo o projeto, os resultados são excelentes já que todos os testes são aprovados, sinal que evidencia a boa codificação do projeto.

# Conclusão

Após a realização da primeira parte do projeto, concluímos que as tarefas por nós desenvolvidas nesta reta inicial, são bastante “simples” em comparação às que virão no futuro. Apoiamos esta afirmação nos resultados apresentados em cima, e que estão relacionados com a complexidade temporal dos métodos que codificamos.

# Bibliografia

<https://moodle.isep.ipp.pt/acesso/>