INSTITUTO SUPERIOR TÉCNICO

MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

# Programação Orientada por Objetos

# Projeto Final

*Grupo: 39*

*Afonso Soares nº 81086*

*Miguel Rodrigues nº 76176*

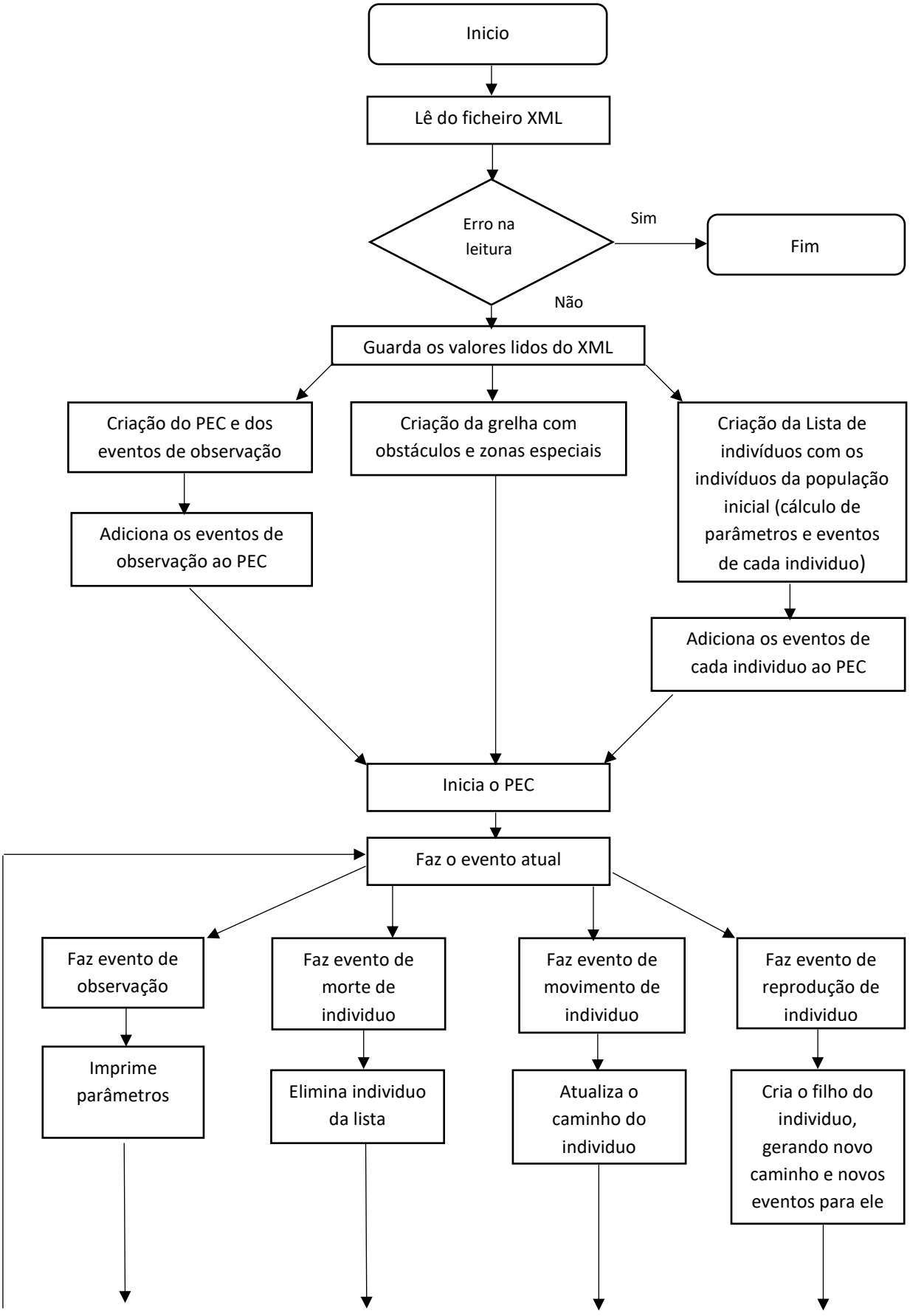
***11 de Maio de 2018***

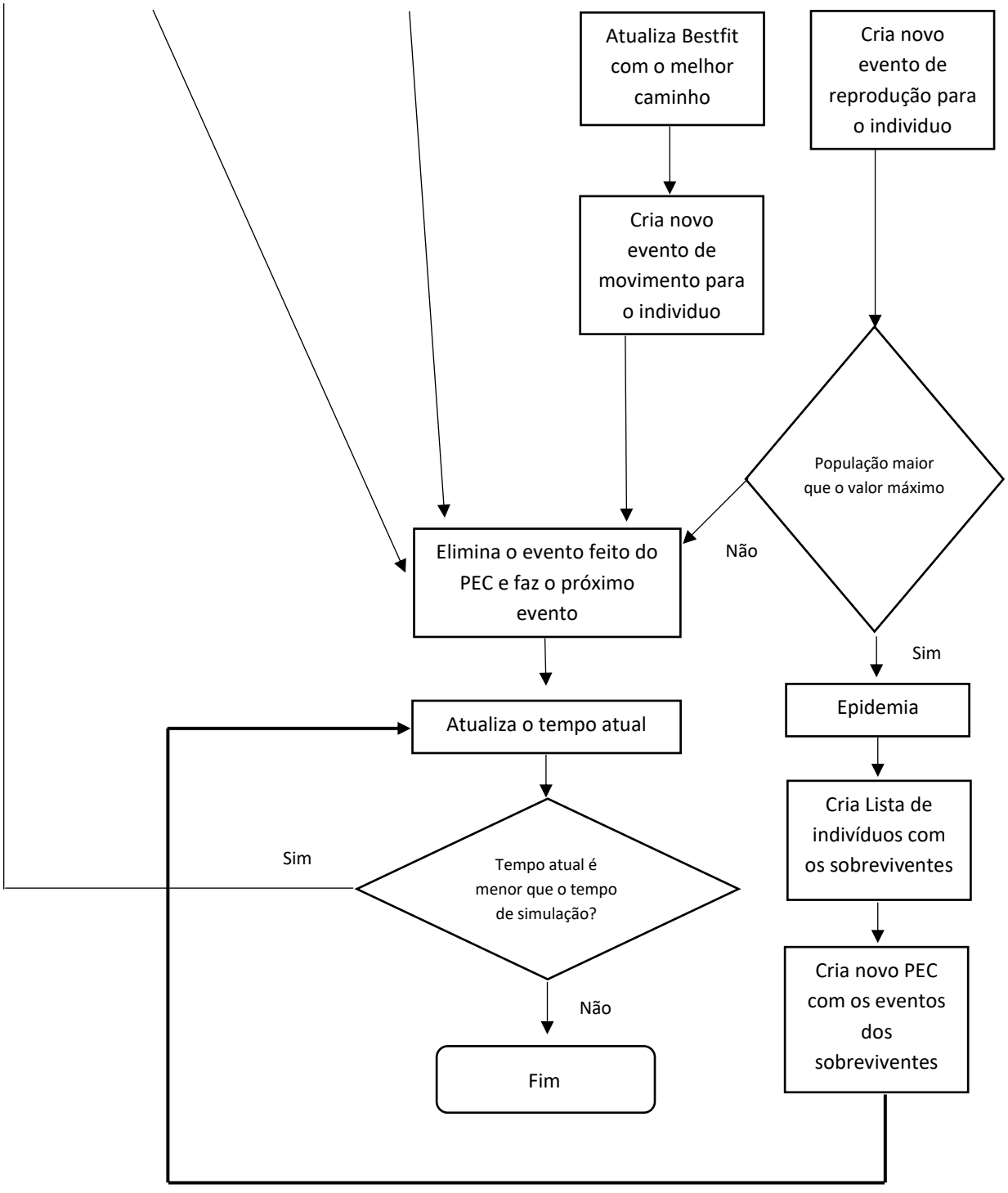
# 1. Objetivo

Este projeto tem como objetivo criar um programa, que seja extensível e com níveis de abstração suficientes permitindo um desenvolvimento por parte de outros utilizadores sobre ele, capaz de realizar o andamento de uma população sobre um mapa. Existindo uma grelha de pontos 2D que definem um mapa, mapa esse que possui obstáculos e arestas de custo especial, pretende-se que dando uma posição inicial e uma posição final, os indivíduos sejam capazes de tentar atingir essa posição final, sendo que o andamento destes sobre o mapa é feito de forma aleatória. Cada individuo possui 3 eventos, um para o seu movimento, um para a sua reprodução e um para a sua morte. Cada evento tem um tempo associado, tempo esse que é usado num PEC, *Pending Event Container*, onde os eventos são feitos por ordem temporal. Sempre que um individuo se reproduz, um filho é criado com parte do caminho do pai. Existe também um valor máximo para a população de indivíduos que possibilita a geração de uma epidemia, caso o número de indivíduos seja maior que o valor máximo de população. Essa epidemia permite que sobrevivam os 5 indivíduos com maior conforto enquanto que o resto sobrevive ou morre com base numa relação aleatória. Por fim existem eventos de observação que indicam o estado da simulação apresentado valores como o melhor caminho, o instante atual e se o ponto final foi atingido ou não.

Posto isto, foi realizado um programa que lê os parâmetros necessários à simulação, cria todas as estruturas necessárias para guardar os indivíduos, o mapa, o PEC, os eventos, entre outros, e inicia o PEC correndo os eventos por ordem temporal, sendo que novos eventos podem ser adicionados ao PEC, até um instante final de simulação, até que não haja mais eventos sem ser os de observação antes do tempo final de simulação ou até que morram todos os indivíduos, não havendo, assim, mais eventos a realizar.

## 2. Diagrama


Inicio
Lê do ficheiro XML
Erro na leitura
Sim
Fim
Não
Guarda os valores lidos do XML
Criação do PEC e dos eventos de observação
Criação da grelha com obstáculos e zonas especiais
Criação da Lista de indivíduos com os indivíduos da população inicial (cálculo de parâmetros e eventos de cada individuo)
Adiciona os eventos de observação ao PEC
Adiciona os eventos de cada individuo ao PEC
Inicia o PEC
Faz o evento atual
Faz evento de observação
Faz evento de morte de individuo
Faz evento de movimento de individuo
Faz evento de reprodução de individuo
Imprime parâmetros
Elimina individuo da lista
Atualiza o caminho do individuo
Cria o filho do individuo, gerando novo caminho e novos eventos para ele


Atualiza Bestfit com o melhor caminho
Cria novo evento de reprodução para o individuo
Cria novo evento de movimento para o individuo
População maior que o valor máximo
Não
Sim
Elimina o evento feito do PEC e faz o próximo evento
Epidemia
Atualiza o tempo atual
Cria Lista de indivíduos com os sobreviventes
Sim
Tempo atual é menor que o tempo de simulação?
Não
Cria novo PEC com os eventos dos sobreviventes
Fim

## 3. Estruturas de Dados, Implementação e Exceções

| Pacotes | Classes | Função |
|---|---|---|
| application | *Application* | Possui o método *main*. Apanha as exceções relacionadas com o *parser* e com os inputs e outputs do sistema. De forma à aplicação ser extensível a um desenvolvimento, a iniciação dos elementos usados na simulação e a iniciação do PEC, ou seja, que este comece a correr os seus eventos, são feitas separadamente. |
| pec | *Event* | Esta classe implementa um *comparable*, pois o PEC utiliza uma *ArrayList* de *Events* ordenados com base no tempo, ascendentemente. |
| simulation | *XML* | Faz a leitura do ficheiro XML e verificação de erros no ficheiro lido, como os limites dos valores que são passados, mas não utiliza exceções para estas ações. |
| | *EvObservation* | É uma subclasse de *Event* que realiza a impressão dos parâmetros de observação. É uma extensão dos eventos pois apenas se refere a estes. |
| | *Grid* | É uma subclasse de *AGrid*. Desta forma a implementação da grelha é mais extensível, sendo que esta classe possui os obstáculos e zonas de custo especial. Os obstáculos são guardados |

| | | |
|---|---|---|
| | | numa *ArrayList*. As arestas com custo especial, maior que 1, são guardadas noutra *ArrayList*. |
| individual | *EvIndividual* | É uma subclasse de *Event* que possui o tempo do evento e a referência do individuo. É uma extensão dos eventos porque apenas se refere a um evento e um evento pode ser de um individuo ou uma observação. |
| | *EvDeath, EvMove e EvReproduction* | São subclasses de *EvIndividual* pois permite fazer extensões de mais eventos para o individuo e referem-se apenas ao individuo. |
| | *Individual* | É uma subclasse da classe abstrata *AIndividual.* Implementa um *comparable* para que se possa ordenar a lista de indivíduos por conforto. |
| | *IndividualList* | É uma subclasse da classe abstrata *AIndividualList.* Define a Lista de indivíduos que é uma *ArrayList* que é ordenada apenas quando ocorrem epidemias. |

# 4. Interfaces

Existem 2 interfaces no projeto, a *IPEC* e a *ISimulation*. Estas interfaces têm como objetivo permitir uma maior extensibilidade do código.

A interface *IPEC* possui os métodos que permitem atuar sobre um PEC, como a adição e remoção de eventos, a adição de uma lista de eventos, a criação de um PEC, o número de eventos no PEC, o próximo evento a ser corrido na lista do PEC e a iniciação do PEC, ou seja, começar a correr por ordem os eventos que se encontram no seu interior. Esta interface define, portanto, métodos gerais para um PEC.

A interface *ISimulation* define os métodos que iniciam uma simulação. De forma ao código ser mais extensível, esta interface indica os métodos para a criação e adição de um mapa para os indivíduos, construção de uma lista com indivíduos e adição de um novo PEC à estrutura que detém os parâmetros da simulação, sendo que as particularidades de cada método irão depender da implementação feita pelo programador.

## 5. Abstração

De forma a tornar o código desenvolvido mais extensível a novas implementações por parte de outros utilizadores, procurou-se que o código fosse abstrato o suficiente para que novos utilizadores possam alterar a forma como querem as estruturas e atuam sobre elas.

Posto isto, foram criadas algumas classes abstratas que permitem novas implementações:

| Pacotes | Classes | Função |
|---|---|---|
| pec | *Event* | É uma classe abstrata de forma a permitir que novos utilizadores criem outro tipo de eventos. Cada evento tem apenas um tempo a ele associado para que possa fazer parte do PEC. |
| simulation | *AGrid* | Esta classe possui todos os elementos associados ao mapa e os métodos que permitem atuar sobre o mapa, como adicionar obstáculos e zonas especiais. Estes métodos são abstratos para que novos utilizadores possam adaptar à sua estrutura. |
| individual | *EvIndividual* | Tal como a classe *Event* é abstrata para que cada individuo possa ter outros eventos associado a ele. |
| | *AIndividual* | Possui todos os parâmetros necessários ao individuo e os métodos abstratos que gerem o seu conforto, os seus eventos, os seus filhos, e para os |

| | | |
|---|---|---|
| | | comparar com base no conforto. Desta forma o utilizador tem uma maior extensibilidade na forma como pode implementar os indivíduos. |
| | *AIndividualList* | À semelhança da *AIndividual* possui os parâmetros que são gerais aos indivíduos, e os métodos abstratos que, quando implementados, vão permitir realizar ações sobre a Lista, como adicionar indivíduos, comparar o melhorar individuo com base no conforto. Esta abstração demonstra ao utilizador as funcionalidades da Lista, mas esta classe implementada desta forma permite que o utilizador adapte a Lista às suas estruturas. Esta classe possui também a atualização do individuo com o melhor caminho. |

# 6. Resultados

Em anexo ao código encontram-se 5 testes feitos ao nosso programa. Esses 5 testes procuram analisar a rapidez e correção da execução do código, se os eventos dos indivíduos e de observações são realizados, se ocorrem epidemias quando devem ocorrer e se são bem realizadas, se o programa continua mesmo depois de um individuo achar o ponto final, se o programa envia para o terminal o que se pretende, se o programa acaba no fim do tempo de simulação, se o programa acaba caso não hajam mais eventos até ao fim do tempo de simulação, tirando os de observação, e se acaba caso todos os indivíduos morram (o que leva a que não existam mais eventos a realizar). Foram também analisadas as exceções existentes no programa para erros no *SAXparser*, nos ficheiros de entrada e saída e nos valores lidos no ficheiro XML que serão guardados em variáveis necessárias à simulação.

Posto isto estes testes, e outros feitos também, demonstram uma boa *performance* do nosso programa e que este realiza o que é pretendido. Dada a abstração realizada, o programa pode funcionar com estruturas diferentes, o que pode alterar ou não o desempenho do mesmo.

# 7. Conclusões

Finalizado este Projeto constata-se que o código feito realiza tudo o que é pretendido e possui alguma análise de erros, dando *feedback* ao utilizador do que está mal. Além disto procurou-se que este fosse extensível para novas implementações e desenvolvimentos por parte de outros utilizadores, e possui também métodos de fácil acesso através das interfaces realizadas.

Em suma, soluções de Programação Orientada por Objetos foram utilizadas de forma a tornar o código mais extensível, mais funcional e mais bem estruturado.