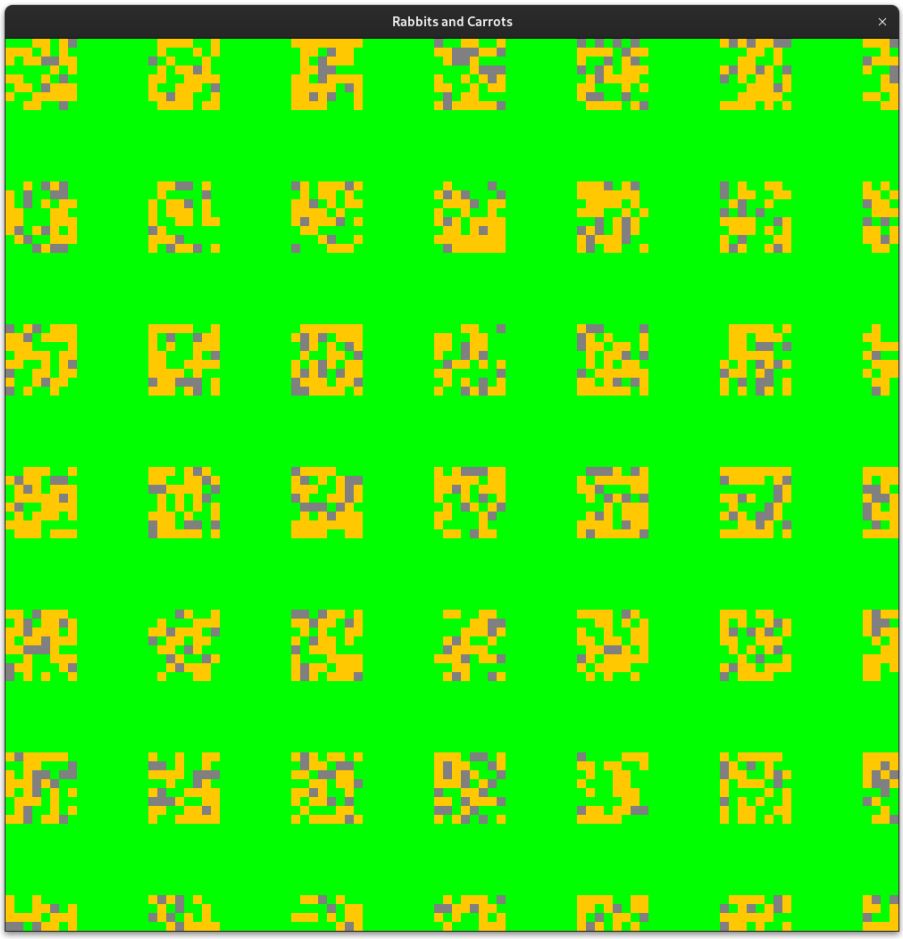
UNIVERSIDADE
DE ÉVORA

Relatório Programação de II

David Marinho
54560

Axel Amoroso Carapinha
55248



Contents

1 Abstract 2
2 Resumo do projeto 2
3 Introdução 2
4 Desenvolvimento 2
4.1 Decisões de implementação 2
4.2 Principais dificuldades 3
5 Procedimentos 4
6 Design Patterns 4
6.1 Execução 7
6.2 Output 7
7 Conclusão 7

1 Abstract

Este trabalho teve como principal objetivo usar Java e o paradigma da programação orientada a objetos (POO) para recriar as regras de uma versão do jogo da vida (de John Conway) com cenouras, coelhos e ervas, sendo usada a biblioteca AWT e a framework Swing, ambas de Java, para mostrar os resultados.

2 Resumo do projeto

O principal objetivo deste projeto foi preencher um prado verde retangular com coelhos e cenouras nas células indexadas, também retangulares.
O prado tem extremidades adjacentes, como num toro topológico.
As (no fundo únicas) entidades deste prado interagem entre si através das regras aplicadas, podendo nascer, sobreviver, morrer e, apenas no caso dos coelhos, passar fome.
Assim, cada prado é definido pelo tamanho, pela posição inicial dos coelhos, pelas cenouras e pelo tempo de fome *starveTime*, que define quanto tempo os coelhos sobrevivem sem comer cenouras.
Tal como no jogo da vida de John Conway, diferentes ajustes nestas definições do prado geram resultados variadíssimos, e padrões complexos podem surgir de regras tão básicas como as utilizadas, e resumidas em *timeStep()*: o mais importante método de instância de cada objeto da classe *Grassland*.

3 Introdução

O objetivo deste relatório é mostrar a estrutura do programa, além de explicar com mais detalhe pormenores de implementação e resumir os resultados.

4 Desenvolvimento

4.1 Decisões de implementação

- Foi usado apenas um array bidimensional.
- Cada entidade tem um ID, que no fundo serve para identificar a classe, o que permite uma conexão facilitada com o ficheiro *Simulation.java*. Acaba por ser a utilização das constantes inteiras de classe GRASS, RABBIT e CARROT.
  Por conta disso, *cellContents()* retorna o ID do objeto retornado pelo método *getCell()* que retorna o endereço da entidade guardado no array do prado instanciado.
- O método de instância *startGrasslandLife()* contém código comentado que se mostrou notoriamente útil na verificação das regras deste projeto e uma lógica com o mesmo intuito da presente no ficheiro *Simulation.java*: preencher o prado aleatoriamente.

– Foram mantidos muitos métodos de instância com o modificador *public* para serem acedidos pelo ficheiro *Simulation.java*. Os únicos métodos privados estão relacionados com implementações das regras e cálculos auxiliares para as mesmas.

– Foi alterada a ordem de utilização dos argumentos dos métodos que acedem ao array devido a preferirmos uma interpretação diferente de posições no array (linha (*row*) e coluna (*column*)), mas mantendo a compatibilidade com a interpretação presente no ficheiro *Simulation.java* (*x*, *y*).

– Foi criada uma classe própria para excepções relacionadas com a ausência de coelhos no prado, para alertar o utilizador de situações que poderiam parecer estranhas, e para conseguir implementar mais partes da matéria lecionada, outro dos objetivos deste projeto.

– Utilizou-se uma JFrame em vez de uma Frame, por ser mais leve em termos de utilização de recursos e por possuir mais funcionalidades.

4.2 Principais dificuldades

Em primeiro lugar, surgiu uma dúvida: seria melhor usarmos um array intermediário ou instanciar e alterar diretamente o novo Grassland?

Por um lado, a primeira opção parecia adequada quando foi primeiramente implementada: teriamos o 'meadowArr', uma variável de instância (um array) do atual Grassland no qual as regras eram baseadas, e o 'newMeadowArr', também variável de instância, mas esta última para se escreverem os resultados da aplicação dessas mesmas regras.

Mas isso originou vários problemas. Com essa implementação faria falta copiar os dados desse 'newMeadowArr' para o *Grassland* instanciado, que à parte de ineficiente, seria menos prático. Além disso, tornou-se mais complicada a sincronização dos dados, devido à desnecessaria complexidade da lógica. E ainda, como todos os objetos instanciados referenciavam o seguinte objeto, preocupou-nos que o *garbage collector* do Java não considerasse o espaço ocupado por prados inutilizados (e por todas as entidades nele criadas, que são mais espaço que referencia) como livre. Isso não só seria um grave problema de memória, como vai contra a natureza do objetivo do próprio programa, pois resultados complexos começam a surgir após um certo número de iterações, e não deve ser um problema de memória a definir quando é que a simulação deveria parar.

Por outro lado, o modo, implementado e mais prático, foi instanciar o novo Grassland através do atual, o que se provou útil em todas as fases do desenvolvimento deste trabalho.

Em segundo e último lugar, surgiu a leitura diferente dos dados por parte do método *printGrassland()* e do ficheiro *Simulation.java*

Então decidimos fazer da seguinte forma para termos a certeza que os Grasslands anteriores são limpos pelo GarbageCollector.

```
public Grassland timeStep() {
  Grassland newGrassland = new Grassland(this.width, this.height, this.starveTime);
    for (int row = 0; row < this.height; row++) {
      for (int column = 0; column < this.width; column++) {
        // Atualização do novo Grassland depois do atual ter passado pelas regras.
      }
    }

  // Retornamos um novo Grassland.
  return newGrassland;
}
```

5 Procedimentos

1. Na execução do programa podem ser passados quatro argumentos por uma JFrame através de quatro objetos 'JTextField'.
    - width → O comprimento do prado.
    - height → A largura do prado.
    - starveTime → Número de timesteps a que os coelhos sobrevivem sem comer qualquer cenoura.
    - maxTime → Número máximo de gerações simuladas.
2. Um prado de comprimento i x j é preenchido com objetos do tipo 'Grass' e são gerados aleatoriamente outros objetos, objetos de tipo 'Rabbit' e 'Carrot'.
3. A cada timestep é gerado um novo prado respeitando as regras a que a geração anterior esteve sujeita.

6 Design Patterns

Recorremos aos três pilares da Programação Orientada a Objetos (encapsulamento herança e polimorfismo) criámos quatro classes para representar entidades específicas:

- LifeBeing → Classe mãe e abstrata das classes 'Rabbit', 'Carrot' e 'Grass'.
- Rabbit → Classe filha que herda 'LifeBeing'. Tem como função organizar todos os dados a respeito dos coelhos.
- Carrot → Classe filha que herda 'LifeBeing'. Organiza toda a estrutura de dados a respeito das cenouras.
- Grass → Classe filha que herda 'LifeBeing' e organiza toda a estrutura de dados a respeitos das ervas.

- Grassland → Classe responsável por organizar todos os dados a respeito do campo, marioritariamente a posição dos objetos no campo e o tempo de longevidade dos mesmos. Mantiveram-se as assinaturas dos métodos presentes na implementação base de *Grassland.java*
- Simulation → Classe responsável pela renderização das gerações.
- Main → Classe responsável pela execução do programa.

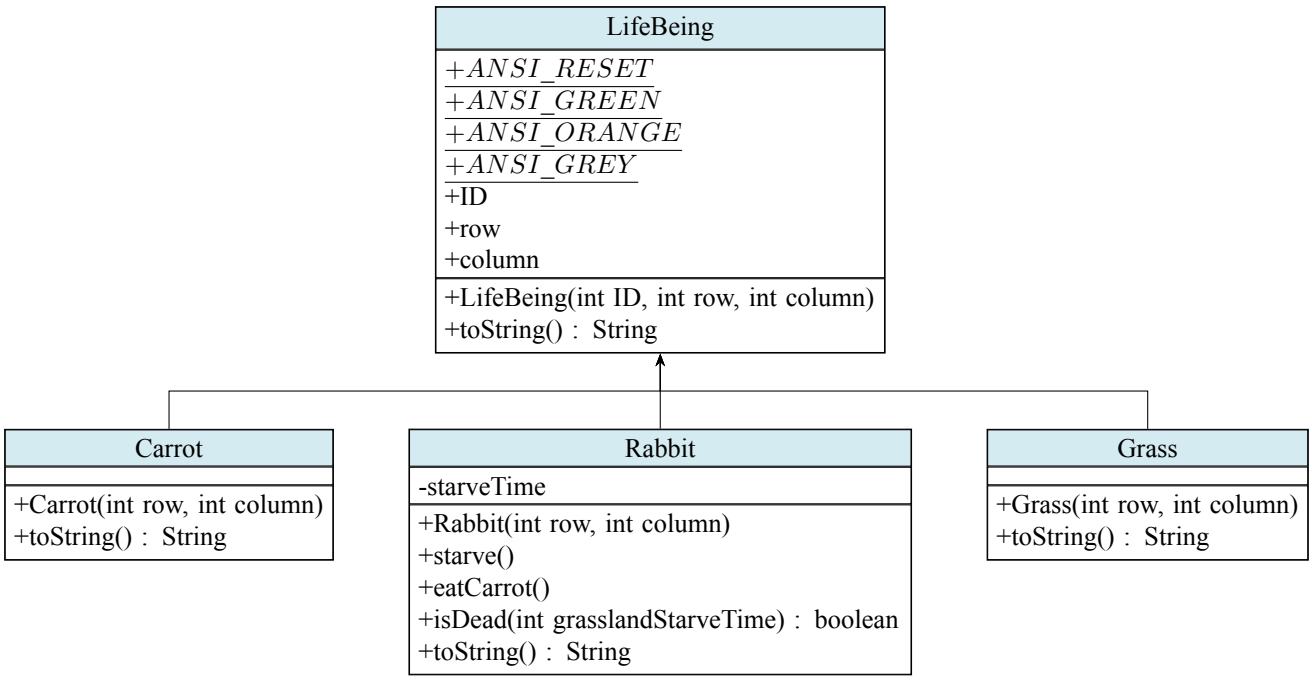


Figure 1: Diagrama de classes

Grassland
+$\underline{GRASS}$ +$\underline{RABBIT}$ +$\underline{CARROT}$ -$\underline{NEWLINE}$ -$\underline{MAX\_PERCENTAGE}$ -GRASS_PERCENTAGE -RABBIT_PERCENTAGE -CARROT_PERCENTAGE -meadowArr -width -height -starveTime
+Grassland(int i, int j, int starveTime, int grassPercentage, int rabbit Percentage, int carrotPercentage) +Grassland(int i, int j, int starveTime) +getRabbitPercentage() : int +getCarrotPercentage() : int +getGrassPercentage() : int -fillWithGrass() +startGrasslandLife() +width() : int +height() : int +starveTime() : int +addCarrot(int column, int row) +addRabbit(int column, int row) +timestep() : Grassland -grasslandRules(LifeBeing currentCell, ArrayList<LifeBeing> neighbors) : int -grassRules(LifeBeing currentCell, ArrayList<LifeBeing> neighbors) : int -rabbitRules(LifeBeing currentCell, ArrayList<LifeBeing> neighbors) : int -carrotRules(LifeBeing oldGen, ArrayList<LifeBeing> neighbors) : int -getOccurrences(ArrayList<LifeBeing> lifeBeings, int lifeId) : LifeBeing[] +cellContents(int column, int row) : int -getCell(int column, int row) : LifeBeing -collectCellNeighbors(int x, int y) : ArrayList<LifeBeing> +printGrassland(boolean cleanConsole) +printGrassland()

Figure 2: Diagrama de classes

6.1 Execução

```
javac life_beings/*.java && javac exceptions/*.java && javac main/*.java && java main.Main
```

6.2 Output

Os resultados podem ser visíveis de duas formas:

1. Na linha de comandos, permitindo a execução do método de instância *printGrassland()* no método *startGrasslandLife()*.
2. Numa janela que surge no meio do monitor.

7 Conclusão

Em suma, as principais dificuldades foram mitigadas aplicando os resultados das regras diretamente no prado (*Grassland*) instanciado, e o objetivo foi cumprido, havendo resultados visíveis da aplicação das regras da vida.

Além disso, concluiu-se que o paradigma da POO é especialmente útil para representar problemas de uma forma mais natural a como os vemos, com objetos com várias partes que interagem entre si.

Pode-se referir, por último, que regras simples demonstraram ser capazes de gerar padrões complexos, o que criou um elo de ligação entre o (quase antagónico) determinismo da tecnologia e a imprevisível natureza da vida.