Programação Avançada

2016/17

Trabalho Prático

Pretende-se implementar uma aplicação que permita ao utilizador jogar o jogo *Mini Rogue*. As regras originais do jogo encontram-se nos ficheiros anexos.

Regras gerais

O trabalho deve ser realizado por grupos de dois alunos.

A elaboração do trabalho está dividida em duas fases separadas.

As datas de entrega do trabalho nas duas fases são as seguintes:

1. Primeira fase: **14 de Maio**;
2. Segunda fase: **11 de Junho**.

As entregas correspondentes às duas fases do trabalho devem ser feitas através do *moodle* num ficheiro compactado. O nome deste ficheiro deve incluir o primeiro nome, o último nome e o número de estudante de cada um dos elementos do grupo, bem como a indicação da turma prática a que pertencem. O ficheiro deve conter o projecto NetBeans, incluindo todo o código fonte, eventuais ficheiros de dados e recursos auxiliares necessários à execução do programa, e o relatório em formato pdf.

O relatório deve incluir em ambas as fases:

1 O diagrama da máquina de estados que controla o fluxo do jogo e os diagramas de outros padrões de desenho que tenham eventualmente sido aplicados no trabalho;

2 A descrição das classes utilizadas no programa (máximo de 50 palavras / 3 linhas por classe);

3 A indicação de funcionalidades ou regras não implementadas.

Ambas as fases estão sujeitas a defesas que consistem na apresentação do trabalho. Estas podem incluir a realização de alterações ao trabalho entregue, feitas individualmente por cada um dos elementos do grupo.

À primeira fase do trabalho corresponde a cotação 3 valores e à segunda fase a cotação de 5 valores.

Funcionalidade e requisitos

As funcionalidades requeridas nas duas fases do trabalho prático são as seguintes:

1. **Primeira Fase**:
   Implementação do jogo *Mini Rogue* com interface em modo texto;
2. **Segunda Fase**:
   Implementação do jogo *Mini Rogue* com interface em modo gráfico.

As regras originais estão sujeitas às seguintes alterações:

1. A opção *Skill Check* associada à carta *Event* deixa de existir;
2. A acção *Skill Check* associada à carta *Trap* é automaticamente desencadeada;
3. É possível indicar a área em que se pretende iniciar o jogo (1 por omissão).

Em ambas as fases, deve ser possível gravar em ficheiro o jogo que se encontra a decorrer, bem como continuar um jogo previamente guardado.

A implementação do trabalho deve **obrigatoriamente** obedecer aos requisitos seguintes:

1. Deve ser seguida a arquitectura indicada na secção seguinte deste enunciado;
2. Deve ser utilizada, de forma adequada, uma máquina de estados para realizar a lógica do jogo (a Figura 1 apresenta, de um modo incompleto, uma possível abordagem);
3. A fase de combate na Figura 1 deve incluir um número de estados compreendido entre 2 e 5;
4. Na segunda fase do trabalho, deve ser utilizado o padrão MVC (Modelo-Vista-Controlador).

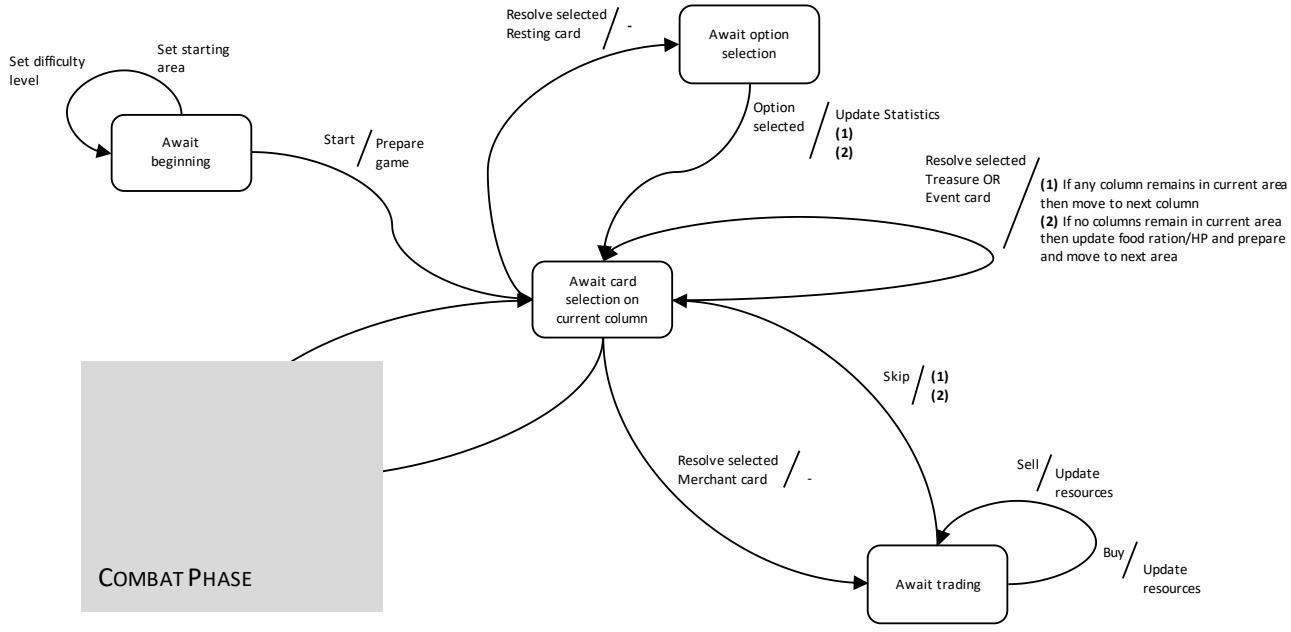

Figura 1 – Sugestão de uma máquina de estados (diagrama incompleto)

Arquitectura (fase 1)

A arquitectura da aplicação, na fase 1, deve estar condicionada de acordo com o apresentado na Figura 2.

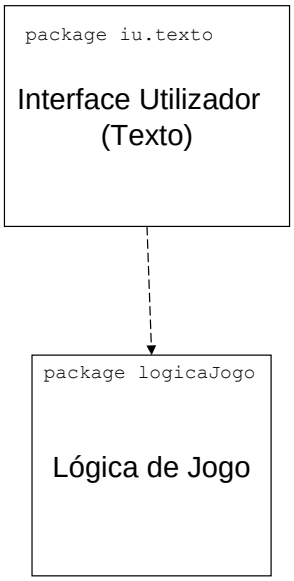

Figura 2 – Arquitectura (fase 1)

A aplicação deve estar organizada em (pelo menos) duas *packages* conforme apresentado na Figura 2. Pode existir um número qualquer de classes em cada *package*. Conforme indicado pela seta na Figura 2, a *package iu.texto* (e as classes que esta inclui) é dependente de classes definidas na *package logicaJogo*. No entanto, o inverso não é verdade. Isto significa que, em qualquer classe definida na *package iu.Texto,* podemos encontrar:

```
package iu.texto;
import logicaJogo.*;
...
```

No entanto, NÃO será possível fazer um *import* no sentido inverso:

```
package logicaJogo;
import iu.texto.*;  //ERRADO: NÃO CUMPRE A ESPECIFICAÇÃO.
...
```

Adicionalmente, cada uma das *packages* deve cumprir as seguintes restrições:

- *Package iu.texto* – Não deve ser encontrada nesta *package* qualquer lógica relacionada com o jogo e as suas regras. Por exemplo, aqui é possível imprimir o estado do jogo ou perguntar ao utilizador qual a sua jogada. No entanto, os métodos que determinam aquilo que acontece em resposta a uma jogada devem estar localizados na *package logicaJogo*;

- *Package logicaJogo* – A lógica do jogo deve estar aqui. Não pode, no entanto, realizar-se acções de leitura do teclado ou escrita no ecran no código incluído nesta *package* (ou em código chamado por código incluído nesta *package*).

Porque é que é necessário seguir esta arquitectura? Aparentemente, estas restrições só tornam as coisas mais difíceis… Para perceber isso, vamos ver o que acontece na fase 2...

Arquitectura (fase 2)

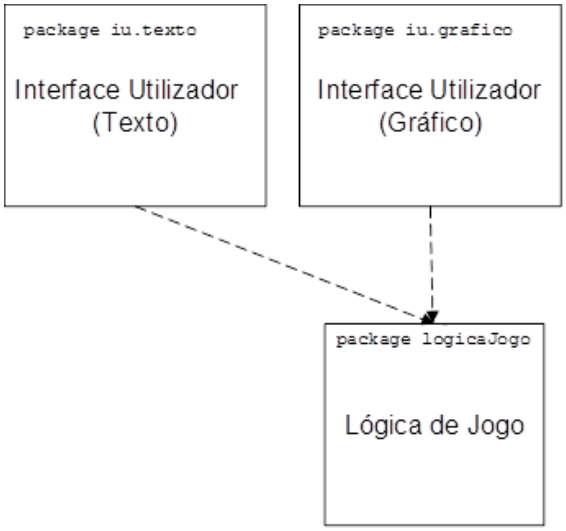

Figura 3 - Arquitectura (fase 2)

Conforme apresentado na Figura 3, é criada uma *package* adicional que fornece uma interface gráfica adequada. O que é que acontece se as indicações anteriores não forem cumpridas?

- Se existir lógica do jogo na *package iu.texto*, a *package logicaJogo* não inclui tudo o que é necessário. É, então, preciso refazê-la (a solução menos má) ou reimplementar a lógica que falta novamente na outra *package* (o que apenas adia o problema…). É preferível construir de raiz a *package logicaJogo* para que esta inclua tudo o que é necessário…
- Se a *package logicaJogo* tratar de pedir ao utilizador para que introduza informação através do teclado e/ou tratar de imprimir informação no écran, esta não pode, obviamente, ser usada por um componente que pretenda implementar uma interface gráfica ou automatizar as escolhas do jogo através de um módulo que, eventualmente, implemente um jogador virtual…

Resumindo, as restrições arquiteturais aqui descritas, que constituem um exemplo de boas práticas de programação orientada a objetos, destinam-se a poupar trabalho e maximizar as possibilidades de reaproveitar, na segunda fase, aquilo que é feito na primeira.