

Programação Avançada

2022/2023

Enunciado do Trabalho Prático

Nota prévia: O enunciado poderá não estar completo em alguns pontos. Os alunos deverão avaliar as opções e discuti-las com um docente da unidade curricular. As decisões tomadas deverão ser sucintamente explicadas no relatório.

Pretende-se que desenvolva, em linguagem *Java*, uma versão do jogo clássico *Pac-Man*, designada *Tiny-PAC*. A leitura deste enunciado deve ser complementada com a explicação detalhada dos pormenores do jogo original disponível em: <https://en.wikipedia.org/wiki/Pac-Man>



No âmbito da unidade curricular de Programação Avançada este jogo será aproveitado para experimentação e aplicação de muitas das funcionalidades oferecidas pela linguagem *Java*, bem como dos *Software Design Patterns* que são estudados nas aulas. A primeira meta do trabalho foca a elaboração dos mecanismos básicos do jogo, não sendo ainda obrigatória a realização de uma interface com utilizador que permita jogar. Pretende-se apenas que essa interface, obrigatoriamente em modo texto, permita testar as várias fases do jogo. A segunda meta exige a realização da interface gráfica com o utilizador, recorrendo a *JavaFX* (sem recurso a *FXML*).

Descrição do jogo *Tiny-PAC*

O jogo deverá possuir um ecrã inicial no qual deverão ser disponibilizadas ao utilizador as opções para “Iniciar Jogo”, “Consultar Top 5” e “Sair”. Para além das informações que são necessárias apresentar neste ecrã inicial, deve surgir obrigatoriamente uma menção ao DEIS-ISEC-IPC (na segunda meta deverá ser incluído o logotipo do ISEC), ao curso (LEI, LEI-PL ou LEI-CE), à unidade curricular, ao ano letivo e ao nome e número do aluno, bem como a indicação de que é um trabalho académico. Caso o utilizador escolha a opção sair, deverá surgir no ecrã uma janela de diálogo para confirmar se pretende mesmo sair.

Assumindo uma leitura prévia da descrição do jogo original no *site* referido atrás, descreve-se de seguida o funcionamento geral da versão pretendida.

A ação consiste em controlar o personagem *pac-man*, fazendo-o navegar num labirinto, onde vai recolhendo (“comendo”) itens e evitar fantasmas que são controlados pelo computador. Tal como no jogo original existem 4 fantasmas – *Blinky*, *Pinky*, *Inky* e *Clyde* – embora apresentem comportamentos diferentes dos do jogo original, os quais serão descritos mais à frente. No jogo original o labirinto é sempre igual, sendo a evolução entre níveis de dificuldade crescente feita com base na velocidade e tempos usados para os vários momentos do jogo. Na versão a implementar, poderão existir diversos labirintos

como forma diferenciadora de níveis. O labirinto correspondente a cada nível poderá ser definido através de um ficheiro de texto com nome “Level nn .txt” e, no máximo, existirão 20 níveis (“Level01.txt” – “Level20.txt”). Caso não esteja especificado um labirinto para um determinado nível, será assumido o labirinto do nível inferior que esteja definido (é obrigatório que o primeiro nível esteja definido através do ficheiro “Level01.txt”). Para cada nível, a velocidade dos fantasmas deve ir aumentando e o tempo de vulnerabilidade dos fantasmas deverá ir diminuindo (com valores à escolha do aluno).

No ficheiro de definição de um labirinto (exemplo no anexo A), existirão tantas linhas quanto a altura do tabuleiro onde está definido o labirinto e, em cada linha, tantos caracteres quanto o número de colunas do mesmo. Cada carácter corresponde a uma célula do labirinto, em que cada letra permitida tem o seguinte significado:

- x – Parede;
- W – Zona *Warp*. Devem existir duas destas células por labirinto. Quando o *pac-man* entra numa destas células, deverá aparecer de imediato na outra. Este efeito só acontece para o *pac-man*: os fantasmas tratam a zona *Warp* como uma parede;
- o – Local com uma bola (item para ser recolhido). A sua recolha, que consiste em passar por cima (sendo removida do labirinto), faz ganhar 1 ponto. Os fantasmas ignoram as bolas comestíveis, passando por cima destas sem as fazer desaparecer;
- F – Local onde deverá aparecer uma fruta a cada 20 bolas que são comidas. Apenas existe 1 ocorrência desta célula no ficheiro de definição de um labirinto. Durante o jogo, a fruta aparece apenas se nesse instante não estiver disponível outra fruta. Quando recolhida, uma fruta dá pontuação em múltiplos crescentes de 25. Em cada nível começa com 25, 50, 75 e assim sucessivamente. Os fantasmas ignoram as frutas;
- M – Local onde o *pac-man* aparece no início do nível (1 ocorrência no ficheiro);
- O – Local com uma *bola comestível com poderes*. Estas bolas são normalmente representadas através de um símbolo maior do que as bolas sem poderes. Quando recolhidas, estas bolas dão 10 pontos ao jogador e tornam os fantasmas vulneráveis, podendo ser comidos. Quando o *pac-man* come esta bola, os fantasmas devem mudar de aparência (ficando todos iguais) e devem movimentar-se para trás seguindo o percurso realizado até ao momento, mas no sentido contrário (percurso realizado desde que saíram da zona inicial onde aparecem, marcada no ficheiro com Y). Ao chegar à célula inicial, cada fantasma volta à aparência e comportamento normais. Independentemente de os fantasmas atingirem a sua posição inicial, este efeito tem uma duração limitada definida pelo aluno. Existem 4 ocorrências deste tipo de bola por labirinto;
- Y – Portal pelo qual surgem os fantasmas no labirinto. Só deverá existir uma ocorrência deste tipo no tabuleiro;
- y – Locais que definem a caverna onde inicialmente estão os fantasmas, antes de saírem pelo portal. Esta zona é encarada como parede pelo *pac-man*. Deve garantir que todas as células ‘y’ aparecem numa única zona contígua e que a célula ‘Y’ aparece numa célula adjacente a uma célula ‘y’.

Nota: os caracteres indicados acima são representações para efeito de configuração inicial do labirinto através dos **ficheiros** de texto já referidos. Estes caracteres não constituem por si só qualquer indicação acerca da forma como esta informação é representada em **memória**, sendo este um assunto que deve ser cuidadosamente ponderado.

De acordo com o jogo original, existem 4 fantasmas que, no contexto desta versão do jogo, deverão ter os seguintes comportamentos:

- Blinky* – Este fantasma desloca-se sempre em frente e quando chega a uma parede ou cruzamento deve sortear uma das direções possíveis, voltando para trás apenas se não existirem outras direções disponíveis;
- Pinky* – Este fantasma desloca-se na direção de um dos cantos do labirinto de acordo com a seguinte ordem: canto superior direito, canto inferior direito, canto superior esquerdo, canto inferior esquerdo, retomando novamente a sequência. Ao chegar a um cruzamento, sorteia uma direção que o faça, no imediato, aproximar-se mais do canto pretendido do labirinto. Quando encontra um obstáculo, tendo de sortear uma nova direção, se já está a menos de uma determinada distância do canto pretendido, então altera o objetivo para o próximo canto do labirinto. A distância referida deve ser definida pelo aluno - por exemplo 10-15% da largura do tabuleiro;
- Inky* – Este fantasma desloca-se na direção de um dos cantos do tabuleiro, tal como o fantasma *Pinky* e segundo uma lógica igual, mas seguindo a ordem: canto inferior direito, canto inferior esquerdo, canto superior direito, canto superior esquerdo;
- Clyde* – Inicialmente, este fantasma desloca-se como o fantasma *Blinky*. No entanto, sempre que o *pac-man* se encontra na sua linha de visão, ou seja, no mesmo corredor (horizontal ou vertical) que ele próprio, define como objetivo seguir a direção que o leva de encontro ao *pac-man*. Quando está a perseguir o *pac-man* e chega a uma parede ou cruzamento, deve escolher a direção que lhe permita continuar a perseguição. Nesse momento de escolha da nova direção, caso deixe de “ver” o *pac-man* volta ao funcionamento normal (similar ao *Blinky*).

O *pac-man* deverá ser controlado através de teclas de mudança de direção. O movimento será automático ao longo da direção correspondente à tecla, não sendo necessário manter a tecla premida, e continuará nessa direção até encontrar uma parede, onde deverá parar. A pressão repetida da tecla de direção correspondente à direção seguida pelo *pac-man* não deverá fazer com que este ande mais depressa. Quando passa por uma bola comestível normal, bola comestível com poderes ou fruta, a mesma deverá desaparecer e o jogador acumulará os pontos referidos anteriormente. Os fantasmas podem ser comidos quando estão vulneráveis (após o *pac-man* comer a bola com poderes), recebendo o jogador 50 pontos pelo primeiro, 100 pontos pelo segundo, 150 pelo terceiro e 200 pelo quarto. Para cada bola com poderes recolhida, esta sequência de atribuição de pontos volta a ser reiniciada em 50 pontos.

O nível termina quando todas as bolas do labirinto são comidas (incluindo as bolas com poderes), passando o jogo para o nível e labirinto seguinte. Caso o *pac-man* seja apanhado por um fantasma

(encontram-se na mesma posição ao mesmo tempo) o jogador perde uma vida, reiniciando-se o nível (o labirinto é reiniciado). O jogo termina ao chegar ao vigésimo nível ou quando o jogador *pac-man* perde todas as vidas (sugestão: 3 vidas, mas pode ser definido outro valor pelo aluno), sendo a pontuação incluída no *Top 5* caso seja superior a uma outra aí existente. Caso alcance o *Top 5*, o jogador será inquirido sobre o nome a registar. O *Top 5* não deverá ser perdido quando o jogo é encerrado, devendo ser guardado em ficheiro através do processo de “serialização” para permitir a sua reposição numa execução posterior.

Desenvolvimento

O código fonte da aplicação pretendida deve apresentar uma divisão clara entre o modelo e a interface com o utilizador. O código do modelo não deve conter qualquer tipo de interação com o utilizador e a interface com o utilizador não deve incluir qualquer tipo de lógica do jogo. O código da interface com o utilizador poderá incluir lógica de pré-tratamento das ações do utilizador (por exemplo, filtrar ou pré-processar os *inputs* do utilizador, mas o modelo deve ser robusto de modo a atuar nas ordens e receção de dados incorretos ou fora de contexto). A estrutura do projeto, em termos de *packages*, deve refletir esta separação, conforme explicado mais à frente.

A lógica do jogo deve ser desenvolvida seguindo o padrão *State (FSM – Finite-State Machine)*, segundo uma abordagem polimórfica e organização semelhante à apresentada durante as aulas. No contexto da aplicação deverá existir obrigatoriamente uma máquina de estados para controlar a evolução de jogo. Na situação inicial, o labirinto e respetivo conteúdo são apresentados no ecrã, não existindo qualquer ação nem movimento. Ao ser pressionada uma tecla de direção, a ação tem início: o *pac-man* movimenta-se segundo a direção escolhida. Após 5 segundos (o aluno pode definir outro valor) os fantasmas entram no labirinto e começam a deslocar-se. O jogo desenrola-se a partir daí até o *pac-man* ser apanhado por um fantasma ou uma bola com poderes ser comida. Neste caso, o jogo entrará num novo estado, de tempo limitado (definido pelo aluno), correspondente à fase em que os fantasmas ficam vulneráveis. Este estado termina quando se esgota o tempo especificado ou quando todos os fantasmas forem comidos e reverterem ao comportamento normal. A totalidade dos estados e transições entre os mesmos deverão ser identificadas pelo aluno de acordo com o funcionamento definido para o jogo.

Notar que para a gestão da evolução do estado em que os fantasmas se encontram pode também recorrer a uma máquina de estados independente da referida acima, mas esta linha de desenvolvimento é opcional.

Como é evidente, para que o jogo possa decorrer, será necessário incluir um “motor de jogo” que permita realizar a evolução dos elementos do jogo com base num temporizador. Para esse efeito, deve usar o motor simplificado fornecido no Anexo B. Notar que o temporizador atuará sobre a máquina de estados, de modo a fazer evoluir todos os elementos.

Durante o jogo o mesmo poderá ser colocado em pausa. Quando estiver em pausa deverá ser possível sair do jogo ou salvar o jogo para o poder restaurar numa execução futura da aplicação. Quando for escolhida a opção do menu principal para iniciar um jogo, deve ser detetado se existe um jogo gravado

e deve perguntar se pretende continuar o jogo anterior. Caso o jogador responda que não, essa possibilidade não deve ser facultada novamente para esse jogo guardado.

A máquina de estados apenas se aplica ao momento de jogo, não devendo ser usada para gerir as opções no ecrã inicial da aplicação ou, por exemplo, a consulta do *Top 5*.

Para representação em memória do tabuleiro que suporta o labirinto deverá ser usada a `class Maze` do Anexo C. Os elementos geridos pela classe devem implementar a `interface IMazeItem` também definida no Anexo C. As classes e interfaces referidas não podem ser alteradas. Caso venha a ser identificado algum problema no enunciado relativamente a esta estrutura, serão fornecidas novas versões que permitam resolver esses problemas.

Não é admitido o recurso a bibliotecas externas. As implementações devem recorrer exclusivamente à Java API, JavaFX (na Meta 2) ou bibliotecas referidas explicitamente no contexto deste enunciado.

Objetivos e requisitos

Os objetivos das duas metas do trabalho prático são os seguintes:

- **Primeira meta:**
 - Estruturação do projeto em *packages*;
 - Diagrama de estados completo;
 - Classes representativas dos dados e de todos os elementos que integram o jogo;
 - Máquina de estados para gerir a evolução do jogo;
 - Interface básica para testes dos desenvolvimentos
 - Nesta meta, caso pretenda, pode recorrer a uma biblioteca que faça a implementação do sistema *Curses* para facilitar o posicionamento do cursor no ecrã, por exemplo, a biblioteca *Lanterna*;
- **Segunda meta** (adicionalmente ao implementado na primeira meta):
 - Implementação do jogo com uma interface com o utilizador (IU) em modo gráfico (*JavaFX*)
A atualização desta interface, bem como outras atualizações de informação, deverá estar de acordo com o padrão de notificações assíncronas estudado nas aulas;
 - Implementação do *Top 5*, incluindo a criação, registo de novo resultado, persistência (serialização) e apresentação através de interface gráfica;
 - As interfaces com o utilizador básicas que não sejam significativamente distintas de uma versão em modo texto serão bastante penalizadas. Espera-se que as interfaces com o utilizador sejam apelativas, intuitivas e funcionais;
 - Gravação e restauro do jogo em ficheiro binário através do processo de serialização;
 - Código devidamente comentado usando *JavaDoc*;
 - Incorporação de testes unitários nas classes relativas à máquina de estados.

Estrutura de projeto

A aplicação deve estar organizada em *packages*, incluindo os seguintes (podem existir outros):

- **pt.isec.pa.tinypac** – *package* que abrange toda a aplicação;
- **pt.isec.pa.tinypac.model** – tem a classe que constitui a fachada de acesso à lógica, que internamente distribui as responsabilidades que lhe são pedidas, gerindo a dinâmica de processamento através de uma máquina de estados. Esta classe apenas será obrigatória na segunda meta. Na primeira meta a interface com o utilizador deverá aceder às funcionalidades através da classe *Context* da FSM;
- **pt.isec.pa.tinypac.model.fsm** – contém as classes referentes à máquina de estados e respetiva hierarquia de estados;
- **pt.isec.pa.tinypac.model.data** – contém as classes que representam as estruturas de dados e respetiva gestão. Inclui a classe *Maze* e a interface *IMazeItem*;
- **pt.isec.pa.tinypac.gameengine** – Código fornecido relativo ao “motor de jogo”;
- **pt.isec.pa.tinypac.ui.text** – classe(s) que implementa(m) a interface em modo texto;
- **pt.isec.pa.tinypac.ui.gui** – classes que implementam a interface em modo gráfico em *JavaFx* (apenas na segunda meta).

Regras gerais

O trabalho deve ser realizado individualmente, sendo a elaboração do trabalho dividida em duas metas separadas.

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

- Primeira meta: **2 de maio (8h00)**;
- Segunda meta: **19 de junho (8h00)**.

As entregas correspondentes às duas metas do trabalho devem ser feitas através do *Nónio/InforEstudante* num ficheiro compactado no formato **ZIP** e apenas neste formato. O nome deste ficheiro deve obrigatoriamente incluir o primeiro nome, o último nome e o número de estudante (constante no Nóncio).

Exemplo: Antonio-Ferreira-202006104.zip

Em ambas as metas, o ficheiro **ZIP** deve conter, pelo menos:

- O projeto com todo o código fonte produzido (projeto *IntelliJ* ou *Visual Studio Code*, sem diretórios de *output* da compilação: *bin*, *build*, *out* ou similares);
- Eventuais ficheiros adicionais de dados e recursos auxiliares necessários à execução do programa, caso não estejam incluídos nos diretórios do projeto referidos no ponto anterior;
- O relatório em formato **PDF**.

O relatório deve incluir, em ambas as metas:

- Uma descrição sintética acerca das opções e decisões tomadas na implementação (máximo de uma página);
- O diagrama da máquina de estados que controla o jogo, devidamente explicado; neste diagrama, o nome atribuído às transições de estado deve corresponder ao nome dado às funções da hierarquia de estados a que correspondem e o nome dos estados deve também corresponder ao nome das classes que os representam;
- Diagramas de outros padrões de programação que tenham eventualmente sido aplicados no trabalho;
- Descrição sucinta das classes utilizadas no programa (o que representam e os objetivos);
- Descrição sucinta do relacionamento entre as classes (podem ser usados diagramas UML);
- Para cada funcionalidade esperada da aplicação, a indicação de cumprido/implementado totalmente ou parcialmente (especificar o que foi efetivamente cumprido neste caso) ou não cumprido/implementado (especificar a razão). O uso de uma tabela pode simplificar a elaboração desta parte.

A primeira meta será sujeita a uma apresentação e discussão da linha de desenvolvimento que se encontra a ser seguida, de modo a identificar situações que poderão causar dificuldades na segunda meta. Na segunda meta os trabalhos serão sujeitos a defesa, a qual incluíra a apresentação, explicação e discussão do trabalho apresentado, podendo ainda incluir a realização de alterações ao que foi entregue.

À primeira meta do trabalho corresponde um coeficiente (fator multiplicativo) igual a **0.9** ou **1.0** e à segunda meta **12** valores. A classificação final será o resultado da multiplicação do coeficiente da primeira meta com o resultado obtido na segunda.

Anexo A

Exemplo de um ficheiro de definição de um nível (exemplo de nome: `Level01.txt`)

[illegible]

Anexo B

```
package pt.isec.pa.tinypac.gameengine;

public enum GameState {READY, RUNNING, PAUSED}
```

```
package pt.isec.pa.tinypac.gameengine;

public interface IGameEngineEvolve {
    void evolve(IGameEngine gameEngine, long currentTime);
}
```

```
package pt.isec.pa.tinypac.gameengine;

public interface IGameEngine {
    void registerClient(IGameEngineEvolve listener);
    void unregisterClient(IGameEngineEvolve listener);

    boolean start(long interval); //ms; only works in the READY state
    boolean stop(); // works in the RUNNING or PAUSED states

    boolean pause(); // only works in the RUNNING state
    boolean resume(); // only works in the PAUSED state

    GameState getCurrentState(); // returns the current state

    long getInterval();
    void setInterval(long newInterval); // change the interval

    void waitForTheEnd(); // wait until the engine stops
}
```

```
package pt.isec.pa.tinypac.gameengine;

import java.util.HashSet;
import java.util.Set;

public final class GameEngine implements IGameEngine {
    private GameState state;
    private GameEngineThread controlThread;
    private Set<IGameEngineEvolve> clients;
    System.Logger logger;

    private void setState(GameEngineState state) {
        this.state = state;
        logger.log(System.Logger.Level.INFO, state.toString());
    }

    public GameEngine() {
        logger = System.getLogger("GameEngine");
        clients = new HashSet<>();
        setState(GameEngineState.READY);
    }
}
```

```

@Override
public void registerClient(IGameEngineEvolve listener) {
    clients.add(listener);
}

@Override
public void unregisterClient(IGameEngineEvolve listener) {
    clients.remove(listener);
}

@Override
public boolean start(long interval) {
    if (state != GameEngineState.READY)
        return false;
    controlThread = new GameEngineThread(interval);
    setState(GameEngineState.RUNNING);
    controlThread.start();
    return false;
}

@Override
public boolean stop() {
    if (state == GameEngineState.READY)
        return false;
    setState(GameEngineState.READY);
    return true;
}

@Override
public boolean pause() {
    if (state != GameEngineState.RUNNING)
        return false;
    setState(GameEngineState.PAUSED);
    return true;
}

@Override
public boolean resume() {
    if (state != GameEngineState.PAUSED)
        return false;
    setState(GameEngineState.RUNNING);
    return false;
}

@Override
public GameEngineState getCurrentState() {
    return state;
}

@Override
public long getInterval() {
    return controlThread.interval;
}

@Override
public void setInterval(long newInterval) {
    if (controlThread != null)
        controlThread.interval = newInterval;
}

```

```

@Override
public void waitForTheEnd() {
    try {
        controlThread.join();
    } catch (InterruptedException e) {}
}

private class GameEngineThread extends Thread {
    long interval;

    GameEngineThread(long interval) {
        this.interval = interval;
        this.setDaemon(true);
    }

    @Override
    public void run() {
        int errCounter = 0;
        while (true) {
            if (state == GameEngineState.READY) break;
            if (state == GameEngineState.RUNNING) {
                new Thread(() -> {
                    long time = System.nanoTime();
                    clients.forEach(
                        client -> client.evolve(GameEngine.this, time)
                    );
                }).start();
            }
            try {
                //noinspection BusyWait
                sleep(interval);
                errCounter = 0;
            } catch (InterruptedException e) {
                if (state == GameEngineState.READY || errCounter++ > 10)
                    break;
            }
        }
    }
}
}

```

Exemplo de utilização:

```

class TestClient implements IGameEngineEvolve {
    int count = 0;
    @Override
    public void evolve(IGameEngine gameEngine, long currentTime) {
        System.out.printf("[%d] %d\n", currentTime, ++count);
        if (count >= 20) gameEngine.stop();
    }
}

public class TinyPacMain {
    public static void main(String[] args) {
        IGameEngine gameEngine = new GameEngine();
        TestClient client = new TestClient();
        gameEngine.registerClient(client);
        gameEngine.start(500);
        gameEngine.waitForTheEnd();
    }
}

```

Anexo C

```
package pt.isec.pa.tinypac.model.data;

public interface IMazeElement {
    char getSymbol(); // returns the symbol of this element
                      // The list of symbols is available
                      // in the description of this work
}

package pt.isec.pa.tinypac.model.data;

public final class Maze {
    private final IMazeElement[][] board;

    public Maze(int height, int width) {
        board = new IMazeElement[height][width];
    }

    public boolean set(int y, int x, IMazeElement element) {
        if (y < 0 || y >= board.length || x < 0 || x >= board[0].length)
            return false;
        board[y][x] = element; // can be null
        return true;
    }

    public IMazeElement get(int y, int x) {
        if (y < 0 || y >= board.length || x < 0 || x >= board[0].length)
            return null;
        return board[y][x]; // can be null
    }

    public char[][] getMaze() {
        char[][] char_board = new char[board.length][board[0].length];
        for(int y=0; y<board.length; y++)
            for(int x=0; x<board[y].length; x++)
                if (board[y][x]==null)
                    char_board[y][x] = ' ';
                else
                    char_board[y][x] = board[y][x].getSymbol();
        return char_board;
    }
}
```