Instituto Superior de Engenharia
Politécnico de Coimbra

Programação Avançada

2023/2024

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 desenvolver, em linguagem *Java*, uma aplicação para simular a evolução de um ecossistema ao longo do tempo: *JavaLife*.

A área que define o ecossistema é caracterizada por um comprimento e uma largura, cujos valores são representados em unidades genéricas. A visualização do ecossistema na interface com o utilizador (*UI*) deverá ser realizada em duas dimensões (o comprimento corresponderá à altura no espaço disponível no ecrã) e ocupará na totalidade a área que lhe for alocada, aplicando factores de escala se necessário. Por exemplo, se a área do ecossistema for definida com uma largura de 500 (unidades genéricas) e lhe for alocado um espaço no ecrã com 1000 pixéis de largura, então cada unidade de largura do ecossistema corresponderá a 2 pixéis.

O ecossistema evolui numa escala de unidades temporais, que poderão ser configuradas para alterar o comportamento da simulação. Por exemplo, pode ser definida a unidade temporal com o valor 100ms para se ter uma evolução mais rápida ou um valor de 2000ms (2s) para uma simulação mais lenta.

No ecossistema poderão existir vários elementos dispostos aquando da definição da simulação, sendo estes elementos dos seguintes de 3 tipos:

- *Inanimados*
  - Estes elementos são colocados na área de simulação, ocupando de forma fixa um determinado local (não possuindo capacidade de movimento nem crescimento) e impedindo qualquer outro elemento de ocupar a zona em causa;
  - Para além de poderem existir elementos deste tipo em qualquer parte da área de simulação, também deverão existir a toda a sua volta formando uma cerca que impede os elementos animados de saírem do ecossistema (podem ser definidos com as dimensões mínimas para exercerem esta funcionalidade);
  - Estes elementos podem existir em diversos tamanhos, mas sempre com uma forma retangular.
- *Flora*
  - Estes elementos não possuem movimento, mas possuem um nível de força que poderá desencadear a sua reprodução;

- A força destes elementos varia entre 0 e 100, nunca ultrapassando este valor. Se um elemento fica sem força (correspondente a um nível de força igual a zero), o elemento desaparece. Por omissão, a força inicial é 50;
- A força aumentará a cada unidade de tempo, sendo o valor de crescimento definido por um valor absoluto/unidade de tempo (ex.: +0.5);
- Quando a sua força atingir 90 poderá reproduzir-se, gerando um novo elemento do mesmo tipo com força inicial igual ao valor por omissão (50) numa célula adjacente que esteja livre, caso exista alguma (se não existir não se reproduz; se em qualquer momento da sua vida voltar a ter um espaço livre adjacente, então reproduzir-se-á). Depois de se reproduzir, a força do progenitor reduzirá para 60. Este tipo de elemento apenas se pode reproduzir duas vezes durante a sua vida;
- Estes elementos podem existir em diversos tamanhos, mas sempre com uma forma retangular, considerando-se adjacente uma área de igual dimensão à sua (em cima, em baixo, à esquerda ou à direita);
- Sempre que estiver sobreposto por um ou mais elementos do tipo fauna, perderá força por unidade de tempo, num valor a definir nas configurações gerais para este tipo de elemento. Por omissão, esse valor é igual a uma (1) unidade de força por elemento fauna. Essa força perdida é acumulada a cada um dos elementos do tipo fauna sobrepostos (por omissão, 1 unidade de força por cada unidade de tempo);
- Este elemento será representado graficamente por uma cor à escolha com um nível de transparência inversamente proporcional à sua força.

- *Fauna*
  - Estes elementos encontram-se sempre em movimento, excepto quando se estão a alimentar (sobrepostos a um elemento flora). O movimento é definido por uma velocidade (distância por unidade de tempo) e uma direção, em graus a variar de 0° a 359° (0° corresponderá a um movimento para a direita, 90° para cima, 180° para a esquerda e 270° para baixo);
  - A sua força varia entre 0 e 100, nunca ultrapassando este valor. Se a sua força reduzir para 0, o elemento desaparece. Por omissão, o valor inicial da força e 50. A força dos elementos do tipo fauna deve ser representada no ecrã, por exemplo, por um valor ou barra junto de cada um dos elementos (pode ser usada outra forma de representação que se considere mais adequada);
  - Quando se movimenta, perde uma parte da sua força (valor a definir nas configurações do elemento; por omissão, 0.5);
  - Quando a sua força está abaixo de 35, entra num estado de procura de comida tendo em consideração os seguintes pontos:
    - Deverá dirigir-se para o elemento do tipo flora mais perto segundo a distância mais curta (caso encontre um elemento no caminho, então deverá sortear uma direção alternativa para o seu movimento, retomando posteriormente o seu objetivo – não necessita alterar o estado de procura de comida em que se encontra);

- Ao atingir um elemento do tipo flora (sobrepondo-se parcial ou totalmente com esta) pára e, por cada unidade de tempo que permanece nessa posição, recebe a força perdida pelo elemento do tipo flora (conforme já referido);
- Permanece em procura de comida/alimentação enquanto a sua força estiver abaixo de 80, tendo em consideração as seguintes situações:
    - Se esgotar a flora existente no local (a flora desapareceu por ficar com força zero) e não lhe tiver permitido atingir 80 de força, o elemento fauna continuará à procura de comida, dirigindo-se para outro elemento do tipo flora;
    - Caso o elemento flora tenha força suficiente, o elemento fauna poderá continuar a alimentar-se, abandonando o local quando atingir o nível de força 100;
- Caso não exista qualquer elemento do tipo flora e o elemento fauna ainda não tiver atingido o nível de força 80, dirige-se para o elemento de fauna com menos força (focando-se nesse elemento mesmo que, entretanto, o mais fraco passe a ser outro elemento). Ao chegar a uma posição adjacente, ataca-o e mata-o (o outro elemento desaparece), com um custo de 10 no seu nível de força;
    - Caso permaneça vivo (ou seja, tinha força superior a 10 antes de atacar) então somará à sua força a força do elemento que matou;
    - Caso contrário (caso morra), a força que tinha no início do ataque é acumulada ao elemento que atacou e que acabou por não conseguir matar;
    - Se os dois elementos estiverem em modo de procura de comida então morrerá o mais fraco;
    - Considera-se que uma posição adjacente para ataque é aquela em que efetuando um novo movimento, no sentido em que se estava movimentar, iria originar a sobreposição, pelo menos parcial;

o Quando a sua força está acima de 50 e não está em procura de comida, dirigir-se-á no sentido do elemento fauna com mais força (notar que o elemento mais forte poderá mudar e, nesse caso, o sentido seguido também mudará);

o Se ele se mantiver 10 unidades de tempo a uma distância inferior a 5 de um outro elemento fauna (pode não ser o mesmo durante este período) então reproduzir-se-á, gerando outro elemento com força 50 na zona livre ou com flora mais perto do local onde se encontra (se não existir zona livre não se reproduzirá, podendo-se reproduzir quando essa zona livre existir). A reprodução tem um custo de 25 na força do elemento que se reproduziu;
    - Se os elementos A e B estiveram "perto um do outro" durante 10 unidades de tempo, ambos se poderão reproduzir, ou seja, surgem dois novos elementos, A' e B', e A e B perdem 25 de força cada um;

o Este tipo de elemento deverá ser representado no ecrã através de uma imagem a carregar entre um conjunto de possibilidades e com dimensão definida na sua criação.

Num ecossistema mais completo poderão existir diversos subtipos de elementos inanimados (cerca, pedras, ...), elementos flora (erva, árvores, ...) e elementos fauna (ovelhas, lobos, ...), mas por questões de simplificação deste trabalho, assuma que só existirá um subtipo de cada um dos tipos principais indicados acima: *Pedra*, *Erva* e *Animal*, respetivamente.

Interface gráfica da aplicação

A interface da aplicação deverá ser desenvolvida em *JavaFX*. Neste contexto e de acordo com o formato típico de aplicação usados nas aulas, o objeto do tipo *Application* deverá receber ou obter a referência para a classe *Facade* do modelo de dados, devendo-a propagar para todos os elementos que dela necessitem. (NOTA: Não pode usar o padrão *singleton* para aceder ao modelo de dados a partir dos objetos que constituem a interface *JavaFX*).

A aplicação de simulação deverá possuir um *menu* com as seguintes opções, as quais poderão necessitar de implementações adequadas no contexto do modelo de dados:

- Ficheiro
    - Criar – permite criar uma nova simulação, entrando no modo de configuração;
    - Abrir – permite abrir e continuar uma simulação previamente gravada;
    - Gravar – permite gravar o estado atual da simulação, indicando o nome do ficheiro;
    - Exportar – exportar para ficheiro de texto (formato *csv*) os atuais elementos da simulação, incluindo informação sobre o tipo, força e posição *x* e *y*;
    - Importar – importar os elementos guardados num ficheiro de texto (formato *csv*). Os elementos importados que se sobrepõem (posição) aos existentes na simulação em curso deverão ser ignorados, assim como aqueles cuja posição fique fora da área da simulação;
    - Sair – sair da simulação, perguntando se pretende gravar caso não o tenha feito anteriormente.
- Ecossistema
    - Configurações gerais do ecossistema – definição de valores alternativos para os valores por omissão referidos no enunciado, bem como a dimensão do ecossistema. A dimensão do ecossistema só pode ser alterada antes da simulação ser iniciada;
    - Adicionar elemento inanimado;
    - Adicionar elemento flora;
    - Adicionar elemento fauna;
    - Editar elemento – permite alterar os parâmetros do elemento selecionado;
    - Eliminar elemento – ter em atenção que os elementos correspondentes à cerca não poderão ser eliminados (a cerca deve ser criada automaticamente quando uma nova simulação é criada e, por isso, não devem ser eliminados);
    - Undo/Redo (de todas as operações no contexto da configuração do ecossistema).
- Simulação
    - Configuração da simulação – por exemplo, para definir a unidade de tempo e as dimensões da área visual de simulação;
    - Executar/Parar;

    - Pausar/Continuar;
    - Gravar *snapshot* – permite registar o momento atual da simulação (esta informação nunca é gravada para ficheiro, nem quando se ordena a gravação do estado atual da simulação para ficheiro);
    - Restaurar *snapshot* – permite retomar a simulação num ponto salvaguardado anteriormente.
- Eventos
    - Aplicar Sol – durante 10 unidades de tempo a flora ganha força ao dobro da velocidade e a fauna desloca-se a metade da velocidade;
    - Aplicar herbicida – aplica-se apenas a elementos do tipo flora, matando o elemento selecionado;
    - Injetar força – aplica-se apenas a elementos do tipo fauna, permitindo aumentar a força do elemento selecionado em 50 unidades.

As opções de configuração apenas poderão estar ativas quando existir uma simulação criada ou lida de ficheiro e não se encontrar em execução. Dessa forma, deverá ser possível pausar a simulação para a poder reconfigurar de acordo com as preferências do utilizador. De cada vez que se adiciona um elemento, essa ação ativará automaticamente o modo de edição para esse elemento, para que os seus parâmetros possam ser ajustados.

Notas sobre o desenvolvimento

No modelo de dados, deverá definir uma hierarquia de classes *Java*, com recurso a generalização de elementos através de interfaces *Java* (*interface X{…}*), adequada à representação do ecossistema e de todos os elementos que nele existam.

O ecossistema deverá ser representado pela classe *Ecossistema*, na qual deverá existir um único conjunto, *Set<IElemento>*, que agregue todos os elementos que o constituem (poderão existir outros campos para gerir as configurações do ecossistema). Cada elemento apenas estará armazenado no conjunto referido, não devendo ser permitidas réplicas. Na classe *Ecossistema* poderão existir outros campos essenciais para o seu funcionamento, mas os elementos deverão ser apenas armazenados através do *Set* referido.

Um elemento deverá ter um identificador único por tipo que o identifica (*id*, do tipo *int*). Desta forma, por exemplo, poderá existir um elemento do tipo fauna com *id* 10 e um elemento do tipo flora com *id* 10, mas não poderão existir dois elementos com *id* 10 do tipo fauna.

Assuma as seguintes definições, sugestivas do tipo de organização pretendido. Devem ser mantidos os membros e os modificadores já existentes (por exemplo, devem ser mantidos os modificadores *sealed* e

o conjunto de classes derivadas permitidas), mas os seus membros podem ser completados para incorporação de mais funcionalidades:

```
public enum Elemento {
    INANIMADO, FLORA, FAUNA
}

public sealed interface IElemento
        extends Serializable
        permits ElementoBase {
    int getId(); // retorna o identificador
    Elemento getType(); // retorna o tipo
    Area getArea(); // retorna a área ocupada
}

public record Area(double cima, double esquerda,
                   double baixo, double direita) {}
//Alternativa:
//   public record Area(double xi, double yi, double xf, double yf) {}

public sealed interface IElementoComForca
        permits Fauna, Flora {
    double getForca();
    void setForca(double forca);
}

public sealed interface IElementoComImagem
        permits Fauna, Flora {
    String getImagem(); // path da imagem
    void setImagem(String imagem);
}

public abstract sealed class ElementoBase
        implements IElemento
        permits Inanimado, Flora, Fauna {
    //TODO
}

public class Ecossistema
        implements Serializable, IGameEngineEvolve{
    private Set<IElemento> elementos;
    //TODO
}
```

Na realização deste trabalho deverão ser aplicados os *Software Design Patterns* e técnicas de acordo com as seguintes indicações:

- *Serialização*:
    - Gravação de simulação em ficheiro;
    - Leitura de simulação gravada em ficheiro.
- *Command*:
    - Configuração dos parâmetros do ecossistema;
    - Adição, edição e remoção de elementos à simulação;
    - Operações de Undo e Redo das operações anteriores.
- *FSM*:
    - Gestão dos estados dos elementos do tipo fauna durante a simulação;
        - Notar que cada elemento pode estar num estado diferente.
- *Factory*:
    - Para criação dos elementos dos diversos tipos;
    - Criação de estados no contexto da FSM.
- *Memento*:
    - Criação de *snapshots* da simulação;
    - Restauro de momentos da simulação anteriormente gravados;
    - Sugestão: usar a *serialização* para memória.
- *Facade* com funcionalidades de observação:
    - Disponibilização e controlo de acesso a todas as funcionalidades do modelo de dados;
        - Deve garantir que não são retornadas referências para objetos que permitam posteriormente a sua alteração fora do seu controlo;
    - Sinalização aos clientes do modelo de dados (p.ex: *UI*) da existência de qualquer atualização do modelo.
- *Multiton*:
    - Para gestão das imagens usadas para cada elemento do tipo fauna.

A forma de implementação dos padrões é muito diversa, pelo que se deve restringir às formas de implementação usadas no contexto das aulas. Outras formas de implementação não serão avaliadas.

O código fonte da aplicação deve apresentar uma divisão clara entre o modelo e a *UI*. 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 da simulação. 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 atualização da interface com o utilizador deverá estar de acordo com o padrão de notificações assíncronas estudado nas aulas. 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.

Como é evidente, para que a simulação possa decorrer, será necessário incluir um "gerador de *ticks* de relógio" que permita realizar a evolução dos elementos da simulação com base num temporizador. Para esse efeito, deve usar o motor simplificado fornecido no Anexo. O temporizador deverá fazer parte do modelo de dados sendo gerido diretamente pela classe que implementa a funcionalidade de *Facade*.

A classe *Facade* deverá possuir comentários compatíveis com a geração de documentação usando *JavaDoc*. A documentação gerada não necessita ser entregue com o trabalho, mas deverão estar aptos a gerá-la durante a defesa do trabalho.

Embora seja aconselhado qualquer projeto possuir teste unitários (*JUnit*), devido ao peso deste trabalho na avaliação da unidade curricular, esta metodologia essencial à aferição da qualidade de software não é obrigatória. A realização e entrega de testes unitários a pelo menos um dos padrões *FSM* ou *Command* será bonificada até +5% (dentro dos limites da cotação atribuída ao trabalho).

No desenvolvimento desta aplicação não poderá recorrer a bibliotecas externas não referidas explicitamente no contexto deste enunciado ou das aulas.

Estrutura de projeto

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

- **`pt.isec.pa.javalife`** – *package* que abrange toda a aplicação;
- **`pt.isec.pa.javalive.model`** – inclui a classe que constitui a fachada de acesso à lógica, que internamente distribui as responsabilidades que lhe são pedidas;
- **`pt.isec.pa.javalive.model.data`** – contém as classes que representam as estruturas de dados e respetiva gestão;
- **`pt.isec.pa.javalive.model.gameengine`** – Código fornecido relativo ao "gerador de *ticks* de relógio";
- **`pt.isec.pa.javalive.ui.gui`** – classes que implementam a interface em modo gráfico em *JavaFx*.

Exemplo de outros possíveis *packages* (são apenas exemplificativos e poderão não se adequar à forma como o código foi organizado):

- **`pt.isec.pa.javalife.model.command`** – classes de suporte das funcionalidades implementadas com auxílio do padrão *Command*;
- **`pt.isec.pa.javalife.model.memento`** – classes de suporte das funcionalidades implementadas com auxílio do padrão *Memento*;
- **`pt.isec.pa.javalive.ui.gui.res`** – package base para gestão dos recursos multimédia da aplicação, recorrendo, por exemplo, ao padrão *multiton*.

Regras gerais

Forma de realização: em grupos de 3 elementos.

Data de entrega do trabalho: **31 de maio (23h59)**.

Material a entregar:

- As entregas 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 os números de estudante dos elementos do grupo.

  Exemplo: `JavaLive_201906104_202006401_202106014.zip`

- O ficheiro *Zip* deve conter:
  - O projeto com todo o código fonte produzido (projeto *IntelliJ*, 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:

- Uma descrição sintética acerca das opções e decisões tomadas na implementação (máximo de uma página);
- O diagrama da máquina de estados que controla os elementos do tipo fauna, 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 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.

Anexo

```
package pt.isec.pa.javalife.model.gameengine;

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

```
package pt.isec.pa.javalife.model.gameengine;

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

```
package pt.isec.pa.javalife.model.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

    GameEngineState 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.javalife.model.gameengine;

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

public final class GameEngine implements IGameEngine {
    private GameEngineState state;
    private GameEngineThread controlThread;
    private final 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 true;
}

@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 true;
}

@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 ignored) {}
}
```

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