Práctica 5

Genericidad, Colecciones, Lambdas y Patrones de Diseño

Inicio: Semana del 11de abril.
Duración: 3 semanas.
Entrega: Viernes 29 de abril, 23:55 CEST (todos los grupos)
Peso de la práctica: 30%

El objetivo de esta práctica es ejercitar conceptos de orientación más avanzados, como son

- *Diseño de clases genéricas, y altamente reutilizables*
- *Uso de la librería de colecciones Java.*
- *Empleo de patrones de diseño y estrategias de diseño de APIs*
- *Uso de expresiones lambda*

Apartado 0. Introducción

En esta práctica vamos a ir desarrollando progresivamente una aplicación para la simulación basada en agentes. Este tipo de simulaciones consisten en una serie de agentes, cada uno con ciertas propiedades y comportamiento, que se mueven e interaccionan en una rejilla bidimensional. Un esquema de los elementos de este tipo de simulación se muestra en la Figura 1.

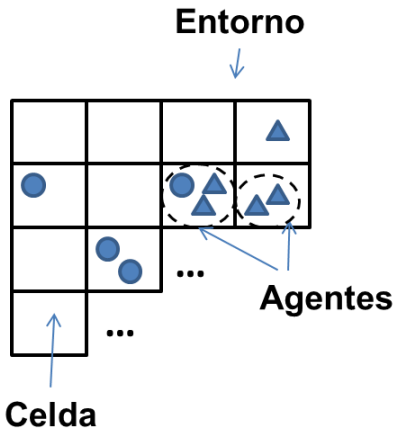

Figura 1: Estructura de una simulación basada en agentes

La simulación basada en agentes es un instrumento muy común para el estudio de fenómenos en muchas disciplinas, como la ecología, economía y sociología, entre muchas otras. Existen muchos lenguajes y herramientas para la simulación basada en agentes, como Swarm, NetLogo o Mason. En esta práctica desarrollarás tu propio entorno básico para realizar este tipo de simulaciones. Para ello, en primer lugar crearemos una estructura genérica para almacenar elementos en una matriz, y posteriormente iremos añadiendo funcionalidad para la simulación basada en agentes.

Apartado 1. ***Genericidad y Colecciones*****: Matrices bidimensionales genéricas (2 puntos)**

En este primer apartado comenzaremos definiendo el tipo abstracto de datos necesario para dar soporte al entorno de simulación. Más concretamente, se implementará una matriz a la que se le añadirán algunas primitivas básicas para su posterior uso en un sistema de simulación basada en agentes.

Para abstraernos de las particularidades de la implementación, cualquier matriz que implemente nuestro sistema deberá implementar la siguiente interfaz:

```
public interface IMatrix<T> {
	int getCols();
	int getRows();
	boolean isLegalPosition(int i, int j);
	void    addElement(IMatrixElement<T> element) throws IllegalPositionException;
	IMatrixElement<T>       getElementAt(int i, int j) throws IllegalPositionException;
	List<IMatrixElement<T>> getNeighboursAt(int i, int j) throws IllegalPositionException;
	List<IMatrixElement<T>> asList();
}
```

Como puede apreciarse, la interfaz `IMatrix` define el conjunto de primitivas que debe tener cualquier clase que implemente una matriz. El índice i se refiere a las filas (que empieza en 0 y llega hasta `getRows()-1`), y el índice j se refiere a las columnas (de 0 a `getCols()-1`). Asimismo, en dicha interfaz puede verse cómo los elementos que podrán almacenarse en la matriz son de tipo genérico.

Para encapsular más aún el comportamiento de los elementos que podrán insertarse en la matriz, se definirá una interfaz que modelará su funcionalidad, tal y como se muestra en el siguiente código.

```
public interface IMatrixElement<T> {
	int     getI();
	int     getJ();
	T       getElement();
	void    setElement(T element);
}
```

En definitiva, las matrices no son más que una estructura lógica cuya implementación puede realizarse de diferentes maneras. Más concretamente, **se deberá realizar una implementación** como una **matriz dispersa**.

La implementación como array bidimensional es la primera aproximación que viene a la mente, donde los elementos de la matriz se almacenarán en un array bidimensional. Sin embargo, esta implementación tiene dos inconvenientes principales que son el enorme consumo de memoria y tiempo de procesamiento necesario si se tiene una matriz de grandes dimensiones con pocos elementos en ella.

La solución a dichos inconvenientes son las *matrices dispersas* (también llamadas huecas o ralas). Deberás elegir una estructura de datos adecuada considerando dos criterios: (i) Eficiencia de almacenamiento: sólo deberán almacenarse los elementos de la matriz que realmente tienen un valor, (ii) Eficiencia en el recorrido de valores no nulos (por ejemplo por filas), y eficiencia en el acceso aleatorio a los índices seleccionados. Dado que solo se reservará memoria cuando sea necesario almacenar algún dato en la matriz, no se desperdiciará memoria para el resto de la matriz.

Realiza un programa de prueba que cree dos matrices de 4x8 (de Strings) y 6x7 (de enteros), y ejercite cada uno de los métodos de la interfaz `IMatrix`, y las imprima por pantalla. Se recomienda que estas pruebas se realicen usando JUnit.

Apartado 2. *Comparadores*: Uso de interfaces *Comparable* y *Comparator* (1,5 puntos)

A fin de poder comparar matrices, en este apartado se pide añadir un método `equals` a la implementación (así como el correspondiente método `hashCode`). Dos matrices se considerarán iguales si cada elemento en la posición i, j de una matriz es igual a cada elemento i, j de la otra. Asimismo, deberá implementarse la correspondiente prueba, permitiendo comprobar si dos matrices son iguales o diferentes.

También se desea ordenar los elementos de las matrices empleando determinados criterios. Para ello deberá hacerse uso de comparadores mediante la interfaz `Comparator`. Para permitir ordenar los elementos de la matriz de diferentes maneras, deberá añadirse a la interfaz `IMatrix` un método con la siguiente definición:

```
List<IMatrixElement<T>> asListSortedBy(Comparator<IMatrixElement<T>> c);
```

Donde se devuelve una lista con los elementos de la matriz ordenada mediante el criterio de comparación indicado por parámetro.

De nuevo, para probarlo deberán implementarse dos casos de prueba, uno que permita crear una matriz y listar sus elementos ordenados primero por columnas y luego por filas, y otro que muestre primero los valores pares y después las impares en orden creciente.

Apartado 3. Simulación básica (2 puntos)

En este apartado, utilizarás las matrices creadas en apartados anteriores para modelar los agentes, su entorno y el simulador. El entorno será una matriz en la que se almacenarán celdas. Estas podrán contener un número arbitrario de agentes de distinto tipo. El simulador se encargará de ir aumentando progresivamente el tiempo de la simulación, invocando el comportamiento definido para cada agente en cada paso de simulación e imprimiendo el entorno en cada paso. En este apartado, empezaremos por crear las clases básicas para la simulación, y en apartados posteriores iremos añadiendo funcionalidad a los agentes.

El siguiente fragmento de código muestra la creación de un simulador básico para un entorno de tamaño 10 por 10. El código muestra cómo se crean dos tipos de agentes (llamados `random` y `outer`), y se añaden a la simulación con el método `create`. Dicho método debe crear una *copia* de los agentes y añadirlos a la posición indicada (celdas en las posiciones [5, 5] y [7,7]). Ya que de momento no consideramos el comportamiento de los agentes, en este apartado, el método `run` solamente imprimirá el entorno de la simulación, donde cada celda del entorno imprimirá el número de agentes que contiene.

```
BasicSimulator s = new BasicSimulator(10,10);

BasicAgent random = new BasicAgent("random");
BasicAgent outer  = new BasicAgent("outer");

s.create(random, 10, 5, 5);     // Crea 10 agentes de tipo random en la posición (5,5)
s.create(outer, 10, 7, 7);      // Crea 10 agentes de tipo outer en la posición (7,7)
s.run(2);                       // Ejecutamos la simulación durante 2 pasos
```

Como guía, un agente básico podría implementar la siguiente interfaz:

```
public interface IBasicAgent {
      Cell        cell();
      IBasicAgent copy();
}
```

Donde puedes observar que el agente guarda la celda (clase `Cell`) en la que está ubicado. Además, incluye un método `copy`, que sirve para realizar una copia del agente.

El programa anterior debería imprimir lo siguiente en cada paso de simulación (con valores sucesivos para la variable del tiempo):

```
++++++++++++++++++++++++++++++++++++++++++++
Time = 0
  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
  0|  0|  0|  0|  0| 10|  0|  0|  0|  0|
  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
  0|  0|  0|  0|  0|  0|  0| 10|  0|  0|
  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|
```

En este apartado deberás entregar:

- Las clases e interfaces necesarias para la implementación del entorno, el simulador y los agentes.
- Un programa de prueba con el código anterior.

Apartado 4. Lambdas e Interfaces Funcionales: Agentes con comportamiento (2,5 puntos).

En este apartado, extenderemos los agentes para poder añadirles reglas de comportamiento. Una regla de comportamiento tiene una condición de aplicación (que puede omitirse) y una acción. La acción se ejecuta si la condición se cumple. Si se omite la condición de aplicación, la acción se ejecuta siempre que se invoca. Las condiciones y acciones se darán mediante expresiones lambda.

Un agente tendrá una lista ordenada de comportamientos. La ejecución del comportamiento de un agente llamará sucesivamente a cada regla de comportamiento. Las acciones devolverán un booleano, que indica si se deben seguir ejecutando las siguientes acciones de la lista. En este caso, el agente deberá implementar la siguiente interfaz:

```
public interface IAgent extends IBasicAgent{
	void   moveTo(Cell destination);          // Mover a una celda adyacente
	void   exec();                            // Ejecutar comportamiento del agente
	IAgent addBehaviour(Predicate<IAgent> trigger, Function<IAgent, Boolean> behaviour);
	IAgent addBehaviour(Function<IAgent, Boolean> behaviour);
	IAgent copy();                            // Realiza una copia del agente
}
```

Donde puedes ver que se han añadido métodos para mover un agente (`moveTo`), ejecutar sus comportamientos (`exec`), añadirle comportamiento (`addBehaviour`) y crear una copia (`copy`). Este último método sobrescribe el método análogo de `IBasicAgent` para clonar un `IAgent`. Todos los agentes que son copia de uno dado compartirán su comportamiento. Esto quiere decir que si se cambia el comportamiento de la copia original, se cambia en todas las copias. Para ello simplemente no clones la estructura que almacena los comportamientos, y todos los agentes la compartirán.

El siguiente listado muestra como configurar un simulador para agentes con comportamiento, y configurar dichos agentes:

```
Simulator s  = new Simulator(10,10);
Agent random = new Agent("random");
Agent outer  = new Agent("outer");

random.addBehaviour( agent -> { // se ejecuta siempre, movemos el agente a una casilla aleatoria vecina
                        List<Cell> neighbours = agent.cell().neighbours();
                        Cell destination = neighbours.get(new Random().nextInt(neighbours.size()));
                        agent.moveTo(destination);
                        return true;
                     });
outer.addBehaviour( agent -> agent.cell().getElement().size()>5,  // Lo ejecutamos si hay más de 5 agentes
                                                                  // en la celda actual
                    agent -> { // nos movemos a la celda destino con menos agentes
                        List<Cell> neighbours = agent.cell().neighbours();
                        Integer minAgents = neighbours.stream().
                                                mapToInt( c -> c.agents().size() ).
                                                min( ).
                                                getAsInt();
                        List<Cell> destinations = neighbours.stream().
                                                filter( c -> c.agents().size() == minAgents ).
                                                collect(Collectors.toList());
                        Cell destination = destinations.get(new Random().nextInt(destinations.size()));
                        agent.moveTo(destination);
                        return true;
                     });

s.create(random, 100, 5, 5); // Crear 100 agentes random
s.create(outer, 100, 7, 7);  // Crear 100 agentes "outer"

s.run(60);                   // Ejecutar 60 pasos de simulacion
```

En el simulador, deberás tener cuidado de no ejecutar los agentes siempre en el mismo orden, sino en orden aleatorio. Además, el simulador ha de mover los agentes (si se invocó `moveTo` sobre alguno) después de la ejecución de todas las acciones. La clase `Simulator` puede extender `BasicSimulator` (del apartado anterior), haciendo que en cada paso de simulación se ejecuten los comportamientos de todos los agentes y estos se muevan, en caso necesario.

Como resultado de este apartado, en la simulación deberás ver cómo los agentes se van moviendo por el entorno, de acuerdo con el comportamiento indicado. Debes entregar las nuevas clases e interfaces y el programa de prueba.

Apartado 5. Patrones de diseño: Agentes con estado (2 puntos).

En este apartado, vamos a añadir estados a los agentes. Un agente podrá estar en uno de entre varios estados. El agente se mueve de un estado a otro cuando se cumple una condición, que se dará como una expresión lambda. Cada estado podrá tener comportamientos asociados. De esta manera, el comportamiento de un agente viene dado por la ejecución de sus comportamientos globales (como en el apartado anterior), así como de la ejecución de los comportamientos asociados a su estado actual.

En esta práctica, usaremos el *patrón de diseño State* (https://en.wikipedia.org/wiki/State_pattern), que establece que los distintos estados de un objeto se han de modelar con objetos, que deben seguir una interfaz uniforme. Mientras que la implementación tradicional del patrón State utiliza *subclases* de la clase State de manera estática, nosotros utilizaremos *instancias* de State, con comportamientos definidos mediante expresiones lambda.

Para ello, los agentes con estados podrán implementar la siguiente interfaz:

```
public interface IAgentWithState extends IAgent{
  IAgentState     state(String name);
  IAgentWithState addBehaviour(String state, Predicate<IAgent> trigger, Function<IAgent, Boolean> behaviour);
  IAgentWithState addBehaviour(String state, Function<IAgent, Boolean> behaviour);
}
```

Donde `IAgentState` es la interfaz que implementan los estados del agente, que podría ser como sigue:

```
public interface IAgentState {
  String          name();
  void            toState(String target, Predicate<IAgent> trigger);
  IAgentState     changeState();
  IAgentWithState addBehaviour(Predicate<IAgent> trigger, Function<IAgent, Boolean> behaviour);
  IAgentWithState addBehaviour(Function<IAgent, Boolean> behaviour);
  void            exec();
  void            setOwner(IAgentWithState aws);
  IAgentState     copy();
}
```

Como puedes ver, tenemos métodos que devuelven el nombre del estado (`name`), que establecen la condición para ir a otro estado (`toState`), para cambiar el estado y devolver el nuevo estado actual o el mismo si no cambia (`changeState`), para añadir comportamientos a un estado (`addBehaviour`), ejecutar los comportamientos del estado (`exec`), establecer el agente dueño del estado (`setOwner`) y copiar un estado (`copy`). Todas las copias del agente pueden compartir la definición de sus estados, pero cada uno debe tener su propio estado actual.

El siguiente fragmento de código muestra un ejemplo de cómo configurar un agente con estados:

```
AgentWithState outer = new AgentWithState("outer", "idle", "active"); // dos estados: idle y active

outer.state("idle").toState("active", agent -> agent.cell().agents().size()>5);
outer.state("active").toState("idle", agent -> agent.cell().agents().size()<=5);

outer.state("active").addBehaviour(
                             agent -> {
                                    List<Cell> neighbours = agent.cell().neighbours();
                                    Cell destination = neighbours.get(new Random().nextInt(neighbours.size()));
                                    agent.moveTo(destination);
                                    return true;
                             });
```

Deberás entregar las nuevas interfaces y clases, así como un programa de prueba que incluya el código anterior.

Apartado 6. (Opcional, 1 punto): Agentes con propiedades e interacción entre agentes

6.a) **Propiedades (0,5 puntos)**: Los agentes que hemos programado en esta práctica son extremadamente simples. En entornos de simulación más reales, los agentes deberían tener una serie de propiedades, así como métodos para acceder y cambiar dichos valores. En este apartado, se pide extender los agentes con propiedades, de manera que se puedan configurar de la siguiente forma:

```
Agent ant = new Agent("ant").
              set("energy", 50).
              set("food", 0);

Agent nest = new Agent("nest").
              set("food", 0);

Agent food = new Agent("food").
              set("amount", 20);
```

Ten en cuenta que cada copia del tipo de agente debe crear su propia copia de las propiedades.

6.b) **Interacción (0,5 puntos):** Típicamente los agentes no se comportan de manera aislada, sino que interaccionan unos con otros. Para ello, crea una serie de métodos que modelen la interacción entre dos agentes, y modifica el simulador para que los agentes que estén en la misma celda interaccionen los unos con los otros (en orden aleatorio).

El siguiente fragmento de código muestra cómo se podría configurar el comportamiento de interacción de un agente:

```
// La interacción sucede entre un agente tipo ant (hormiga) y un agente comida, siempre que haya
// un valor positivo de comida
ant.addInteraction((self, agent) -> agent.name().equals("food") && agent.get("amount") > 0,
                   (self, agent) -> {
                                    System.out.println("Getting food ("+agent.get("amount")+")");
                                    self.increase("energy", 10);
                                    self.increase("food", 1);
                                    agent.increase("amount", -1);
                                    return true;
                                });
```

En este apartado deberás entregar las nuevas clases e interfaces, así como un programa de prueba que ejercite la nueva funcionalidad.

Normas de Entrega:

- Se deberá entregar
  - un directorio **src** con el código Java de cada apartado, incluidas las pruebas JUnit.
  - un directorio **doc** con la documentación generada
  - un archivo PDF con una breve justificación de las decisiones que se hayan tomado en el desarrollo de la práctica, los problemas principales que se han abordado y cómo se han resuelto.
- Se debe entregar un único fichero ZIP con todo lo solicitado, que deberá llamarse de la siguiente manera: GR<numero_grupo>_<nombre_estudiantes>.zip. Por ejemplo Marisa y Pedro, del grupo 2261, entregarían el fichero: GR2261_MarisaPedro.zip.