Práctica 5

Colecciones, genericidad, expresiones lambda y patrones de diseño

Inicio: Semana del 7 de Abril
Duración: 4 semanas
Entrega: 8 de Mayo, 23:55h (todos los grupos)
Peso de la práctica: 30%

El objetivo de esta práctica es diseñar programas genéricos que usen colecciones avanzadas, sean capaces de adaptarse a tipos paramétricos, y empleen expresiones lambda y patrones de diseño de manera práctica. Para ello, tomando como inspiración librerías actuales para programación *multi-agente* – como LangGraph (https://www.langchain.com/langgraph) o CrewAI (https://www.crewai.com/) – construiremos una librería para la creación y ejecución de flujos de trabajo (*workflows*).

Un flujo de trabajo es un grafo con un nodo inicial, y cero o más nodos finales. Cada nodo representa una actividad y tiene código ejecutable asociado. Los nodos del flujo pueden acceder a un estado común (que representaremos mediante un objeto) y pueden modificarlo como parte de su ejecución. Las relaciones entre nodos establecen el orden de ejecución de los nodos, de tal manera que la ejecución del flujo termina al ejecutar un nodo final, o si el nodo ejecutado no tiene un nodo de salida.

Apartado 1: Creación y ejecución de flujos de trabajo (3 puntos)

Tu diseño de flujos de trabajo se basará en una clase genérica `StateGraph` que representa un flujo de trabajo, y que se parametrizará con el tipo del estado común. Podremos añadir nodos al grafo mediante el método `addNode`, que recibirá como argumentos el nombre del nodo y una expresión lambda con el código que ejecutará el nodo cuando sea activado. `StateGraph` deberá incluir métodos para indicar el nodo inicial, los nodos finales, y para añadir enlaces entre dos nodos (referenciados mediante su nombre). La clase tendrá además un método `run` que recibirá la entrada del flujo (que será un objeto del tipo paramétrico de `StateGraph`, con el estado inicial del flujo) y un *flag* que indique si se quiere una traza de *debug* o no. En caso afirmativo, cada paso de ejecución mostrará el nodo ejecutado y el valor del estado común.

A modo de ejemplo, el siguiente código contiene un flujo que suma dos números y luego calcula su cuadrado. El constructor de `StateGraph` recibe el nombre del flujo y su descripción.

```
public class Main {
  public static void main(String[] args) {
        StateGraph<NumericData> sg = buildWorflow();

        System.out.println(sg);

        NumericData input = new NumericData(2, 3);
        System.out.println("input = " + input);
        NumericData output = sg.run(input, true);   // ejecución con debug
        System.out.println("result = " + output);
  }
  private static StateGraph<NumericData> buildWorflow() {
        StateGraph<NumericData> sg = new StateGraph<>("math2", "Add two numbers, and then square");

        sg.addNode("sum",    (NumericData mo) -> mo.put("result", mo.get("op1")+mo.get("op2")) )
          .addNode("square", (NumericData mo) -> mo.put("result", mo.get("result")*mo.get("result")) );

        sg.addEdge("sum", "square");

        sg.setInitial("sum");
        sg.setFinal("square");

        return sg;
  }
}
```

En el ejemplo, el estado es un objeto de tipo `NumericData`, definido como sigue:

```
public class NumericData extends LinkedHashMap<String, Integer> {
        public NumericData(int op1, int op2) {
                this.put("op1", op1);
                this.put("op2", op2);
                this.put("result", 0);
        }
}
```

Salida esperada:

```
Workflow 'math2' (Add two numbers, and then square):
- Nodes: {sum=Node sum (1 output nodes), square=Node square (0 output nodes)}
- Initial: sum
- Final: square
input = {op1=2, op2=3, result=0}
Step 1 (math2) - input: {op1=2, op2=3, result=0}
Step 2 (math2) - sum executed: {op1=2, op2=3, result=5}
Step 3 (math2) - square executed: {op1=2, op2=3, result=25}
result = {op1=2, op2=3, result=25}
```

Nota: si un nodo N1 tiene varios nodos de salida (p. ej. N2 y N3), entonces N2 y N3 se ejecutarán después de N1, en el orden en el que se conectaron a N1**.**

Se pide: Usando principios de orientación a objetos, crea todo el código necesario para la creación y ejecución de flujos de trabajo. Crea *testers* que prueben la funcionalidad que has implementado. Debes incluir testers que creen flujos de trabajo con estados distintos a la clase `NumericData`.

Apartado 2: Enlaces condicionales (1.5 puntos)

En este apartado, añadirás soporte para incluir enlaces condicionales entre nodos. Un enlace condicional incluye una condición sobre el estado común, que ha de cumplirse para que el nodo destino pueda ejecutarse. Si la condición no se cumple, el nodo destino no es elegible para ser ejecutado, y la ejecución seguiría por otro nodo (si es posible).

Para dar soporte a enlaces condicionales, añade a `StateGraph` un método `addConditionalEdge` que reciba el nombre de los nodos origen y destino, así como una expresión lambda que describa una condición sobre el estado común.

A modo de ejemplo, la siguiente salida corresponde a un flujo de trabajo similar al del apartado 1, pero que sólo realiza el cuadrado si el resultado de la suma es un número par.

Salida esperada:

```
Workflow 'math1' (Add two numbers, and square if even):
- Nodes: {sum=Node sum (1 output nodes), square=Node square (0 output nodes)}
- Initial: sum
- Final: square
input = {op1=2, op2=3, result=0}
Step 1 (math1) - input: {op1=2, op2=3, result=0}
Step 2 (math1) - sum executed: {op1=2, op2=3, result=5}
result = {op1=2, op2=3, result=5}
input = {op1=2, op2=2, result=0}
Step 1 (math1) - input: {op1=2, op2=2, result=0}
Step 2 (math1) - sum executed: {op1=2, op2=2, result=4}
Step 3 (math1) - square executed: {op1=2, op2=2, result=16}
result = {op1=2, op2=2, result=16}
```

Se pide: Usando principios de orientación a objetos, crea todo el código necesario para dar soporte a enlaces condicionales. Crea *testers* que prueben la funcionalidad que has implementado. Debes incluir testers que creen flujos de trabajo con estados distintos a la clase `NumericData`. Los enlaces condicionales permiten la creación de bucles con entrada o salida condicional. Crea un flujo de trabajo que contenga dos nodos, donde el primero produce un entero positivo aleatorio, y el segundo incluye un bucle que lo decrementa hasta llegar a cero.

Apartado 3: Flujos de trabajo anidados (2 puntos)

En este apartado, darás soporte a la llamada de un flujo de trabajo desde otro. En particular, debes añadir a `StateGraph` un método `addWfNode` que permita incluir un flujo de trabajo como si fuera un nodo. Esto permitirá conectar el flujo incluido con otros nodos del grafo padre. Como el flujo incluido puede tener un tipo de estado distinto al del grafo padre, debes añadir métodos para especificar *inyectores* (que traducen del tipo del estado padre al del flujo incluido) y *extractores* (que reflejan la salida del flujo incluido en el estado del flujo padre). El inyector y el extractor podrán darse como expresiones lambda.

A modo de ejemplo, el siguiente código incluye el grafo del apartado anterior en un flujo que replica una palabra tantas veces como la salida del flujo incluido:

```
public static StateGraph<StringData> buildWorflow(StateGraph<NumericData> wfNumeric) {
  StateGraph<StringData> sg = new StateGraph<>("replicate", "Replicates a given word");

  sg.addWfNode("calculate", wfNumeric)
         .withInjector ((StringData sd) -> sd.toNumericData())
         .withExtractor((NumericData nd, StringData sd) -> sd.setTimes(nd.get("result")));

  sg.addNode("replicate", sd -> sd.replicate());
  sg.addEdge("calculate", "replicate")
    .addConditionalEdge("replicate", "replicate", sd -> sd.times()>0);

  sg.setInitial("calculate");

  return sg;
}
```

Salida esperada:

```
Workflow 'replicate' (Replicates a given word):
 - Nodes: {calculate=Node calculate (1 output nodes), replicate=Node replicate (1 output nodes)}
 - Initial: calculate
 - Final: None
input = word: jamon, times: 2, result:
 Step 1 (replicate) - input: word: jamon, times: 2, result:
 Step 1 (math1) - input: {op1=2, op2=0, result=0}
 Step 2 (math1) - sum executed: {op1=2, op2=0, result=2}
 Step 3 (math1) - square executed: {op1=2, op2=0, result=4}
 Step 2 (replicate) - calculate executed: word: jamon, times: 4, result:
 Step 3 (replicate) - replicate executed: word: jamon, times: 3, result: jamon
 Step 4 (replicate) - replicate executed: word: jamon, times: 2, result: jamonjamon
 Step 5 (replicate) - replicate executed: word: jamon, times: 1, result: jamonjamonjamon
 Step 6 (replicate) - replicate executed: word: jamon, times: 0, result: jamonjamonjamonjamon
result = word: jamon, times: 0, result: jamonjamonjamonjamon
```

En este ejemplo, `StringData` define tres atributos (`String word`, `String result`, `int times`), así como un método `toNumericData` (que produce un `NumericData` copiando `times` en `op1`), un método `replicate` (que decrementa times y añade una copia de `word` a `result`), y setters y getters para el atributo `times` (`setTimes` y `times`).

Se pide: Usando principios de orientación a objetos, crea todo el código necesario para dar soporte a flujos anidados. Crea *testers* que prueben la funcionalidad que has implementado. Debes incluir testers que creen flujos de trabajo con estados distintos a la clase `NumericData` y `StringData`.

Apartado 4: Flujos para streaming de datos (1 punto)

La clase `StateGraph` implementada no tiene memoria del estado común entre distintas ejecuciones del flujo. Sin embargo, en aplicaciones de *streaming* de datos, los flujos deben ejecutarse de manera continua ante una llegada de entradas en tiempo real. En este escenario se requiere acceder al histórico de estados de todas las ejecuciones anteriores del flujo.

Para dar soporte a este escenario, crearemos un tipo especial de `StateGraph` llamado `StreamingStateGraph`. Esta clase, también genérica, deberá guardar una lista con todos los estados de ejecuciones anteriores, de tal manera que la entrada de los nodos no será el estado actual, sino la lista de todos los estados de ejecuciones anteriores.

A modo de ejemplo, el siguiente fragmento de código crea un flujo en streaming, y lo ejecuta con 4 entradas de tipo `DoubleData`. El flujo consta de solo un nodo que calcula la media de todas las entradas recibidas hasta ese momento. El método `history` devuelve la lista con todos los estados anteriores. La clase `DoubleData` contiene dos atributos: `value` y `average`.

```
StreamingStateGraph<DoubleData> sg = buildWorflow();  // el método construye el workflow

System.out.println(sg);
List
 .of(1, 5, 2, 4)
 .forEach( d-> { DoubleData wfInput = new DoubleData(d, 0);
                 System.out.println("Workflow input = "+wfInput);
                 sg.run(wfInput, true);
               });

System.out.println("History="+sg.history());
```

Salida esperada:

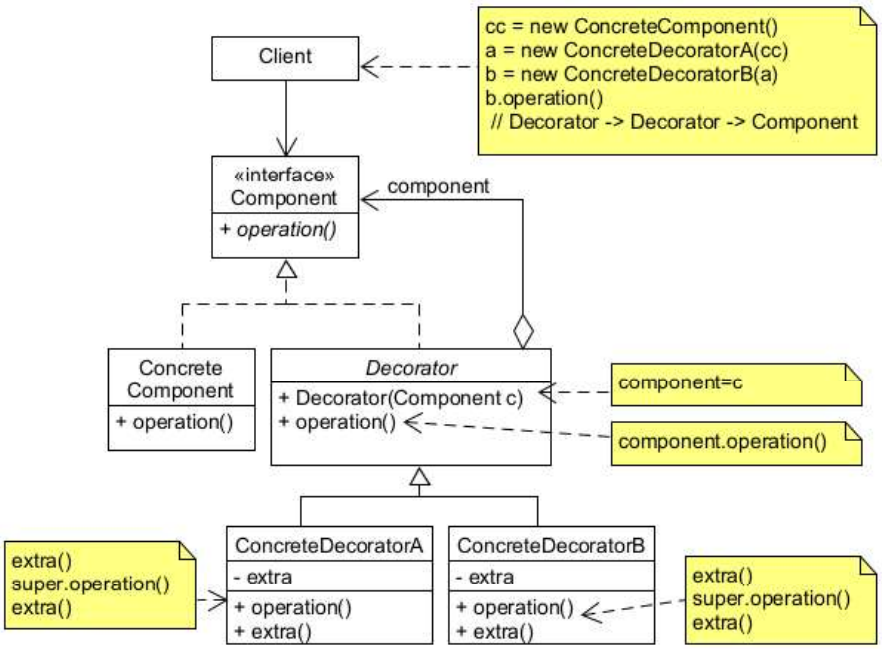
```
Workflow 'average' (Calculates the average of incoming data):
 - Nodes: {average=Node average (0 output nodes)}
 - Initial: average
 - Final: None
Workflow input = 1.0 (avg.= 0.000)
 Step 1 (average) - input: [1.0 (avg.= 0.000)]
 Step 2 (average) - average executed: [1.0 (avg.= 1.000)]
Workflow input = 5.0 (avg.= 0.000)
 Step 1 (average) - input: [1.0 (avg.= 1.000), 5.0 (avg.= 0.000)]
 Step 2 (average) - average executed: [1.0 (avg.= 1.000), 5.0 (avg.= 3.000)]
Workflow input = 2.0 (avg.= 0.000)
 Step 1 (average) - input: [1.0 (avg.= 1.000), 5.0 (avg.= 3.000), 2.0 (avg.= 0.000)]
 Step 2 (average) - average executed: [1.0 (avg.= 1.000), 5.0 (avg.= 3.000), 2.0 (avg.= 2.667)]
Workflow input = 4.0 (avg.= 0.000)
 Step 1 (average) - input: [1.0 (avg.= 1.000), 5.0 (avg.= 3.000), 2.0 (avg.= 2.667), 4.0 (avg.= 0.000)]
 Step 2 (average) - average executed: [1.0 (avg.= 1.000), 5.0 (avg.= 3.000), 2.0 (avg.= 2.667), 4.0 (avg.=
3.000)]
History=[1.0 (avg.= 1.000), 5.0 (avg.= 3.000), 2.0 (avg.= 2.667), 4.0 (avg.= 3.000)]
```

Se pide: Usando principios de orientación a objetos, crea todo el código necesario para dar soporte a flujos en streaming. Crea *testers* que prueben la funcionalidad que has implementado. Debes incluir testers que creen flujos de trabajo con estados distintos a las clases `NumericData`, `StringData` y `DoubleData`.

Apartado 5: Añadiendo funcionalidad a los flujos mediante el patrón *Decorator* (2.5 puntos)

El patrón de diseño *Decorator* permite añadir dinámicamente comportamiento extra a objetos, incluyéndolos dentro de objetos envoltorio (los *decoradores*) que implementan esos comportamientos. De manera dinámica, es posible añadir distintos decoradores a un objeto, que irán añadiéndole los comportamientos deseados.

El diagrama de la derecha muestra el esquema del patrón *Decorator*. Se crea una interfaz (rol `Component`) que modela el comportamiento del objeto real y sus decoradores. De esta manera, tanto el objeto real (`ConcreteComponent`) como sus decoradores (`ConcreteDecorators`) implementan la interfaz. Es frecuente que los decoradores tengan una clase base abstracta común (`Decorator`) con una referencia al objeto que decoran (`component`). Como `component` es de tipo `Component`, este puede ser el objeto real u otro decorador. Al llamar a un método de la interfaz `Component`, se irá ejecutando el código de cada decorador hasta llegar al método del objeto real. El código de los decoradores se podrá ejecutar justo antes y/o después del código del objeto real.



En este apartado usaremos este patrón de diseño para añadir dos funcionalidades a la clase `StateGraph`: un *profiler* (que guarda el tiempo de ejecución de cada nodo), y un *logger* (que guarda información de la ejecución en un fichero). En particular, el profiler debe guardar, por cada nodo ejecutado, su entrada, y el tiempo que tardó en ejecutarse con dicha entrada. El logger debe ir guardando en un fichero la fecha y hora de ejecución de cada nodo, y la salida producida por el nodo.

Para aplicar el patrón *Decorator* en nuestro caso, la clase a decorar será `StateGraph`, pero al decorar un `StateGraph`, también habrá que decorar sus nodos cuando se añadan al flujo. Es decir, habrá que implementar el patrón dos veces, para `StateGraph` y para nodos, de tal manera que: (i) el decorador de `StateGraph` cree un decorador de nodo al añadir un nodo al flujo, y (ii) el decorador de nodo actúe (creando trazas, grabando a fichero) cuando se llame al método que ejecuta el código del nodo.

A modo de ejemplo, el siguiente código decora un `StateGraph` con los dos decoradores. Dado el esquema del patrón, se hace necesario decorar el flujo antes de añadir los nodos al flujo decorado.

```
StateGraph<NumericData>          g = new StateGraph<>("loop-down", "Get a number, and decrease if positive");
StateGraphLogger<NumericData>  lg = new StateGraphLogger<>(g, "traces.txt"); // guarda info en traces.txt
StateGraphProfiler<NumericData>sg = new StateGraphProfiler<>(lg);            // almacena timestamps

sg.addNode("decrease", (NumericData mo) -> mo.put("op1", mo.get("op1")-1 ))           // decrementa op1
  .addConditionalEdge("decrease", "decrease", (NumericData mo) -> mo.get("op1") > 0 ) // decrementa hasta 0
  .setInitial("decrease");

NumericData input = new NumericData(3, 0);
System.out.println(sg+"\ninput = " + input);
NumericData output = sg.run(input, true);
System.out.println("result = " + output);
System.out.println("history = "+ sg.history());
```

El decorador `StateGraphProfiler` tendrá un método `history()` que devuelve una lista de trazas, cada una indicando el nombre del nodo, su entrada, y el tiempo de ejecución. Por ejemplo, el resultado de ejecutar el flujo anterior es:

Salida esperada:

```
Workflow 'loop-down' (Get a number, and decrease if positive):
 - Nodes: {decrease=Node decrease (1 output nodes) [logged] [profiled]}
 - Initial: decrease
 - Final: None
input = {op1=3, op2=0, result=0}
 Step 1 (loop-down) - input: {op1=3, op2=0, result=0}
 Step 2 (loop-down) - decrease executed: {op1=2, op2=0, result=0}
 Step 3 (loop-down) - decrease executed: {op1=1, op2=0, result=0}
 Step 4 (loop-down) - decrease executed: {op1=0, op2=0, result=0}
result = {op1=0, op2=0, result=0}
history = [[decrease with: {op1=3, op2=0, result=0} 20.6227 ms], [decrease with: {op1=2, op2=0, result=0}
0.1469 ms], [decrease with: {op1=1, op2=0, result=0} 0.1101 ms]]
```

En la salida anterior, puedes observar que los decoradores de nodo también actúan sobre el método `toString` (añaden "*[logged]*" o "*[profiled]*" a continuación del string, como puedes ver en la segunda línea de la salida). Por otra parte, tras la ejecución, el fichero *traces.txt* tendrá el siguiente contenido:

```
[05/04/2025 - 12:06:18] node decrease executed, with output: {op1=2, op2=0, result=0}
[05/04/2025 - 12:06:18] node decrease executed, with output: {op1=1, op2=0, result=0}
[05/04/2025 - 12:06:18] node decrease executed, with output: {op1=0, op2=0, result=0}
```

Se pide: Usando principios de orientación a objetos, crea todo el código necesario para dar soporte a decoradores para `StateGraph` y sus nodos. Crea *testers* que prueben la funcionalidad que has implementado.

Nota: tienes más información del patrón Decorator aquí: https://refactoring.guru/design-patterns/decorator

---

Normas de Entrega. Se deberá entregar:

- Un directorio **src** con el código Java de todos los apartados, incluidos los datos de prueba y **testers adicionales** que hayas desarrollado en los apartados que lo requieren (puedes usar JUnit).
- Un directorio **doc** con la documentación generada.
- Una **memoria** en formato **PDF** con una pequeña descripción de las decisiones del diseño adoptadas para cada apartado, y **con el diagrama de clases** de tu diseño.

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 2213, entregarían el fichero GR2213_MarisaPedro.zip, de manera que cuando se extraiga haya una carpeta con el mismo nombre que contenga el directorio src/, el directorio doc/, y el PDF. El **incumplimiento** de la entrega en este formato supondrá una **penalización en la nota de la práctica**.