Práctica 5

Colecciones, genericidad, expresiones lambda y patrones de diseño

Inicio: Semana del 15 de Abril
Duración: 3 semanas
Entrega: 5 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, empleando expresiones lambda y patrones de diseño de manera práctica. Para ello, construiremos una librería para la gestión de los estados por los que transitan los objetos de clases arbitrarias. Las clases de la librería permitirán definir estados para los objetos de una clase en base a expresiones lambda, monitorizar los cambios de los objetos, y extraer procesos a partir de estos cambios.

Apartado 1: Monitorización de estados (3,5 puntos)

Empezaremos creando una clase genérica `ObjectStateTracker` que permita definir estados para los objetos de cualquier clase, de manera externa. Por ejemplo, podríamos definir estados para objetos de tipo `Registro`, que permitan identificar el estado de las inscripciones a eventos; o bien para clases tales como `Matrícula` (para monitorizar matrículas universitarias) o `Exposición` (para realizar seguimiento de exposiciones artísticas). `ObjectStateTracker` se parametrizará con dos tipos, uno para la clase de los objetos, y otro para los estados. Típicamente, pero no necesariamente, los estados podrán ser tipos enumerados. El único requisito para los estados es que definan un *orden natural*, que la clase `ObjectStateTracker` utilizará para ordenarlos. El constructor de `ObjectStateTracker` recibirá en el constructor todos los estados válidos para la clase.

El método `withState` de `ObjectStateTracker` recibe como parámetro el estado y una expresión lambda que indica que, si la expresión se cumple (se evalúa a *true*), se considera que el objeto está en dicho estado. La evaluación de los estados se realiza en el orden en el que se añaden las condiciones. Además, el método `elseState` indica el estado del objeto si ninguna de las condiciones anteriores se cumple. En estos dos métodos, el estado debe ser uno de los estados válidos que se pasó en el constructor, de otra manera se debe lanzar una excepción `IllegalStateException`. El método `updateStates` actualiza los estados de los objetos de acuerdo con las expresiones lambda.

El siguiente programa muestra el uso de la clase para monitorizar objetos de tipo `Registration` (que modela registros a conferencias). Estos objetos tienen un nombre y un tipo de registro (`FULL`, `STUDENT`, `MEMBER`), y pueden estar en 6 estados, dados por el enumerado `RegistrationState`. En Moodle tienes una implementación de `Registration`, `RegistrationKind` y `RegistrationState`.

```
public class TesterStateChanges {
  protected ObjectStateTracker<Registration, RegistrationState> regState;
  protected Registration annSmith, johnDoe, lisaMartin;

  public static void main(String[] args) {
        TesterStateChanges tsc = new TesterStateChanges();
        tsc.createRegistrations();
        System.out.println(tsc.regState);
        tsc.changeRegistrations();
        System.out.println(tsc.regState);
  }
  protected void changeRegistrations() {
        this.annSmith.setAffiliation("University of Miskatonic"); // now it is filled
        this.johnDoe.pay(STUDENT.getPrice());                     // becomes payed
        this.regState.updateStates();
  }
  protected void createRegistrations() {
        this.regState = new ObjectStateTracker<>(RegistrationState.values());
        regState.withState(PAYED,     r -> r.getAmountPayed()==r.getTotalAmount() && !r.getValidated())
                .withState(STARTED,   r -> r.getAffiliation()==null && !r.getValidated())
                .withState(FILLED,    r -> r.getAffiliation()!=null && !r.getValidated())
                .withState(VALIDATED, r -> r.getAmountPayed()==0 && r.getValidated())
                .withState(FINISHED,  r -> r.getAmountPayed()==r.getTotalAmount() && r.getValidated())
                .elseState(REJECTED);

        this.annSmith   = new Registration("Ann Smith", FULL);
        this.johnDoe    = new Registration("John Doe", STUDENT);
        this.lisaMartin = new Registration("Lisa Martin", MEMBER);
        this.regState.addObjects(annSmith, johnDoe, lisaMartin);
  }
}
```

Salida esperada:

```
{STARTED=[Reg. of: Ann Smith, Reg. of: John Doe, Reg. of: Lisa Martin], FILLED=[], VALIDATED=[], PAYED=[], FINISHED=[], REJECTED=[]}
{STARTED=[Reg. of: Lisa Martin], FILLED=[Reg. of: Ann Smith], VALIDATED=[], PAYED=[Reg. of: John Doe], FINISHED=[], REJECTED=[]}
```

Nota: La salida muestra los estados ordenados por su orden natural.

Apartado 2: Igualdad de objetos (0,75 puntos)

La clase `ObjectStateTracker` debe evitar almacenar objetos que se consideran iguales. En nuestro ejemplo, consideraremos iguales dos objetos `Registration` si tienen el mismo nombre. Modifica las clases `Registration` y `ObjectStateTracker` (en caso necesario) para que el siguiente programa produzca la salida de más abajo.

```
public class TesterRepeatedObjects extends TesterStateChanges {
  public static void main(String[] args) {
        TesterRepeatedObjects tsc = new TesterRepeatedObjects();
        tsc.createRegistrations();
        tsc.regState.addObjects(new Registration("Ann Smith", STUDENT)); // Discarded, since repeated
        System.out.println(tsc.regState);
  }
}
```

Salida esperada:

```
{STARTED=[Reg. of: Ann Smith, Reg. of: John Doe, Reg. of: Lisa Martin], FILLED=[], VALIDATED=[], PAYED=[], FINISHED=[], REJECTED=[]}
```

Apartado 3: Trayectorias de estados (2 puntos)

A continuación, extenderemos la clase `ObjectStateTracker` para soportar el trazado de los cambios de estado de cada uno de los objetos almacenados. Para ello, añadiremos un nuevo método `trajectory` que devolverá la trayectoria del objeto que se le pasa como parámetro. La trayectoria será una secuencia de elementos con información del estado origen y destino, y la fecha y hora del cambio.

```
public class TesterTrajectories extends TesterStateChanges{
  public static void main(String[] args) {
        TesterTrajectories tsc = new TesterTrajectories();
        tsc.createRegistrations();
        tsc.changeRegistrations();
        tsc.displayTrajectories();
  }
  @Override
  protected void changeRegistrations() {
        super.changeRegistrations();
        this.johnDoe.setValidated(true);
        this.lisaMartin.setAffiliation("Arkham College");
        this.regState.updateStates();
  }
  protected void displayTrajectories() {
        for (Registration r : List.of(annSmith, johnDoe, lisaMartin))
              System.out.println(r+": "+this.regState.trajectory(r));
  }
}
```

Salida esperada (las marcas de tiempo variarán en función de la ejecución):

```
Reg. of: Ann Smith: [(in: STARTED at: 2024-03-30T19:16:08.259699600), (from: STARTED to FILLED at: 2024-03-30T19:16:08.259699600)]
Reg. of: John Doe: [(in: STARTED at: 2024-03-30T19:16:08.259699600), (from: STARTED to PAYED at: 2024-03-30T19:16:08.259699600), (from: PAYED to FINISHED at: 2024-03-30T19:16:08.259699600)]
Reg. of: Lisa Martin: [(in: STARTED at: 2024-03-30T19:16:08.259699600), (from: STARTED to FILLED at: 2024-03-30T19:16:08.259699600)]
```

Nota: La clase `LocalDateTime` puede serte útil para almacenar las marcas temporales.

Apartado 4: Procesos (2,5 puntos)

A continuación, crearemos una clase llamada `Process` para extraer un proceso a partir de una colección de trayectorias. Un proceso agrega todas las posibles transiciones efectuadas por los objetos, y sirve para comprender qué cambios son posibles y con qué frecuencia se realizan. Un proceso contiene la siguiente información sobre cada posible estado: número de veces que el estado ha sido inicial y final de una trayectoria, y número de veces que algún objeto ha cambiado desde ese estado a otro estado.

La clase `Process` será genérica, parametrizada con el tipo de los estados, y recibirá en el constructor todos los posibles estados. Para facilitar la obtención de las trayectorias a partir de un objeto `ObjectStateTracker`, debemos poder usar dichos objetos dentro de un `for` mejorado, lo que devolverá cada uno de los objetos almacenados en el `ObjectStateTracker`. Esto se consigue haciendo que la clase `ObjectStateTracker` implemente la interfaz genérica `Iterable<T>`.

A modo de ejemplo, el siguiente programa debe resultar en la salida de más abajo.

```
public class TesterProcess extends TesterTrajectories{
  public static void main(String[] args) {
        TesterProcess tsc = new TesterProcess();
        tsc.createRegistrations();
        tsc.changeRegistrations();
        tsc.displayTrajectories();
        tsc.buildProcess();
  }
  @Override
  protected void changeRegistrations() {
        super.changeRegistrations();
        this.lisaMartin.setValidated(true);
        this.regState.updateStates();
        this.lisaMartin.pay(MEMBER.getPrice());
        this.regState.updateStates();
  }
  protected void buildProcess() {
        Process<RegistrationState> regProcess = new Process<>(RegistrationState.values());
        for (Registration r : this.regState) // iterates on all Registrations
              regProcess.add(this.regState.trajectory(r));
        System.out.println(regProcess);
  }
}
```

Salida esperada (las marcas de tiempo variarán en función de la ejecución):

```
Reg. of: Ann Smith: [(in: STARTED at: 2024-03-30T19:22:33.490964), (from: STARTED to FILLED at:
2024-03-30T19:22:33.494979900)]
Reg. of: John Doe: [(in: STARTED at: 2024-03-30T19:22:33.494979900), (from: STARTED to PAYED at:
2024-03-30T19:22:33.494979900), (from: PAYED to FINISHED at: 2024-03-30T19:22:33.494979900)]
Reg. of: Lisa Martin: [(in: STARTED at: 2024-03-30T19:22:33.494979900), (from: STARTED to FILLED at:
2024-03-30T19:22:33.494979900), (from: FILLED to VALIDATED at: 2024-03-30T19:22:33.494979900), (from:
VALIDATED to FINISHED at: 2024-03-30T19:22:33.494979900)]
STARTED (initial 3 times, final 0 times):
  to state FILLED: 2 times
  to state PAYED: 1 times
FILLED (initial 0 times, final 1 times):
  to state VALIDATED: 1 times
VALIDATED (initial 0 times, final 0 times):
  to state FINISHED: 1 times
PAYED (initial 0 times, final 0 times):
  to state FINISHED: 1 times
FINISHED (initial 0 times, final 2 times):
REJECTED (initial 0 times, final 0 times):
```

Apartado 5: Mejorando la usabilidad mediante el patrón *Observer* (1,25 puntos)

Un inconveniente de la librería que hemos diseñado es que se ha de llamar al método `updateStates` de `ObjectStateTracker` cada vez que se quieren actualizar los estados. Una mejor solución – aunque más intrusiva para el código que use `ObjectStateTracker` – es usar el patrón de diseño `Observer` (se recomienda estudiar las transparencias del tema 3 de teoría de la asignatura). Para ello, permitiremos registrar objetos monitorizadores (que actúan como observadores en el patrón) usando el método de clase `withTracker`. A diferencia del patrón estándar, aquí registraremos los observadores a nivel de clase. Por generalidad, permitiremos más de un objeto observador, y no necesariamente de tipo `ObjectStateTracker`, por lo que debes realizar un diseño que lo permita. Los métodos de la clase `Registration` que modifiquen los atributos de la clase deberán notificar a todos los objetos registrados, lo que evita tener que llamar a `updateStates` de manera externa.

A modo de ejemplo, el siguiente código debe dar la salida de más abajo (y el código de los apartados anteriores debe seguir funcionando).

```
public class TesterObserver extends TesterStateChanges{
  public static void main(String[] args) {
        TesterObserver tsc = new TesterObserver();
        tsc.createRegistrations();
        Registration.withTracker(tsc.regState);
        System.out.println(tsc.regState);
        tsc.changeRegistrations();
        System.out.println(tsc.regState);
  }
  @Override
  protected void changeRegistrations() {
        this.annSmith.setAffiliation("University of Miskatonic"); // now it is filled
        this.johnDoe.pay(STUDENT.getPrice());                     // becomes payed
        // regState.updateStates(); // not needed anymore
  }
}
```

Salida esperada:

```
{STARTED=[Reg. of: Ann Smith, Reg. of: John Doe, Reg. of: Lisa Martin], FILLED=[], VALIDATED=[], PAYED=[],
FINISHED=[], REJECTED=[]}
{STARTED=[Reg. of: Lisa Martin], FILLED=[Reg. of: Ann Smith], VALIDATED=[], PAYED=[Reg. of: John Doe],
FINISHED=[], REJECTED=[]}
```

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 de 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, y dentro de la misma, el directorio src/, el doc/ y el PDF. El **incumplimiento** de la entrega en este formato supondrá una **penalización en la nota de la práctica**.