Práctica 5

Colecciones, genericidad, expresiones lambdas y patrones de diseño

Inicio: Semana del 18 de Abril
Duración: 3 semanas
Entrega: 6 de Mayo, 23:55h (todos los grupos)
Peso de la práctica: 30%

El objetivo de esta práctica es aprender el uso de diferentes tipos de colecciones, diseñando programas genéricos que sean capaces de adaptarse a tipos paramétricos, y empleando patrones de diseño de manera práctica. Para ello, construiremos una librería de plantillas de generación de texto, que permitan parametrizar la salida de acuerdo a una clase, y a una serie de estrategias de generación. Estas plantillas serán útiles en múltiples escenarios, por ejemplo, para la generación de emails personalizados a partir de datos de clientes, para la síntesis de páginas HTML a partir de objetos con información de productos, la persistencia de objetos en ficheros JSON, etc.

Apartado 1: Listas ordenadas (2 puntos)

En primer lugar, diseñaremos un tipo de lista – al que llamaremos `SortedList` – que mantenga sus elementos ordenados *en todo momento*. Esta lista debe ser compatible con la interfaz `java.util.List`, pero requerirá que sus elementos definan el *orden natural*. La ordenación se realizará mediante el orden natural de los elementos, así como por medio de objetos `Comparator`, que pueden añadirse mediante el método `addCriterion`. De esta manera, si dos objetos se consideran iguales por el orden natural, se utilizará el primer objeto `Comparator`. Si persiste la igualdad se pasará al siguiente objeto `Comparator` (si lo hay), y así sucesivamente. Si la igualdad persiste y no existen más `Comparators`, los objetos se consideran iguales. Debes procurar un diseño lo más general posible, de tal forma que sea posible añadir objetos `Comparator` que comparen objetos de las superclases que la lista almacena (p.ej, si una `SortedList` almacena `Persons`, le podremos pasar tanto `Comparator<Person>` como `Comparator<Object>`).

El siguiente listado muestra un ejemplo de uso de `SortedList`, que almacena objetos de tipo `Person`. `Person` es una clase que define un orden natural basado en el orden alfabético del nombre. El `Comparator` que se pasa como argumento de `addCriterion` define la fecha de nacimiento como factor de desempate en caso de igualdad de nombre.

```
public class SorterMain {
        public static void main(String[] args) {
                SortedList<Person> sorted = createPeopleList();
                List<Person> list = sorted;              // SortedList is compatible with List
                System.out.println(list);
                sorted.addCriterion(new Comparator<>() {
                        @Override public int compare(Person o1, Person o2) {
                                return o1.getBirthDate().compareTo(o2.getBirthDate());
                        }
                });
                System.out.println(list);
        }

        public static SortedList<Person> createPeopleList() {
                Person people[] = {
                                new Person("Peter", LocalDate.of(2005, 10, 12)),
                                new Person("Peter", LocalDate.of(1974, 3, 29)),
                                new Person("Paul",  LocalDate.of(2014, 6, 19)),
                                new Person("Mary",  LocalDate.of(2001, 1, 1))};

                SortedList<Person> sorted = new SortedList<>(List.of(people));
                return sorted;
        }
}
```

Salida esperada:

```
[Mary (born: 2001-01-01), Paul (born: 2014-06-19), Peter (born: 2005-10-12), Peter (born: 1974-03-29)]
[Mary (born: 2001-01-01), Paul (born: 2014-06-19), Peter (born: 1974-03-29), Peter (born: 2005-10-12)]
```

Se pide:
La clase `SortedList`, así como la clase `Person` para que el programa de ejemplo produzca la salida indicada.

Apartado 2: Plantillas simples de generación de texto (3.5 puntos)

En este apartado, construiremos una clase genérica `Template`, que servirá para definir plantillas de generación de texto, que será parametrizables con un tipo. En el listado de más abajo, se parametriza con el tipo `Person`, pero cualquier otro tipo puede usarse también.

```
public class TemplateMain {
  public static void main(String[] args) {
        Template<Person> simpleLetter = createLetterTemplate();
        addDataObjects(simpleLetter);

        Map<Person, String> result = simpleLetter.emit();
        for (Person p: result.keySet())
                System.out.println(result.get(p));

        System.out.println(simpleLetter.emit(new Person("Jude",  LocalDate.of(2018, 5, 5))));
  }
  public static void addDataObjects(Template<Person> simpleLetter) {
        simpleLetter.addObjects( new Person("Peter", LocalDate.of(1974, 4, 1)),
                                 new Person("Peter", LocalDate.of(2005, 10, 12)),
                                 new Person("Paul",  LocalDate.of(2014, 6, 19)),
                                 new Person("Mary",  LocalDate.of(2001, 1, 1)))
                    .withSortingCriteria((Person p1, Person p2) -> p1.getAge()-p2.getAge());
  }
  public static Template<Person> createLetterTemplate() {
        Template<Person> simpleLetter = new Template<>();
        simpleLetter.add("Dear ##,\nHow are you today?", p -> p.getName())
                    .add("Since you were born on ##, you are ## years old.",p -> p.getBirthDate(), p -> p.getAge());
        return simpleLetter;
  }
}
```

La clase `Template` tendrá un constructor vacío, y el objeto `Template` se irá construyendo mediante llamadas al método `add`. Este método recibe un String con el texto a emitir, con "huecos" definidos mediante los caracteres ##. De esta manera, la cadena: "*Dear ##,\How are you today?*." contiene 1 hueco. Los huecos deben rellenarse con valores que se obtienen de objetos del tipo del parámetro de la clase `Template`. El cálculo del valor del hueco, se especifica mediante expresiones lambda, que reciben como parámetro un objeto (del tipo del `Template`), y generan un valor de cualquier tipo. Por ejemplo, en el listado, la expresión lambda `p -> p.getName()` recibe un parámetro `p` (cuyo tipo implícito es el parámetro del `Template`, `Person` en este caso), y devuelve un `String` (el nombre de la persona). Debes diseñar un método que reciba un número variable de expresiones lambda. Si se pasan menos lambdas que huecos existen en el `String`, los huecos restantes mantienen su valor "##". Si se pasan más lambdas, simplemente se ignoran.

Un objeto `Template` también debe configurarse con los objetos para los que se quiere emitir el texto mediante el método `addObjects`. Adicionalmente, se pueden añadir criterios de ordenación para los objetos, con el método `withSortingCriteria`. Internamente, el `Template` guardará estos objetos en una SortedList.

Finalmente, el método `emit` debe generar el texto sobre cada objeto, y debe devolver el resultado en un mapa, donde cada objeto tendrá asociado el `String` con el resultado de la generación. Como ves en el listado, existe otra variante de `emit` que recibe un objeto y devuelve el `String` resultado de la emisión. Así, el listado anterior debe producir la siguiente salida:

```
Dear Mary,
How are you today?
Since you were born on 2001-01-01, you are 21 years old.

Dear Paul,
How are you today?
Since you were born on 2014-06-19, you are 7 years old.

Dear Peter,
How are you today?
Since you were born on 2005-10-12, you are 16 years old.

Dear Peter,
How are you today?
Since you were born on 1974-04-01, you are 48 years old.

Dear Jude,
How are you today?
Since you were born on 2018-05-05, you are 3 years old.
```

Apartado 3: Plantillas condicionales e iteradoras (2 puntos)

En este apartado, añadiremos la posibilidad de construir plantillas con condicionales e iteraciones, mediante los métodos `addWhen` y `addForEach`, como muestra el siguiente listado.

```
public class IteratedTemplateMain {
  public static void main(String[] args) {
    Template<Person> simpleLetter = createLetterTemplate();
    addDataObjects(simpleLetter);
    Map<Person, String> result = simpleLetter.emit();

    for (Person p: result.keySet())
       System.out.println(result.get(p));
  }
  public static Template<Person> createLetterTemplate() {
    Template<Person> simpleLetter = TemplateMain.createLetterTemplate();
    simpleLetter.addWhen(p -> p.getAge()>=65, "Contact us if thinking about retirement...")
                .addWhen(p->p.getMascots().size()>0, "... and greetings to your mascots:")
                .addForEach(p->p.getMascots(),
                            "  Ey ##, you are a nice ##!",
                            m -> m.getMascotName(),
                            m -> m.getMascotType());
    return simpleLetter;
  }
  public static void addDataObjects(Template<Person> simpleLetter) {
    Person peter = new Person("Peter", LocalDate.of(1957, 4, 1)),
           mary = new Person("Mary",  LocalDate.of(2001, 1, 1));

    peter.addMascots(new Mascot("cat", "Felix"), new Mascot("canary", "Chirps"));
    simpleLetter.addObjects(peter, mary);
  }
}
```

El método `addWhen` recibe como primer parámetro una lambda que expresa una condición booleana sobre el objeto, y luego un `String` con el texto a emitir, y las lambdas para rellenar los huecos. El texto del `String` sólo se emitirá si el resultado de la evaluación de la primera lambda es `true`. De este modo, `p -> p.getAge()>=65` es una lambda que recibe un parámetro p (cuyo tipo implícito es el parámetro del template, `Person` en este caso) y devuelve si su edad es mayor o igual que 65. Así, el texto sólo se emitirá para los objetos `Person` con edad mayor o igual que 18.

El método `addForEach` recibe como primer parámetro una lambda que devuelve una colección de objetos de cualquier tipo (`Mascot`, en el caso de `getMascots`), como segundo parámetro un String con huecos, y a continuación una lista de lambdas para rellenar esos huecos. No obstante, estas lambdas reciben un objeto del tipo de la colección que se devuelve. Por ejemplo, el tipo implícito del parámetro de la expresión lambda `m -> m.getMascotName()` es `Mascot`, y la expresión devuelve un `String` con el nombre de la mascota. El método `emit` generará una substitución del `String` por cada elemento de la colección que recibe del primer parámetro.

Debes actualizar tu diseño para incorporar estos dos métodos – junto con las clases que puedan ser necesarias – actualizar la clase `Person` y añadir la clase `Mascot` para que el programa anterior produzca la siguiente salida.

```
Dear Mary,
How are you today?
Since you were born on 2001-01-01, you are 21 years old.


Dear Peter,
How are you today?
Since you were born on 1957-04-01, you are 65 years old.
Contact us if thinking about retirement...
... and greetings to your mascots:
  Ey Felix, you are a nice cat!
  Ey Chirps, you are a nice canary!
```

Como ves en la salida, Mary no es mayor de 64 años ni tiene mascotas, por lo que no se genera ningún `String` a partir de los métodos `addWhen` y `addForEach`. Sin embargo, Peter tiene 65 años y dos mascotas, por lo que se emite el texto con dos copias del String especificado en el `addForEach`, una por cada mascota.

Apartado 4: Añadiendo estrategias (2.5 puntos)

Por último, extenderemos el diseño realizado para soportar estrategias que permitan configurar la generación de texto. Estas opciones se pasarán como argumentos al método `withOptions`, que puede recibir un número arbitrario de opciones. Como ves en el listado de más abajo, se han creado tres tipos de estrategias (`TimeStamper`, `UpperCaser` y `FilePersister`) que añaden distintos comportamientos:

- `TimeStamper` emite la fecha actual antes de la generación de texto por el resto de la plantilla.
- `UpperCaser` transforma el texto de la plantilla a mayúsculas.
- `FilePersister` imprime en ficheros el texto generado. Su constructor recibe una lambda, que recibe un argumento del tipo de la plantilla, del que obtiene el nombre del fichero a generar. A este nombre se le añade un contador por cada nombre distinto, y la extensión ".txt".

```
public class StrategyMain {
        public static void main(String[] args) {
                Template<Person> simpleLetter = IteratedTemplateMain.createLetterTemplate();
                IteratedTemplateMain.addDataObjects(simpleLetter);

                simpleLetter.withOptions(new TimeStamper<>(),
                                         new UpperCaser<>(),
                                         new FilePersister<Person>(p -> p.getName()));
                System.out.println(writeResult(simpleLetter.emit()));
        }
        private static String writeResult(Map<Person, String> emit) {
                String res = "";
                for (Person p : emit.keySet())
                        res+=emit.get(p)+"\n--------------------------------------------------------\n";
                return res;
        }
}
```

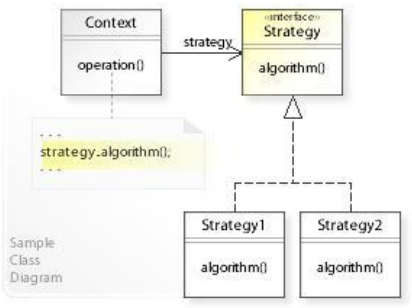
De esta manera, el listado anterior debe producir la salida de más abajo, así como dos ficheros: Mary0.txt y Peter0.txt, cada uno con el String generado por el objeto. Nótese que si hubiera más objetos con nombre Mary, el resultado se salvaría en ficheros Mary1.txt, Mary2.txt, etc. Debes realizar un diseño extensible que permita añadir fácilmente nuevas estrategias.

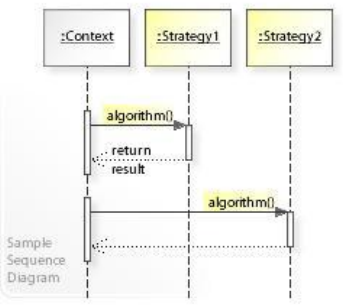
```
2022-04-01
DEAR MARY,
HOW ARE YOU TODAY?
SINCE YOU WERE BORN ON 2001-01-01, YOU ARE 21 YEARS OLD.

--------------------------------------------------------
2022-04-01
DEAR PETER,
HOW ARE YOU TODAY?
SINCE YOU WERE BORN ON 1957-04-01, YOU ARE 65 YEARS OLD.
CONTACT US IF THINKING ABOUT RETIREMENT...
... AND GREETINGS TO YOUR MASCOTS:
  EY FELIX, YOU ARE A NICE CAT!
  EY CHIRPS, YOU ARE A NICE CANARY!

--------------------------------------------------------
```

Pista: para realizar este ejercicio una buena opción es el uso del patrón de diseño Strategy. Este patrón (cuyo esquema se muestra más abajo), permite modelar una familia de algoritmos (estrategias de generación del texto en nuestro caso) que facilitan adaptar el algoritmo base. En este caso, debes tener en cuenta que, cada estrategia de generación afecta al resultado en un punto del proceso (al inicio, en cada String generado, al final), y que podemos tener varias estrategias encadenadas. Además, en el listado, las estrategias concretas son clases genéricas ya que algunas (como `FilePersister`) necesitan información del objeto sobre el que se aplica la plantilla.





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 adoptado para ejecución de cada apartado, y con el diagrama de clases de tu diseño.
- El **diagrama de clases** final resultante.

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**.