Práctica 4

Colecciones, genericidad, expresiones lambdas y patrones de diseño

Inicio: Semana del 26 de Abril
Duración: 2 semanas
Entrega: 9 de Mayo, 23:55h
Peso de la práctica: 25%

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 algunas clases que ayuden a crear formularios de entrada, que tomen la entrada del usuario desde la consola, y que luego sean capaces de procesar los datos recogidos.

Apartado 1: Formularios genéricos de entrada de datos por consola (4,5 puntos)

Queremos construir una clase llamada `Form`, que es un formulario para la recogida de datos de entrada desde consola. La clase debe poder configurarse con los campos del formulario, de tipo `Field`. Estos campos serán genéricos, parametrizados con el tipo esperado de la respuesta del usuario. Además, los campos deben configurarse con una función para convertir desde el `String` leído por consola, al tipo del campo, que se pasará en el constructor. De manera opcional, se podrán añadir uno o más validadores. Estos validadores contendrán una función, sobre el tipo del campo, que hace una comprobación de corrección (por ejemplo, que la edad leída en el campo sea mayor de 18 años) y el mensaje que ha de presentarse al usuario si la validación no se cumple.

La clase `Form` debe tener un método `exec`, que presenta (en orden) cada campo al usuario, recoge la respuesta (validándola) y devuelve un mapa con el texto asociado a cada campo como clave, y la respuesta del usuario (convertida al tipo del campo) como valor.

A modo de ejemplo, se muestra un programa de prueba y una posible ejecución. Como puedes observar, el formulario no realiza preguntas repetidas ("*are you married?*"), y se configura mediante llamadas al método `add`. En este programa, hemos configurado el conversor y los validadores de los campos mediante expresiones lambda.

```
package forms;

import java.io.IOException;

public class FormMain {
  public static void main(String[] args) throws IOException {
        Form enrollForm = new Form();

        Field<Integer> age = new Field<Integer>(s -> Integer.valueOf(s)).
                              addValidation(a -> a > 18, "value should be bigger than 18").
                              addValidation(a -> a < 66, "value should be less than 66");
        Field<Boolean> yesNo = new Field<>(s -> s.toUpperCase().equals("YES"));
        enrollForm.add("What is your age?", age).
                   add("Are you married?", yesNo).
                   add("Do you have children?", yesNo).
                   add("Are you married?", yesNo);

        System.out.println(enrollForm.exec());
  }
}
```

```
What is your age? > 3

Invalid value: 3
  value should be bigger than 18
What is your age? > 34
Are you married? > yes
Do you have children? > no
{What is your age?=34, Are you married?=true, Do you have children?=false}
```

(donde los datos en verde es la información introducida por el usuario)

Apartado 2: Procesando los datos recogidos (3 puntos)

En este apartado crearemos una clase `DataAggregator` que agrupará las respuestas a los formularios, ordenando cada respuesta de acuerdo al *orden natural* definido por cada tipo de cada campo del cuestionario. Para ello, debes exigir que los tipos de datos de cada `Field`, así como los datos del mapa devueltos por el método `exec` sean compatibles con la interfaz `Comparable`. Por simplicidad, la clase `DataAggregator` sólo presentará los datos, pero en una aplicación real tendría muchas otras funcionalidades, como la separación de los datos en intervalos, o el cálculo de estadísticas.

El siguiente programa ejemplifica su uso. Se te pide completar además la clase `Address` para obtener la salida de más abajo. Observa que las respuestas de tipo `Address` se ordenan por código postal, y en caso de igualdad, por orden alfabético de la calle. Las respuestas de tipo `Integer` se ordenan de menor a mayor.

```
class Address _____________________________ /* Completar si es necesario */ {
        private String address;
        private int postalCode;

        public Address(String adr, int pc) {
                this.address = adr;
                this.postalCode = pc;
        }
        public int postalCode() { return this.postalCode; }
        public String toString() {
                return this.address+" at PC("+this.postalCode+")";
        }
        // Completar si es necesario
}

public class ProcessingMain {
  public static void main(String[] args) throws IOException {
        Form censusForm = new Form();

        Field<Address> adr = new Field<Address>(s -> { String[] data = s.split(";");
                                                       return new Address(data[0], Integer.valueOf(data[1].trim()));
                                                     }).
                             addValidation(a -> a.postalCode() >= 0, "Postal code should be positive");
        Field<Integer> np  = new Field<Integer>(s->Integer.valueOf(s)).
                             addValidation(s->s>0, "value greater than 0 expected");

        censusForm.add("Enter address and postal code separated by ';'", adr).
                   add("Number of people living in that address?", np);
        DataAggregator dag = new DataAggregator();
        for (int i=0; i<3; i++)
          dag.add(censusForm.exec());

        System.out.println(dag);
   }
}
```

```
Enter address and postal code separated by ';' > Main St.;45007
Number of people living in that address? > 6
Enter address and postal code separated by ';' > Baker St.;45007
Number of people living in that address? > 2
Enter address and postal code separated by ';' > Regent St.;23008
Number of people living in that address? > 1
{Enter address and postal code separated by ';'=[Regent St. at PC(23008), Baker St. at PC(45007), Main St. at
PC(45007)], Number of people living in that address?=[1, 2, 6]}
```

Apartado 3: Protegiendo los formularios mediante una contraseña (2,5 puntos)

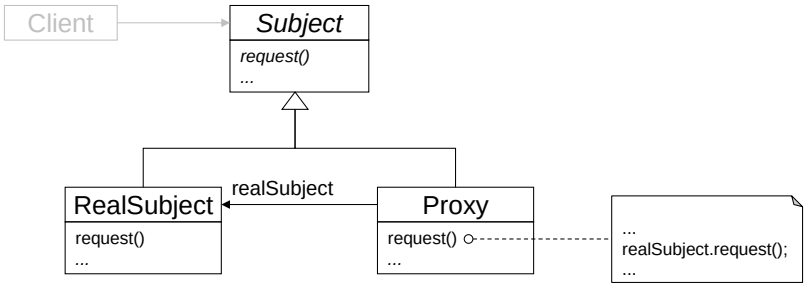
Finalmente, vamos dar la opción de proteger el formulario por medio de una contraseña. Esta modificación en el diseño no debe interferir con el código ya desarrollado, de tal manera que los programas de los apartados 1 y 2 deben seguir funcionando. La idea es que el usuario que responde al formulario introduzca una contraseña antes de usar el formulario, si no lo ha hecho con anterioridad. Se dan tres opciones para introducir la contraseña, quedando bloqueado el cuestionario si no se introduce la contraseña correctamente en estos tres intentos. Si hay varias ejecuciones del formulario, sólo se piden la contraseña una vez.

A modo de ejemplo, el programa de más abajo muestra el modo de uso de los formularios protegidos, y una posible ejecución. Un formulario se protege mediante el método estático `ProtectedForm.protect`, donde se proporciona el formulario a proteger y la contraseña esperada. En la ejecución que se muestra, la segunda vez que se ejecuta el formulario ya se ha introducido la contraseña, por lo que esta no se vuelve a pedir. Si la contraseña hubiese sido incorrecta después de los 3 intentos, el formulario no se mostraría, porque quedó bloqueado.

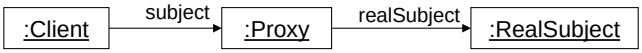
```java
public class ProtectedFormMain {
  public static void main(String[] args) throws IOException {
        Form enrollForm = new Form();

        Field<Integer> age = new Field<Integer>(s -> Integer.valueOf(s)).
                              addValidation(a -> a > 18, "value should be bigger than 18").
                              addValidation(a -> a < 66, "value should be less than 66");
        Field<Boolean> yesNo = new Field<>(s -> s.toUpperCase().equals("YES"));
        enrollForm.add("What is your age?", age).
                   add("Are you married?", yesNo);

        AbstractForm pf = ProtectedForm.protect(enrollForm, "qwerty");
        System.out.println(pf.exec());
        System.out.println(pf.exec());
  }
}
```

```
Enter password: admin
Invalid password (2 remaining attempts)
Enter password: qwerty
What is your age? > 23
Are you married? > no
{What is your age?=23, Are you married?=false}
What is your age? > 34
Are you married? > yes
{What is your age?=34, Are you married?=true}
```

Pista: para realizar este ejercicio una buena opción es el uso del patrón de diseño Proxy, que permite proveer un sustituto para otro objeto (con la misma interfaz que éste) para proteger su acceso. El siguiente diagrama muestra un esquema de su funcionamiento. Puedes encontrar más detalles de este patrón en las transparencias del curso.



Object diagram



Pregunta adicional: esta es una práctica muy simplificada, que esboza un diseño inicial para el manejo de formularios. ¿Cómo podemos extender el diseño para mejorar la usabilidad del *framework* evitando la necesidad de dar en el constructor de `Field` un conversor para tipos de datos estándar como `Integer`, `Double` o `Boolean`?

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.
- 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 la respuesta a la pregunta adicional del apartado 3.
- 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**.