Oriol Estabanell, Albert Gubau u172749, u172750

Lab 1
Programación Orientada a Objetos

1. Descripción del problema a solucionar

En primer lugar, se nos presenta un problema en el que deberemos implementar un código el cual sea capaz de añadir una serie de instrucciones a un programa e implementar las opciones correctas para empezar el programa y realizar las operaciones correspondientes a las subsecuencias de éste.

Por ejemplo, trabajaremos con el programa que dibuja un cuadrado en la pantalla “Square”:

```
REP 4
FWD 100
ROT 90
END 1
```

En nuestro programa podemos ver que se presentan 4 instrucciones como programa y estas cuatro instrucciones están formadas por un código (REP, FWD, ROT, END) y por un parámetro (4, 100, 90, 1).

Nuestro programa deberá ser capaz de leer las instrucciones y comprobar que sean correctas, también, deberá saber cuantas veces se realizará cada instrucción, según si hay REP y su valor de parámetro, y deberá saber si el programa ha finalizado.

En esta serie de instrucciones, nuestro programa deberá imprimir por pantalla lo siguiente:

```
FWD 100
ROT 90
FWD 100
ROT 90
FWD 100
ROT 90
FWD 100
ROT 90
FWD 100
ROT 90
```

Por lo tanto, deberemos definir 3 clases para implementar nuestro código:

LogoProgram → Nuestro programa principal

Instruction → Clase que gestiona las instrucciones

Program → Clase que gestiona el programa, es decir, el conjunto de instrucciones.

Los métodos que deberemos implementar en nuestras clases serán los siguientes:

La clase **LogoProgram** será la clase que utilizaremos para ejecutar nuestro código, es decir, se comportaría como una clase en la que inicializaremos los datos para ejecutar el programa.

Para la clase **Instruction** deberemos implementar los siguientes métodos:

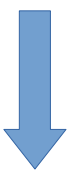
Instruction → Constructor de la clase.
getCode → Devuelve el código de la instrucción.
getParam → Devuelve el parámetro de la instrucción.
isRepInstruction → Devuelve true si es “REP” o “END” y false sino.
isCorrect → Devuelve true si la instrucción es correcta y false sino.
errorCode → Devuelve el código de error si hay un error.
info → Devuelve un string con la instrucción que se ejecuta.

Para la clase **Program** deberemos implementar los siguientes métodos:

Program → Constructor de la clase.
getName → Devuelve el nombre del programa.
addInstruction → Añade una instrucción al programa.
restart → resetea el programa en la primera linea.
hasFinished → Nos devuelve true si se ha acabado el programa y false sino.
getNextInstruction → Nos devuelve la siguiente instrucción y realiza las comprobaciones pertinentes.
isCorrect → Devuelve true si todas las instrucciones son correctas y false sino.
printErrors → Imprime por pantalla si hay errores y de que tipo son.
goToStartLoop → se dirige a la primera instrucción después de la instrucción “REP”.

-Solución escogida para la resolución del problema

La resolución que hemos realizado para nuestro programa es la del código proporcionado en el enunciado para **LogoProgram**, con ciertas modificaciones y la implementación de los métodos para **Instruction y Program** de manera que estén relacionados de la mejor manera posible para que el programa funcione de manera óptima.



-Implementación de la clase Instruction

Para la clase Instruction nos dedicamos a implementar uno por uno sus métodos principales:

-Instruction (Constructor)

El método Instruction será el método constructor de la clase, éste nos permitirá crear nuevas instancias de nuestra clase de manera correcta:

```
Instruction(String c, double p){
    code = c;
    parameter = p;
}
```

En este método asignamos el valor del código y del parámetro de la instrucción a partir de los valores que entran en el constructor asignándolos a los atributos **code** y **parameter** de la clase **Instruction**.

-getCode y getParam (getters)

Estos serán los **getters** de la clase **Instruction** los cuales nos permitirán tomar el valor de el **code** o el parámetro de la instrucción con la que estemos trabajando:

```
public String getCode(){

    return code;
}

public double getParam(){

    return parameter;
}
```

Como podemos observar lo que hacen es devolver el **String** correspondiente al **code** de la instrucción o el valor del parámetro.

-isRepInstruction

Éste método nos permite saber si la instrucción con la que estamos trabajando tiene el **code** como "**REP**" o "**END**" de manera que devolverá **true** si lo tiene y **false** en otro caso:

```
public Boolean isRepInstruction(){

    return code.equals("REP") || code.equals("END");
}
```

Cómo vemos nos bastará con devolver si uno de los dos se cumple (**or**), si uno de los dos se cumpliera el valor de retorno sería **true**, en otro caso sería **false**.

-isCorrect

El método **isCorrect** de **Instruction** nos devolverá true si la instrucción es correcta y false en cualquier otro caso. Éste método se complementa con **errorCode** puesto que si la instrucción es errónea deberemos saber que tipo de error presenta. En el método **errorCode** que veremos más adelante el valor de la instrucción correcta es el código 0, por lo tanto, en **isCorrect** deberemos comprobar si **errorCode** nos ha proporcionado un valor de **0**:

```
public Boolean isCorrect(){

    return errorCode() == 0;
}
```

Si **errorCode** nos devuelve un **0** es que la instrucción es correcta, por lo tanto, debemos comprobar si esta igualdad es cierta para comprobar que sea correcta la instrucción.

-errorCode

El método **errorCode** tendrá que gestionar el tipo de errores que se nos presentan en el código de manera que asigne un entero a cada error posible correspondiente a el parámetro de la instrucción:

```
public int errorCode(){

    if(code.equals("FWD")){

        if(parameter < -1000 || parameter > 1000){
            return 1;
        }
        return 0;

    }else if(code.equals("PEN")){

        if(parameter != 0 && parameter != 1){
            return 2;
        }
        return 0;

    }else if(code.equals("ROT")){

        if(parameter <= -360 || parameter >= 360){
            return 3;
        }
        return 0;

    }else if(code.equals("REP")){

        if(parameter <= 0 || parameter >= 1000){
            return 4;
        }
        return 0;

    }else if(code.equals("END")){

        if(parameter <= 0 || parameter >= 1000){
            return 5;
        }
        return 0;
    }

    return 6;
}
```

Como podemos observar, lo que se hace es comprobar si el parámetro de cada instrucción está fuera de los posibles valores, en el caso de que lo esté asignamos un valor de retorno para cada tipo de instrucción (ej: **1** significa que hay un error de parámetro en la instrucción **FWD**), en caso de que el valor del parámetro sea correcto entonces se devolverá un **0** como hemos visto anteriormente.

Al final de éste método podemos ver como si no se trata de ninguna de las instrucciones posibles entonces devuelve **6**, este será el valor de error que significará que el **code** de la instrucción no es un **code** válido.

-info

Éste método nos permite devolver la instrucción con la que estamos trabajando en cualquier momento en forma de **String**:

```
public String info(){

    return code+" "+Double.toString(parameter);
}
```

Éste método nos será útil para poder imprimir por pantalla las instrucciones que se ejecutan.

-Implementación de la clase Program

Esta clase nos permitirá gestionar el programa formado por una serie de instrucciones almacenadas en una **LinkedList** de instrucciones, un entero **currentLine** que nos permite saber en la línea del programa en la que estamos, un nombre de programa y un entero **loopIteration** que nos permite saber las iteraciones que quedan para acabar el programa.

-Program (Constructor)

El método Program nos permitirá crear instancias de la clase Program asignando valores a sus atributos de la siguiente manera:

```
Program(String name){

    instructions = new LinkedList<Instruction>();
    currentLine = 0;
    loopIteration = 0;
    programName = name;
}
```

Como vemos creamos una nueva **LinkedList** de instrucciones llamada **instructions**, inicializamos el valor de la **currentLine** y **loopIteration** a **0** y como nombre del programa asignamos el nombre que entra por parámetro.

-getName (getter)

Éste será el método **getter** de la clase **Program**, nos permitirá obtener el nombre del programa en cualquier momento y será el único **getter** de la clase **Program**:

```
public String getName(){

    return programName;
}
```

Como podemos ver devolvemos el atributo **programName** de la clase **Program**.

-addInstruction

Éste método añadirá instrucciones a la **LinkedList** de instrucciones (**instructions**) de manera que creará una instancia de **Instruction** con los valores que entran por parámetro del método y la añadirá a la lista. Después aplicará la función **isCorrect** de la clase **Instruction** para comprobar si ésta es correcta, devolverá **true** si lo es y **false** en cualquier otro caso:

```
public Boolean addInstruction(String c, double p){

    Instruction instruction = new Instruction(c, p);
    instructions.add(instruction);

    return instruction.isCorrect();
}
```

Realmente para la solución escogida éste valor de retorno nos es indiferente puesto que no dejará de añadir los elementos a la lista por incorrectos que sean, donde se comprobará que todas las instrucciones sean correctas será en **printErrors**.

-restart

El método **restart** nos sitúa nuestro atributo **currentLine** en **0**, puesto que, para **resetear** el programa deberemos volver a la primera línea de éste:

```
public void restart(){

    currentLine = 0;
}
```

Para volver a la primera línea escribimos que nuestra **currentLine** será **0**.

-hasFinished

Éste método nos permitirá saber si nuestro programa ya ha finalizado, para ello comprobaremos que nuestro atributo **currentLine** sea mayor o igual que el tamaño de nuestra lista de instrucciones:

```
public Boolean hasFinished(){

    return (currentLine >= instructions.size());
}
```

Éste código está muy relacionado con las diferentes comprobaciones que realizaremos en **getNextInstruction**, en ese método restaremos la **currentLine** al llegar a la instrucción de **END** y volveremos al inicio del **loop**, cuando no se vuelva al inicio del **loop** querrá decir que ya no nos quedan más iteraciones y nuestra **currentLine** no se restará sino que se sumará, por lo tanto, llegará a ser igual o mayor a el tamaño de la lista de instrucciones y entonces habrá acabado el programa.

-getNextInstruction

Este es uno de los métodos más importantes del programa puesto que accede a la siguiente instrucción y comprueba la iteración en la que estamos, además si encuentra una instrucción **REP** toma el valor de su parámetro para saber las iteraciones y si encuentra un **END** resta a la variable **loopIteration** puesto que hemos llegado a un final de iteración además de dirigirnos al inicio del **loop** con **goToStartLoop**:

```
public Instruction getNextInstruction(){

    Instruction instruction = instructions.get(currentLine);

    if(instruction.isRepInstruction()){

        if(instruction.getCode() == "REP"){

            loopIteration = (int)instruction.getParam();

        }else{

            loopIteration--;

            if(loopIteration>0){

                goToStartLoop();
                currentLine--;
            }
        }
    }

    currentLine++;

    return instruction;
}
```

En primer lugar, creamos una variable **instruction** que la inicializamos con el valor de la instrucción en la que nos encontramos (instrucción de la **currentLine**), seguidamente, comprobamos si es **REP** o **END** con **isRepInstruction**.
En caso de que sea REP asignaremos el valor de su parámetro a **loopIteration** puesto que será el numero de iteraciones a realizar. En caso de que sea **END** deberemos restar a **loopIteration** puesto que habremos llegado al final de una iteración, y deberemos regresar al inicio del **loop** con **goToStartLoop**. Después de ir al inicio del **loop** tendremos que restar al atributo **currentLine** puesto que en las líneas de después sumamos al **currentLine**.
Las líneas siguientes (**currentLine++** y **return instruction**) nos permiten avanzar en las instrucciones y devolverlas para ser imprimidas por pantalla. Como éste código se realizará sea la instrucción que sea, en el caso de **goToStartLoop** tendremos que compensarla con **currentLine--** porque sino nos situará dos por delante del inicio del **loop** imprimiendo por pantalla lo siguiente:

```
FWD 100.0
ROT 90.0
ROT 90.0
ROT 90.0
ROT 90.0
```

-isCorrect

El método **isCorrect** de **Program** comprobará que para cada instrucción de **REP** en el programa también se tenga una instrucción **END** para cerrar el bucle:

```
public boolean isCorrect(){

    int repcounter = 0;

    for(Instruction instruction: instructions){

        if(!instruction.isCorrect()){

            return false;
        }

        if(instruction.getCode() == "REP"){

            repcounter++;
        }

        if(instruction.getCode() == "END"){

            repcounter--;
        }
    }

    return (repcounter == 0);
}
```

Como vemos lo que hacemos es inicializar un contador de instrucciones **REP** (**repcounter**) e iteramos por la lista de instrucciones, en caso de que encontremos una instrucción **REP** aumentaremos el contador en una unidad y deberemos buscar a ver si hay un **END** que le corresponda. En caso de que haya un **END** en la lista de instrucciones restaremos en una unidad el contador de **REP** de manera que para que el programa sea correcto el contador deberá tener un valor final de **0**, puesto que, querrá decir que por cada **REP** encontrado también hemos encontrado su **END** correspondiente.

-printErrors

El método **printErrors** comprobará que la lista de instrucciones contenga errores, si encuentra errores entonces los imprimirá según su código de error obtenido mediante el método **errorCode** de la clase **Instruction**:

```
public void printErrors(){

    if(!isCorrect()){
        System.out.println("Hay un error.");
    }

    for(Instruction instruction: instructions){

        if(!instruction.isCorrect()){

            int errorcode = instruction.errorCode();

            if(errorcode == 1){
                System.out.println("Error in FWD parameter");


            }else if(errorcode == 2){
                System.out.println("Error in PEN parameter");


            }else if(errorcode == 3){
                System.out.println("Error in ROT parameter");


            }else if(errorcode == 4){
                System.out.println("Error in REP parameter");


            }else if(errorcode == 5){
                System.out.println("Error in END parameter");


            }else if(errorcode == 6){
                System.out.println("Error: there is a not valid instruction added.");


            }else if(errorcode == 0){
                System.out.println("All correct!");
            }
        }
    }
}
```

En primer lugar, si el programa no es correcto imprimirá por pantalla que hay un error. Seguidamente, iterará por la lista de instrucciones y para cada instrucción aplicará la función **isCorrect** de la clase **Instruction** para saber si es correcta, en caso de que no lo sea se preguntará por su código de error y se asignará a la variable **errorcode** que entrará en una serie de casos para saber cual es el error que deberá imprimir por pantalla.

-goToStartLoop

Éste método nos dirigirá al inicio del **loop**, es decir, la primera instrucción después de **REP**, de manera que buscará cual es la línea correspondiente a **REP** y nos situará la **currentLine** una posición por delante:

```
private void goToStartLoop(){

    int counter = 0;

    for(Instruction instruction: instructions){

        if(instruction.getCode() == "REP"){

            currentLine = counter + 1;
            break;
        }

        counter++;
    }
}
```

Como podemos ver, lo que haremos será inicializar un contador de líneas en **0** e iteraremos por la lista de instrucciones, cuando encuentre una instrucción **REP** asignaremos la **currentLine** una por delante de la línea en la que ha encontrado esta instrucción, es decir, la línea **counter**. En caso de que no la encontremos deberemos aumentar el **counter** en una unidad hasta encontrarla.

-LogoProgram

En cuanto a la ventana de LogoProgram, modificamos una serie de líneas de código que no eran correctas para el correcto desarrollo de nuestro código de las clases tal y como las habíamos implementado:

```
public class LogoProgram {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        Program p = new Program("Square");
        p.addInstruction("REP", 4);
        p.addInstruction("FWD", 100);
        p.addInstruction("ROT", 90);
        p.addInstruction("END", 1);

        if(p.isCorrect()){
            p.restart();
            while(!p.hasFinished()){
                Instruction instr = p.getNextInstruction();
                if(!instr.isRepInstruction()){

                    System.out.println(instr.info());
                }
            }
        }else{
            p.printErrors();
        }
    }
}
```

Como podemos ver hemos añadido una condición para que se imprima la info de la instrucción y es que ésta no sea una instrucción de tipo REP o END, porque sino se imprimirán esas instrucciones.

También hemos añadido un else en el if de si el programa es correcto, puesto que, en caso de que no sea correcto deberemos imprimir los errores pertinentes con el método printErrors.

-Solución alternativa

Nuestra solución consistiría en cambiar isCorrect de la clase Program para que cuando encontrara una instrucción REP se almacenara el valor del parámetro en la variable loopIteration. Esto nos permitiría saber el valor de loopIteration al aplicar isCorrect:

```
public boolean isCorrect(){

    int repcounter = 0;

    for(Instruction instruction: instructions){

        if(!instruction.isCorrect()){

            return false;
        }

        if(instruction.getCode() == "REP"){

            loopIteration = (int)instruction.getParam();
            repcounter++;
        }

        if(instruction.getCode() == "END"){

            repcounter--;
        }
    }

    return (repcounter == 0);
}
```

Seguidamente, deberíamos modificar hasFinished para que el programa finalice cuando no queden más iteraciones, es decir, loopIteration = 0:

```
public Boolean hasFinished(){

    return (loopIteration == 0);
}
```

Y en cuanto a getNextInstruction lo modificamos para que quede de la siguiente manera:

```
public Instruction getNextInstruction(){

    Instruction instruction = instructions.get(currentLine);

    if(instruction.isCorrect()){
        if(instruction.isRepInstruction()){

            if(instruction.getCode() == "REP"){

                currentLine++;
            }else {

                loopIteration--;
                goToStartLoop();
            }
        return getNextInstruction();
        }
    }

    currentLine++;
    return instruction;
}
```

En el que podemos ver que primero se comprueba si la instrucción es correcta, y luego si es REP avanzamos la currentLine y si es END restamos una unidad la loopIteration y nos dirigimos al inicio del loop, después de las dos comprobaciones aplicaremos recursivamente getNextInstruction hasta que tengamos una instrucción que no sea ni REP ni END. Finalmente, cuando encontremos una instrucción que no sea ni REP ni END avanzaremos la currentLine y devolveremos la instrucción para que se pueda imprimir por pantalla.

Las modificaciones que haríamos en LogoProgram serían las siguientes:

```
if(p.isCorrect()){
    p.restart();
    while(!p.hasFinished()){
        Instruction instr = p.getNextInstruction();
        if(!instr.isRepInstruction()){
            if(!p.hasFinished()){
                System.out.println(instr.info());
            }

        }
    }
}else{
    p.printErrors();
}
```

Si la instrucción no es ni REP ni END y las iteraciones no han finalizado, entonces imprimiremos por pantalla la info de la instrucción.

La implementación alternativa no sería del todo correcta puesto que en un caso como este no imprimiría el ROT final:

FWD 23
ROT 23
REP 3
FWD 34
END 1
ROT 12

Por lo tanto, quizás dándole unas cuantas vueltas más a esta solución podríamos llegar a implementarla del todo pero por el momento solo se quedaría en una alternativa incorrecta.

-Relación de las soluciones con conceptos de teoria

Como podemos deducir, la relación que existe entre la clase program y la clase Instruction es de agregación, puesto que, una puede existir sin la otra pero a la vez Program utiliza métodos de Instruction de manera que mantienen esa relación vista en teoria.

También podriamos deducir que la relación que mantienen LogoProgram y Program es de asociación directa puesto que Program vive en LogoProgram.

-Dificultades para realizar la práctica

En cuanto a los problemas que nos han surgido para realizar la práctica podemos decir que nos han surgido bastantes. Para empezar, los métodos no han sido correctamente explicados en un inicio por los profesores o por el documento a seguir por lo que se hace más complejo descifrar por un nombre de un método lo que debe de hacer. Esto sumado al poco tiempo del que hemos dispuesto des de que se nos han resuelto las dudas hasta que hemos podido solucionar los problemas ha hecho de esta práctica una tarea difícil de llevar a cabo.

En cuanto al apartado de código, nos resultó difícil entender como podíamos usar según que atributos o métodos si estos no entraban como parámetro del método. Esto lo pudimos solucionar con la implementación el bucle for que recorría la LinkedList de instrucciones para que pudiésemos usar los métodos de la clase Instruction.

También nos costó entender que isCorrect de la clase Program no recorría todas las instrucciones de la LinkedList y comprobaba que eran correctas sino que comprobaba que para cada REP del programa hubiera un END correspondiente y que de esta manera podríamos saber que era correcto el programa.

Otro aspecto que nos costó ver era el de modificar el código de LogoProgram puesto que como venía dado por el documento de la práctica no pensamos que pudiera estar mal o que se tuviera que modificar.

-Conclusión de la práctica y valoración del resultado final

En cuanto al resultado obtenido finalmente en la práctica, podemos decir que nos parece bastante fiable, hemos probado diferentes tipos de código para ver si realmente detectaba errores y si podía realizar bucles que no estuvieran en la primera línea del programa.

Los tests que hemos realizado nos han mostrado que nuestras clases estaban correctamente implementadas y, por lo tanto, que nuestra solución no daba lugar a error.