

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación
III Curso 2 (noche)

Segundo cuatrimestre de 2020

| | |
|-------------------|---|
| Alumnos y Padrón: | Capón Blanquer, Mateo - 104258 Sabatino, Gonzalo - 104609 Gómez, Darío - 104335 Fernández Caruso, Santiago Pablo - 105267 Diaz Miguez, Abril - 104956 |
|-------------------|---|

Índice

| | |
|---|-----------|
| 1. Introducción..... | 2 |
| 2. Supuestos..... | 2 |
| 3. Diagramas de clase..... | 3 |
| 4. Diagramas de secuencia..... | 6 |
| 5. Diagramas de paquetes..... | 8 |
| 6. Detalles de implementación..... | 9 |
| 6.1. <i>Lapiz, Estado Lapiz y patrón State.....</i> | <i>9</i> |
| 6.2. <i>Bloque Mover, Direccion y el patrón Factory</i> | <i>9</i> |
| 6.3. <i>Observadores, Observados y el patrón MVC.....</i> | <i>10</i> |
| 7. Excepciones..... | 12 |

1. Introducción

El presente informe explica los aspectos más destacables del trabajo práctico número dos de la materia Algoritmos y Programación III de la FIUBA. Se mostrarán diagramas de interés; se explicarán las facetas más importantes de la implementación, especialmente los relacionados con patrones de diseño y arquitectura; y se explicitarán los supuestos y las excepciones del presente trabajo.

Cabe mencionar algunas aclaraciones que se tendrán en cuenta durante el resto del informe:

- Los botones o bloques simples hacen referencia a bloques que no pueden almacenar otros bloques dentro.
- Los bloques complejos pueden (y deben) guardar otros bloques dentro, ya sean simples o complejos.
- En nuestra implementación, un bloque personalizado (uno creado por el usuario) es tratado como un algoritmo a nivel modelo. En general, diremos bloque personalizado si es uno que guardó el usuario, y algoritmo si es aquél mostrado en el panel derecho de la interfaz gráfica, que se ejecutará si el botón correcto es presionado.

2. Supuestos









En el presente trabajo, se realizaron los siguientes supuestos:

- Ningún tipo de secuencia de bloques, ya sea bloque repetir, bloque invertir, bloque personalizado, o el algoritmo a ejecutar, puede ejecutarse y/o guardarse vacío. Debe contener al menos un bloque en su interior.
- El mapa es tratado como uno circular, por lo que cuando el personaje llegue a uno de los cuatro límites del terreno (dados por los atributos estáticos de la clase Posicion) y continúe moviéndose en esa dirección, reaparecerá en el lado opuesto.
- Dos bloques personalizados distintos no pueden tener el mismo nombre.
- Se detalla la operación *invertir* de cada tipo de bloque

| Tipo de bloque | Bloque Ingresado | Acción invertida | Ejemplo |
|------------------------|-------------------------|----------------------------------|---|
| Bloque Simple | Mover Derecha | mueve a izquierda | |
| | Mover Izquierda | mueve a derecha | |
| | Mover Arriba | mueve abajo | |
| | Mover Abajo | mueve arriba | |
| Bloque Complejo | Algoritmo ¹ | invierte cada bloque | Si ingresan: Mover Arriba, Mover Derecha, Subir Lapis, se ejecutará: Mover Abajo, Mover Izquierda, Bajar Lapis |
| | Repetición | invierte el orden de los bloques | Si ingresan: Repetirx2 (Mover Arriba, Mover Derecha, Subir Lapis), se ejecutará: Repetirx2(Subir Lapis, Mover Derecha, Mover Arriba) |
| | Invertir | invierte el orden de los bloques | Si ingresan: Invertir{ Invertir (Mover Arriba, Mover Derecha, Subir Lapis) }, se ejecutará: Bajar Lapis, Mover Izquierda, Mover Abajo |
| | Algoritmo Personalizado | invierte el orden de los bloques | Si el bloque personalizado es: Mover Arriba, Mover Derecha, Subir Lapis, se ejecutará: Subir Lapis, Mover Derecha, Mover Arriba |

3. Diagramas de clases

Aquí se muestra un cuadro en el que se explicitan qué símbolos representan cada tipo de visibilidad. Los mismos se estarán usando en todos los diagramas a continuación.

| Character | Icon for field | Icon for method | Visibility |
|-----------|---|---|-----------------|
| - |  |  | private |
| # |  |  | protected |
| ~ |  |  | package private |
| + |  |  | public |

Cuadro 1: Referencias de la visibilidad, tanto para métodos como para campos

¹ Algoritmo hace referencia a lo ingresado dentro de un invertido simple.

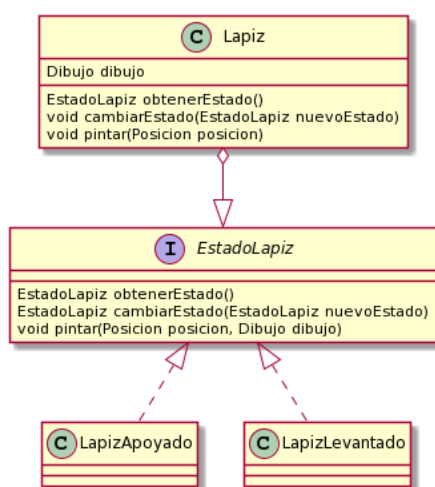


Diagrama 1: Implementación del Patrón State del Lápiz

En este diagrama, mostramos el uso del patrón State para crear las clases Lápiz y EstadoLápiz. Más detalles al respecto se pueden encontrar en el apartado 6.1, dentro de Detalles de Implementación.

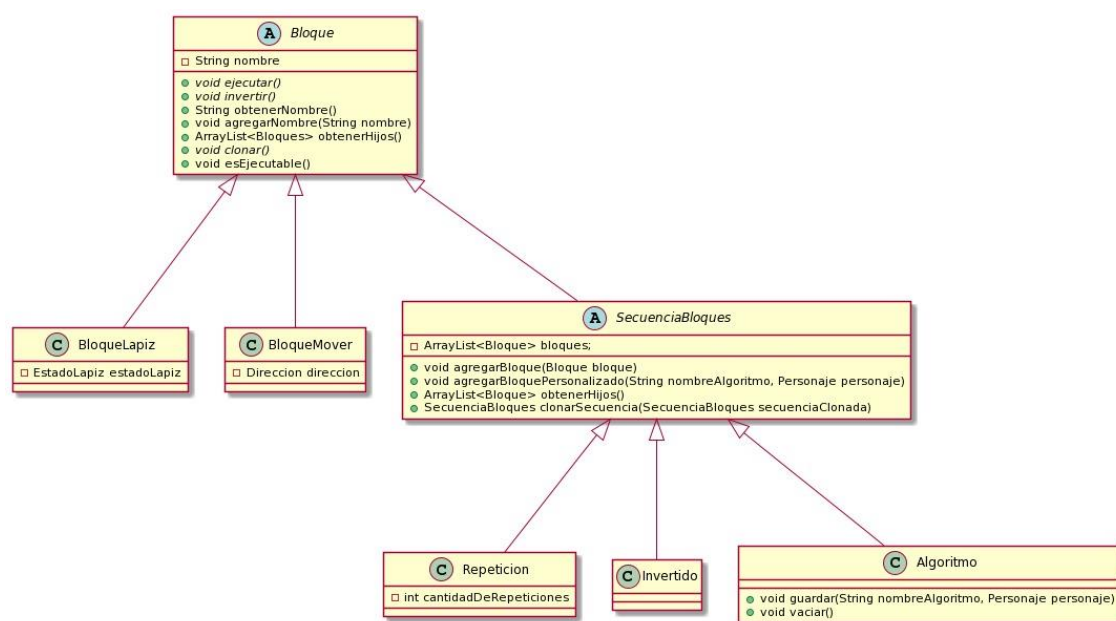


Diagrama 2: Implementación de todos los tipos de Bloque disponibles. Nótese que lo que el usuario guarda como un *BloquePersonalizado*, el código lo trata como un *Algoritmo*.

Aquí, mostramos que hay tres tipos de Bloques, y a la vez tres tipos de SecuenciaBloques. Repeticion, Invertido y Algoritmo son aquellos que se consideran complejos, ya que son capaces de almacenar bloques, además de ser bloques de por sí.

Del BloqueMover, dependiendo de la dirección pasada por parámetro, se podrán crear cuatro tipos de bloques, que se muestran en la interfaz gráfica: BloqueMoverDerecha, BloqueMoverIzquierda, BloqueMoverArriba y BloqueMoverAbajo. Más detalles al respecto pueden encontrarse en el apartado 6.2, dentro de Detalles de Implementación.

Del BloqueLapiz pueden surgir el BloqueLapizApoyado y el BloqueLapizLevantado, que guardan el EstadoLapiz correspondiente. Cómo se crean y ejecutan se puede ver en los diagramas de secuencia 8 y 9.

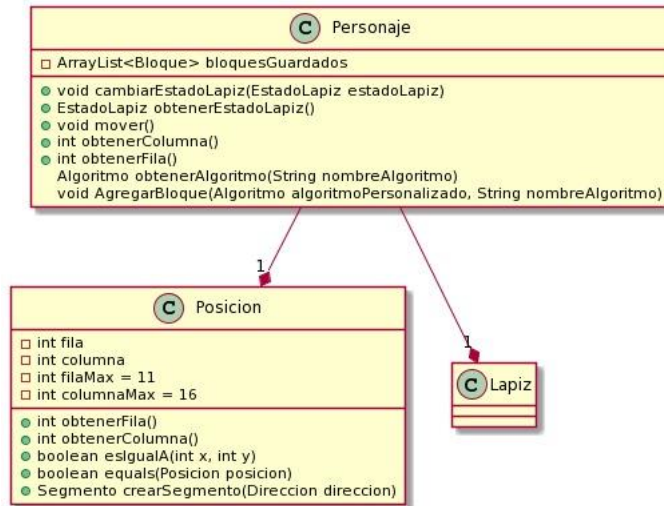


Diagrama 3: Relación entre Personaje, Posicion y Lapiz. Explicación de Lapiz en Diagrama 1.

En este diagrama se puede apreciar que el Personaje es el encargado de guardar los bloques personalizados. A su vez, el mismo posee una posición actual, que será la afectada cuando el usuario desee ejecutar un BloqueMover en el Algoritmo. También es portador de un Lapiz, el cual fue detallado en el diagrama 1.

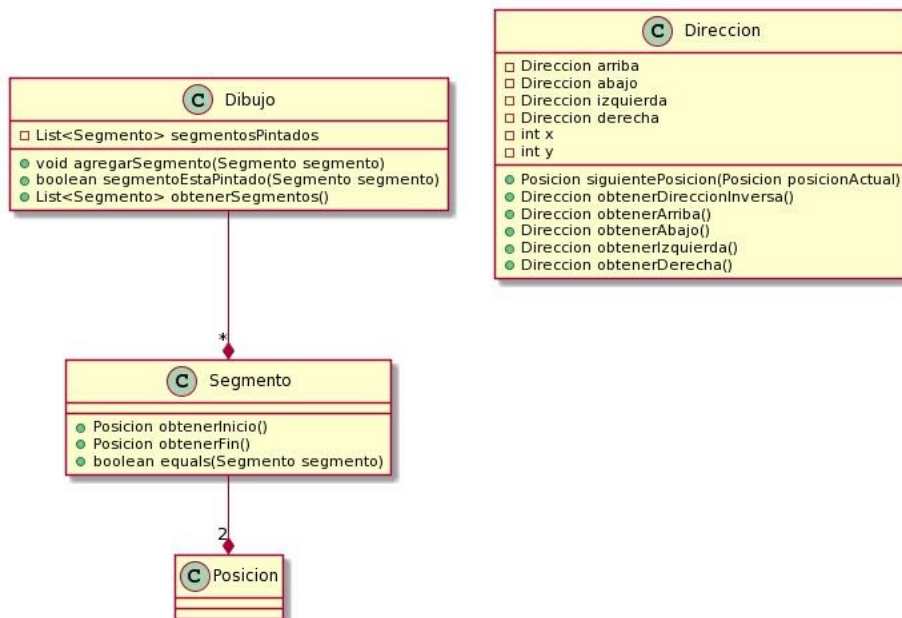


Diagrama 4: Interacciones entre Dibujo, Segmento, Posicion (más detalles en Diagrama 3) y Direccion (más información en Apartado 6.2)

Aquí se muestran todos los métodos de la clase Direccion, que será detallada en el apartado 6.2. De la misma manera, se demuestra la relación entre Dibujo, Segmento y Posición. Cuando

el usuario desee pintar, se le enviará un mensaje al personaje, que se lo delegará al lápiz, que deberá resolverlo. Éste lo hace creando un segmento nuevo, con dos posiciones como atributos, la final y la inicial. Luego, ese segmento nuevo será agregado a la lista de segmentos en el Dibujo, que desencadenará el método `notificarObservadores()`, el cual avisa a la interfaz gráfica que se debe actualizar, porque hay nuevos segmentos pintados.

4. Diagramas de secuencia

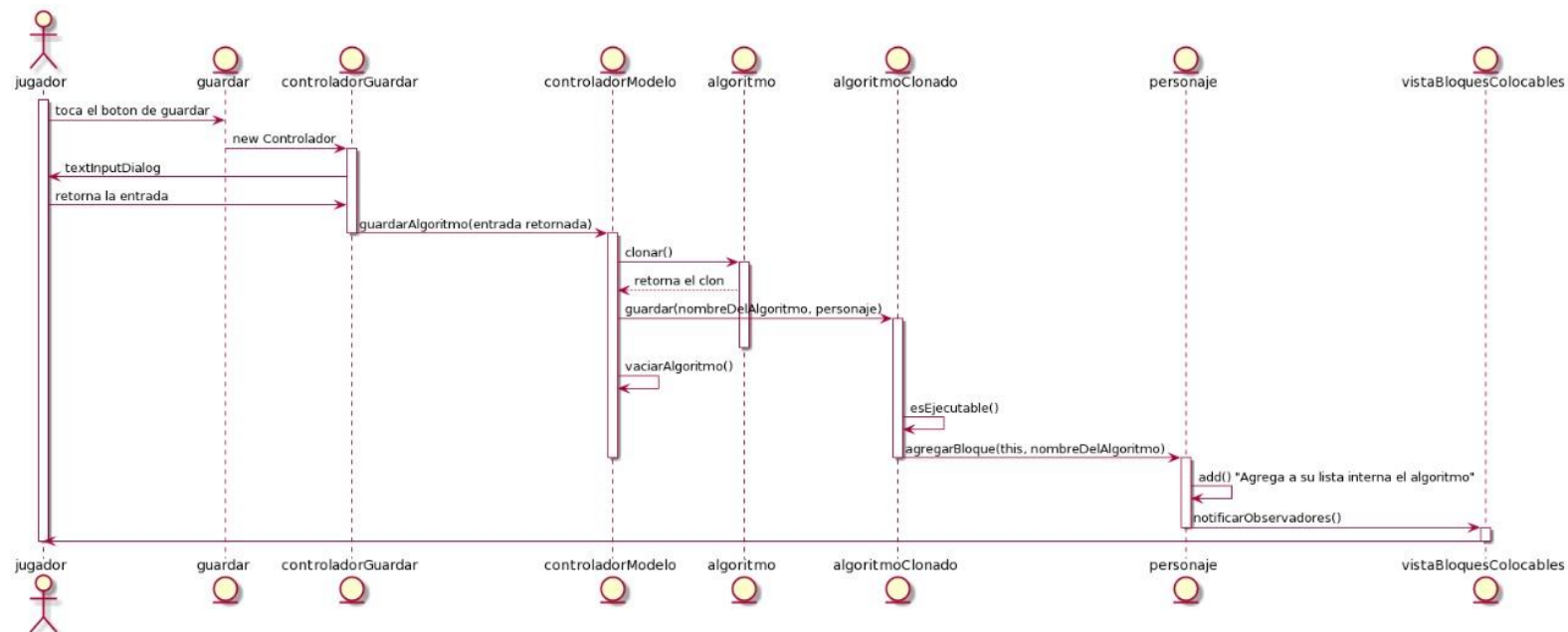


Diagrama 5: Secuencia que explica cómo se guarda un bloque personalizado. Caso Positivo.

Aquí se detalla cómo la vista recibe la solicitud del usuario de guardar el bloque personalizado y cómo lo hace el controlador y el modelo. Éste es el caso positivo de esta interacción, ya que asumimos que hay bloques cargados en el Algoritmo.

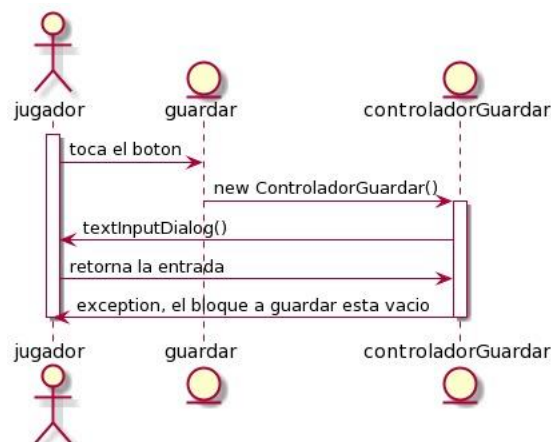


Diagrama 6: Secuencia que explica cómo se guarda un bloque personalizado. Caso Negativo 1.

Es la misma interacción que en el diagrama anterior, el 5, pero aquí el usuario no cargó

previamente bloques al algoritmo, por lo que se lanzó una excepción. La misma fue detallada en la sección 7 Excepciones.

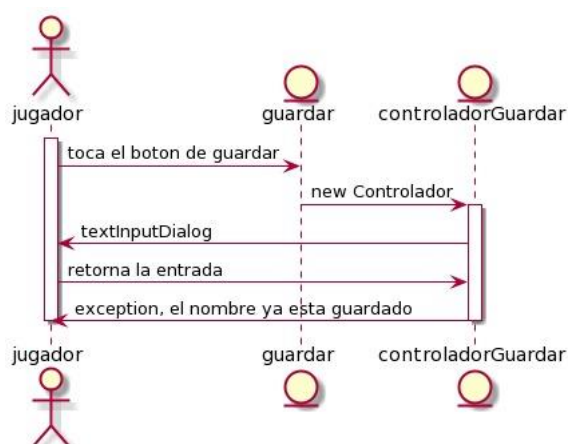


Diagrama 7: Secuencia que explica cómo se guarda un bloque personalizado. Caso Negativo 2.

Es la misma interacción que en el diagrama 5, pero aquí el usuario ingresó un nombre que ya estaba ocupado, por lo que salta una excepción. La misma se detalla en la sección 7 Excepciones.

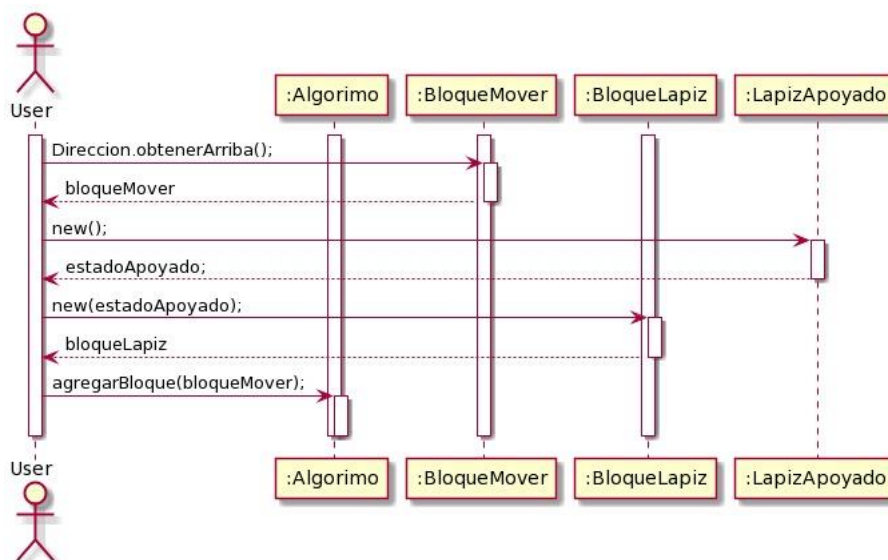


Diagrama 8: Secuencia que se realiza al agregar un BloqueMover y un BloqueApoyarLapiz

En este diagrama se muestra cómo se agregan dos bloques al algoritmo, el BloqueMover y el BloqueApoyarLapiz.

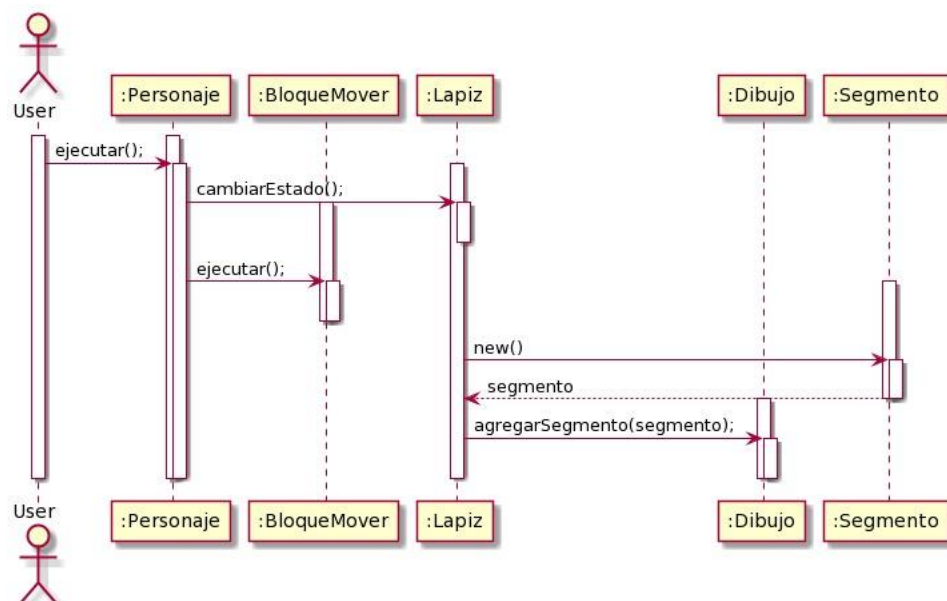


Diagrama 9: Ejecución de los bloques agregados en el diagrama anterior, *BloqueMover* y *BloqueApoyarLapiz*

Este diagrama continúa el 8, y describe cómo se ejecutan esos bloques previamente agregados, y sus respectivas interacciones con el modelo.

5. Diagramas de paquetes

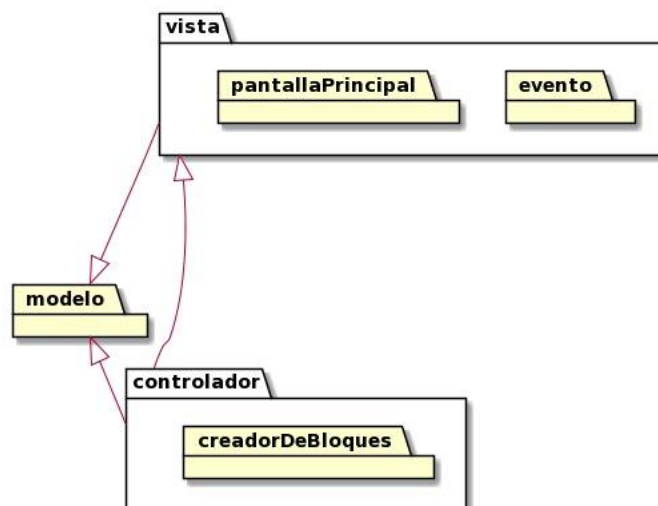


Diagrama 10: Interacciones los diferentes paquetes del programa.

Como se puede apreciar, éstos son los tres paquetes principales que se detallan en el patrón de arquitectura MVC, más al respecto en el apartado 6.3. Dentro de la Vista tenemos dos paquetes más, uno con todo lo necesario para la PantallaPrincipal (donde se juega el juego) y otro para todo el manejo de eventos dentro de la interfaz gráfica (movernos entre pantallas, salir del juego, ver las reglas, etcétera). A su vez, dentro del paquete controlador tenemos una carpeta

donde se guardan los tipos de creadores de Bloques, con sus especificaciones.

6. Detalles de implementación

6.1. Lapiz, Estado Lapiz y el patrón State

El objeto Lapiz tiene dos estados posibles:

- estar levantado, que implica que el personaje, al moverse, no pintará;
- estar apoyado, que le permite al usuario dibujar en la pizarra.

Aunque al principio intentamos implementar dos tipos de lápices distintos y que se fuesen pisando el uno al otro cuando el usuario lo necesitaba, pronto nos dimos cuenta de que el patrón State resolvía el comportamiento deseado de una manera mucho más elegante.

De esta manera, el lápiz siempre es el mismo, pero lo que cambia es su estado, que tendrá su comportamiento particular. El usuario, al interactuar con el programa, podrá cambiar el estado del lápiz y a través del polimorfismo propio del patrón, todos los estados entenderán el mensaje pintar(). El mismo es declarado en la interfaz EstadoLapiz, implementada por LapizApoyado y LapizLevantado tal como se muestra el diagrama de clases correspondiente, véase diagrama 1.

El Lapiz, a su vez, posee como atributo un dibujo, que es el que modificará (o no) según el estado correspondiente. De esta manera, el lápiz le delega a su estado la acción de pintar, detalle que el usuario, desde afuera interactuando con la aplicación, no necesita saber.

6.2. Bloque Mover, Dirección y el patrón Factory

El BloqueMover, como se explicó en los diagramas 2 y 4, según la Dirección pasada por parámetro, es el que genera los diferentes tipos de BloqueMover, con las cuatro direcciones posibles.

La Direccion es una clase que sigue el patrón Factory, en el que solamente genera una dirección arriba, abajo, izquierda y derecha. Esta dirección no es una posición en sí, en realidad es un agregado de las coordenadas de la posición. Entonces, si deseo que mi personaje se mueva una posición hacia la derecha, me pasan su posición actual y le agrego lo necesario a su X e Y.

A continuación, se presenta el código del constructor de Dirección

```
private Direccion(int x, int y){  
    this.x = x;  
    this.y = y;  
}
```

Como se ve, este constructor es privado. Esto es parte del patrón Factory y le prohíbe al usuario crear tantas direcciones como le plazca. La forma de empezar a utilizar una dirección es en realidad, por ejemplo

```
public static Direccion obtenerArriba() {  
    return arriba;  
}
```

Hay métodos similares para Abajo, Derecha e Izquierda. En este caso se llama a *arriba*, que es una dirección creada en un método estático, así

```
static {  
    arriba = new Direccion(0, -1);  
    abajo = new Direccion(0, 1);  
    izquierda = new Direccion(-1, 0);  
    derecha = new Direccion(1, 0);  
}
```

Es posible apreciar que es en este método donde se llama al constructor privado de Direccion.

Ahora se muestra el método siguientePosicion, que devuelve la posición deseada, pasada la actual como parámetro.

```
public Posicion siguientePosicion(Posicion posicionActual){  
    int columna = posicionActual.obtenerColumna() + this.x;  
    int fila = posicionActual.obtenerFila() + this.y;  
  
    return new Posicion(columna, fila);  
}
```

Se puede ver que cuando me pasan la posición que quiero modificar, simplemente le agrego a su columna y a su fila lo necesario. La dirección actual será o derecha, o arriba, o izquierda, o abajo, por lo que modificará solamente la columna o la fila (sumando o restando una unidad).

Ésta es una prueba que se realizó de Dirección, para ilustrar su funcionamiento.

```
@Test  
public void moverConDireccionArribaRetornaLaPosicionCorrecta(){  
    Posicion posicionActual = new Posicion(0, 0);  
    Direccion arriba = Direccion.obtenerArriba();  
    assertTrue(arriba.siguientePosicion(posicionActual).equals(new Posicion(0, -1)));  
}
```

6.3. Observadores, observados y el patrón MVC

El modelo de arquitectura MVC en conjunto con el patrón Observer nos permitió de manera ágil y modularizada conectar la interfaz gráfica del juego sin necesidad de que el modelo la conociese. Esto es de especial utilidad, ya que introduce la posibilidad de crear diferentes interfaces, en diferentes programas, y que el modelo continúe funcionando indistintamente.

La sección Vista del patrón se basa en tres grandes pantallas: la inicial, la primera que aparece cuando se ejecuta el juego; la principal, donde el usuario juega; y la de reglas, que explica cómo jugar, y a la que se puede acceder desde cualquiera de las dos anteriores.

En concreto, se observan el Personaje, el Dibujo y el Algoritmo principal. A través del Personaje, se observan los BloquesPersonalizados guardados. Es importante destacar que el

patrón es utilizado de forma distinta en la vista del Algoritmo, a comparación de las vistas de tanto el Personaje como el Dibujo.

En la primera, dado que el Algoritmo puede ser modificado indirectamente, su controlador es quien advierte al Algoritmo que debe notificar a sus observadores. Tomemos como ejemplo el caso en el que agrego al algoritmo un BloqueInvertir. Cuando se inserten otros bloques en el invertir, se estarán agregando al bloque invertir y se modificará la vista del algoritmo indirectamente.

En contraposición, tanto el Personaje como el Dibujo cambian su estado accediendo a ellos directamente. Por lo tanto, se ejecuta la notificación a los observadores desde los métodos propios de las entidades del modelo.

Por otra parte, en esta sección manejamos los eventos dentro de la aplicación, como es el presionar el botón de Reiniciar Juego. Mostramos el código que describe la acción del botón Reiniciar Juego una vez presionado. Como se puede ver, se vuelve al estado inicial, cuando el jugador apenas había entrado a la pantalla principal.

```
@Override
public void handle(ActionEvent actionEvent) {
    pantallaPrincipal.inicializar(stage);
    stage.setScene(scene);
    stage.setFullScreen(true);
}
```

La sección Controlador conoce al modelo e interactúa con la vista. El principal controlador creado fue el ControladorModelo, que es el que conoce al algoritmo a ejecutar, al dibujo y al personaje. Creamos también otros para todos los botones con acciones imprescindibles, como lo son el guardar un algoritmo o activar un botón simple o complejo. La mayoría de estos últimos interactúan de una manera u otra con el ControladorModelo para comunicarse con el modelo.

Ejemplos de cómo el ControladorModelo interactúa con el algoritmo, ya sea para vaciar el algoritmo actual, o cómo se ejecuta la secuencia cuando el usuario presionó el botón Ejecutar.

```
public void vaciarAlgoritmo(){
    algoritmo.vaciar();
    secuenciasAnidadas.clear();
    secuenciasAnidadas.push(algoritmo);
    algoritmo.notificarObservadores();
    personaje.notificarObservadores();
}
```

```
public void ejecutar() {
    algoritmo.ejecutar(personaje);
}
```

El modelo, independiente de la interfaz gráfica, simplemente tiene la opción de guardar observadores, a los que notificará cuando realice una acción que cambie su comportamiento.

Por ejemplo, al guardar un algoritmo personalizado, vaciar un algoritmo, agregar un segmento pintado al dibujo, etc.

Ejemplo de cómo notificar a los observadores. En este método, se quiere guardar un algoritmo personalizado. En caso de que ese nombre ya esté siendo utilizado en otro algoritmo personalizado, saltará una excepción (se detalla más acerca de la misma en el apartado 7 Excepciones). Si es un nombre disponible, se le agrega el nombre al algoritmo, se lo guarda y se notifica a los observadores.

```
public void agregarBloque(Algoritmo algoritmoPersonalizado, String nombreAlgoritmo) throws
BloquePersonalizadoYaExisteException{

    if (bloquesGuardados.stream().anyMatch(bloque -> bloque.obtenerNombre().equals(nombreAlgoritmo)))
        throw new BloquePersonalizadoYaExisteException("El nombre elegido no esta disponible.");

    algoritmoPersonalizado.agregarNombre(nombreAlgoritmo);
    bloquesGuardados.add(algoritmoPersonalizado);
    notificarObservadores();
}
```

7. Excepciones

Las excepciones creadas para este trabajo fueron las siguientes:

AlgoritmoVacioExcepcion Se lanza cuando el usuario intenta ejecutar una secuencia de bloques vacía; cuando se desea guardar un bloque personalizado, también vacío; y cuando se quiere ejecutar una secuencia que tiene otra secuencia de bloques vacía incluida.

BloquePersonalizadoNoExisteExcepcion Se lanza cuando el usuario quiere buscar un bloque personalizado con un nombre que nunca se registró. Nota: esta excepción no puede lanzarse desde la interfaz gráfica, ya que el usuario no ingresa el nombre del algoritmo que quiere. Sin embargo, desde el código fuente es posible ingresar un nombre incorrecto al método que busca los bloques personalizados, por lo que la excepción debió tenerse en cuenta.

BloquePersonalizadoYaExisteExcepcion Se lanza cuando el usuario quiere agregar un nuevo bloque personalizado con un nombre que ya le pertenece a otro bloque personalizado. En ese caso, deberá ingresarse otro nombre.

A saber, se utilizó la excepción de Java `IllegalArgumentException` si se desea crear un `Repeticion` con una cantidad de repeticiones menor o igual a cero.