FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Curso 1
Segundo cuatrimestre de 2020

Grupo T10		
Nombre	**Padrón**	**Mail**
Bergman, Guido	104030	gbergman@fi.uba.ar
Fábregas, Camilo	103740	cfabregas@fi.uba.ar
Delgado, Nahuel	104078	ndelgado@fi.uba.ar
de Montmollin, Magdalena	99331	mdemontmollin@fi.uba.ar

Índice

1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar una aplicación, de manera grupal, que permite aprender los conceptos básicos de la programación de manera didáctica y visual, en Java utilizando la programación orientada a objetos y las técnicas de TDD e integración continua. La interfaz visual fue relizada en JavaFX. Para la ejecución del programa se necesita una pantalla con límites visuales mínimos de 1150×600.

2. Supuestos

Se supuso que el personaje no tiene restricciones espaciales para moverse, es decir, puede moverse en cualquier dirección tantas veces quiera.

Un lápiz puede estar arriba o abajo, no tiene estados intermedios.

Se puede subir o bajar el lápiz 2 veces. La segunda no produce cambios.

Suponemos que al invertir un bloque repetición, los bloques interiores se recorren en el orden inverso y se invierte cada uno de ellos.

Al invertir un algoritmo personalizado, se lo recorre secuencialmente invirtiendo cada uno de los bloques que contiene.

Un bloque personalizado puede llamarse como desee el usuario, toda combinación de teclas alfanumérica es válida. Además, que el usuario guarde más de un algoritmo con el mismo nombre no se considera un problema.

3. Diagramas de clase

El programa se separó en tres diagramas de clases para facilitar la lectura de ellos. En las figuras 1, 2 y 3 se pueden observar los diagramas mencionados.

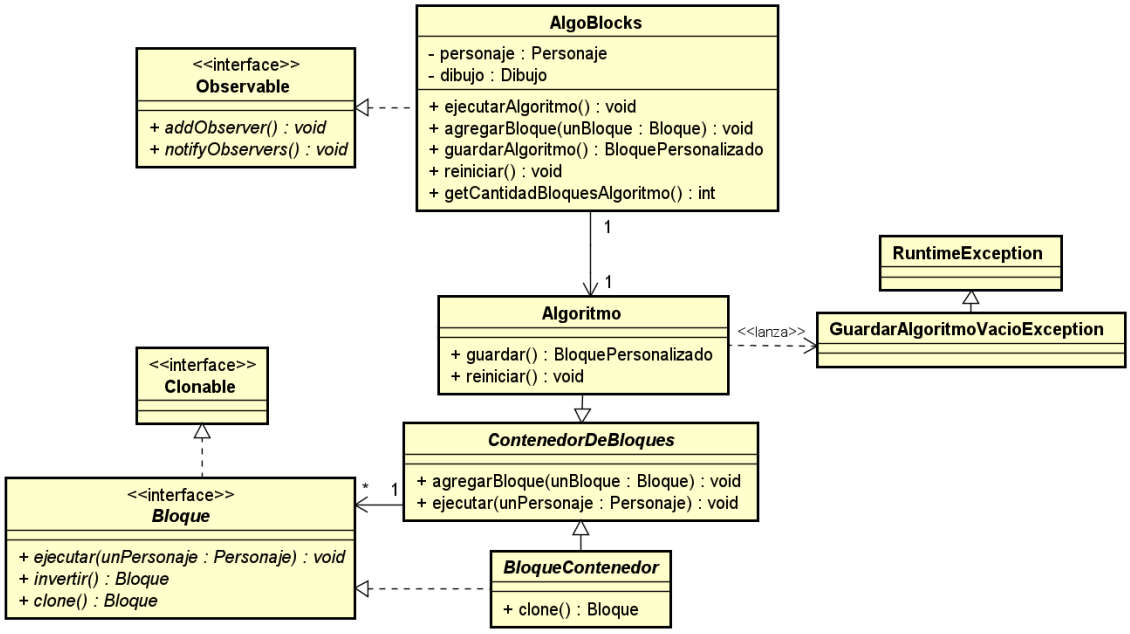

Figura 1: Diagrama de Clases - Algoritmo

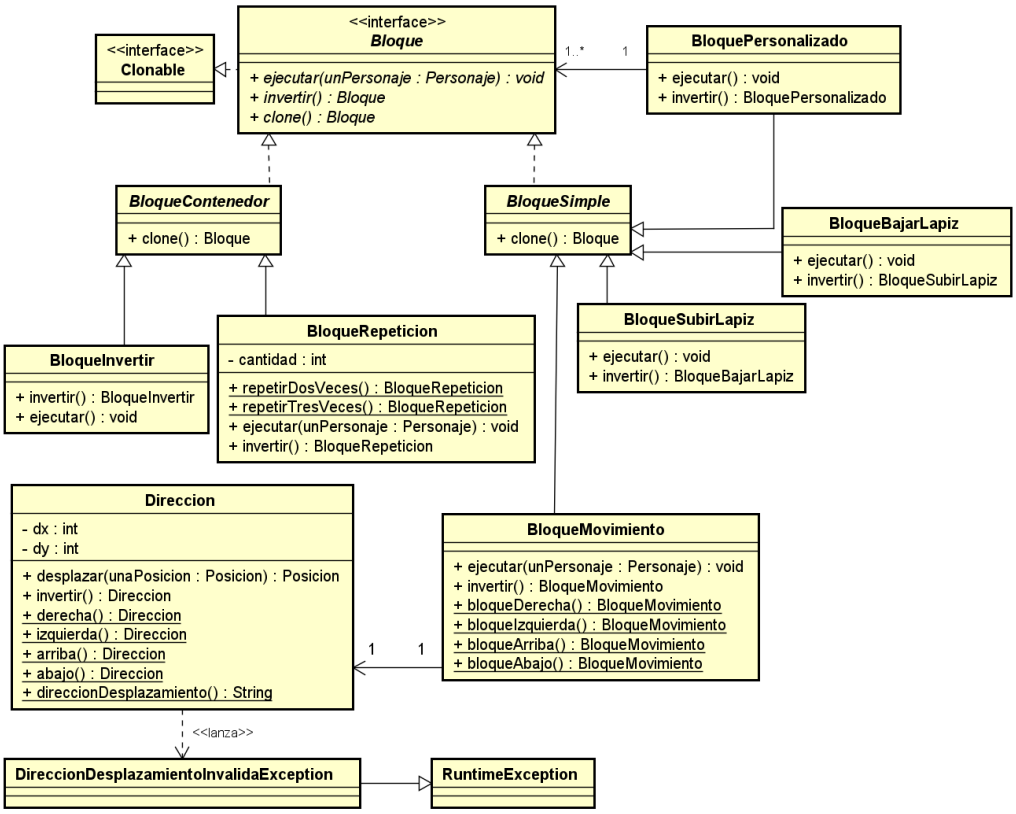

Figura 2: Diagrama de Clases - Bloque

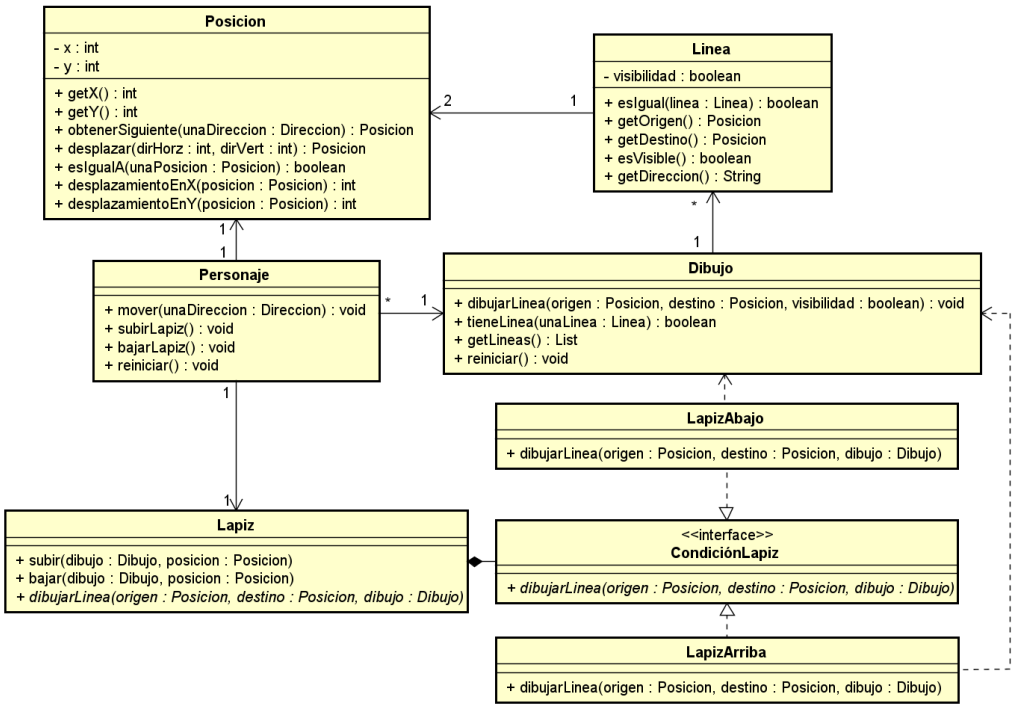

Figura 3: Diagrama de Clases - Personaje

4. Diagramas de secuencia

Los diagramas de secuencia están dirigidos a la ejecución de bloques y al manejo de bloques compuestos.

El primer diagrama representa la ejecución de un algoritmo con solo un bloque de movimiento hacia la derecha con el lápiz abajo. El diagrama se divide en dos, la primera parte muestra el desplazamiento del personaje y la segunda muestra como se dibuja la línea. Si el personaje tuviese el lápiz arriba el método `dibujarLinea(Posicion posVieja, Posicion posNueva, boolean visibilidad)` de **CondicionLapiz** recibiría *false* en vez de *true*. La misma lógica se extiende a la creación de la línea.

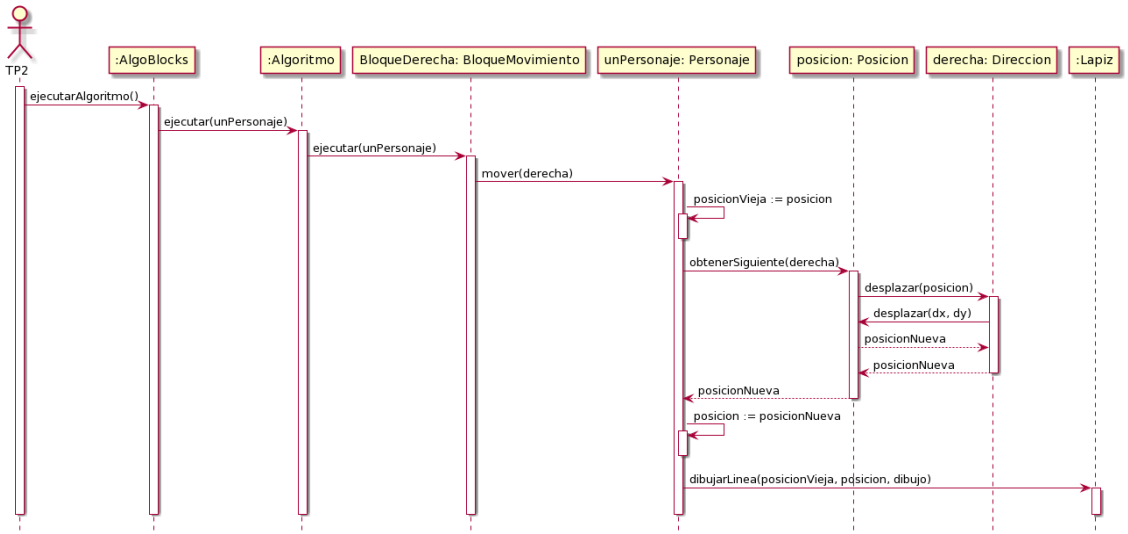

Figura 4: Diagrama de Secuencia - Ejecutar un bloque movimiento a la derecha con lápiz abajo(parte 1).

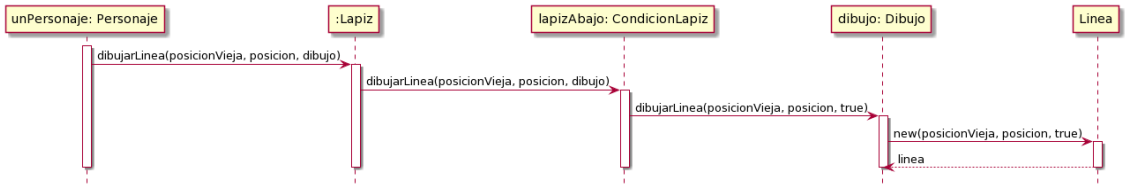

Figura 5: Diagrama de Secuencia - Ejecutar un bloque movimiento a la derecha con lápiz abajo(parte 2).

El siguiente diagrama muestra como se ejecuta un bloque repetición de tres veces con bloques dentro.

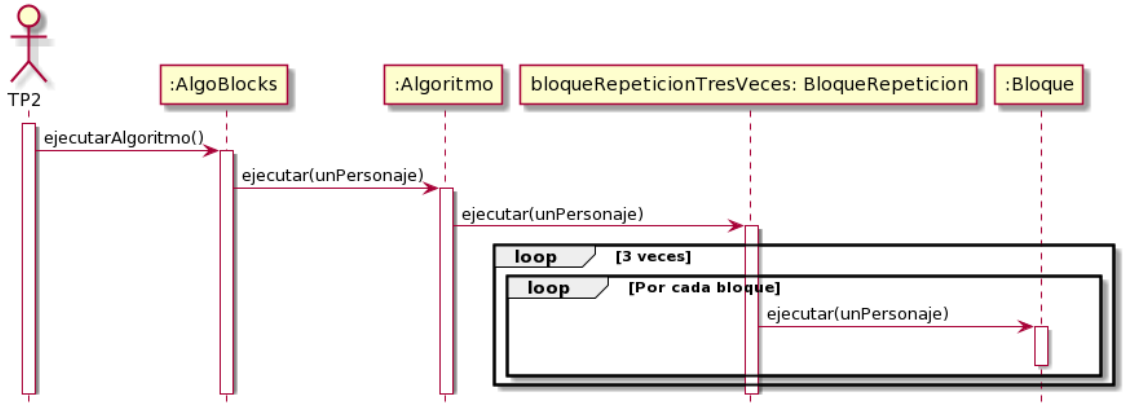

Figura 6: Diagrama de Secuencia - Ejecutar un bloque repetición de tres veces.

En la figura 7 se observa como, para guardar el algoritmo actual, se crea una copia de cada bloque agregado hasta el momento y las almacena en un nuevo bloque.

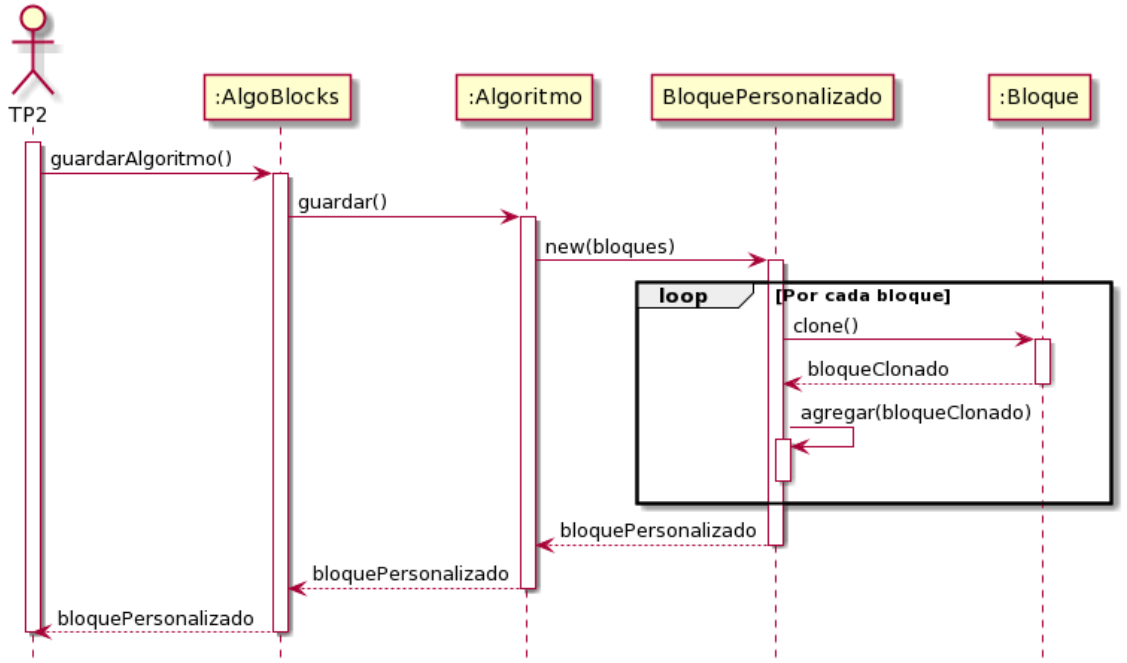

Figura 7: Diagrama de Secuencia - Guardar algoritmo personalizado.

El diagrama de secuencia de ejecutar un **BloqueInversion** (Figura: 8) muestra como se invierte cada bloque y luego se ejecuta su resultado.

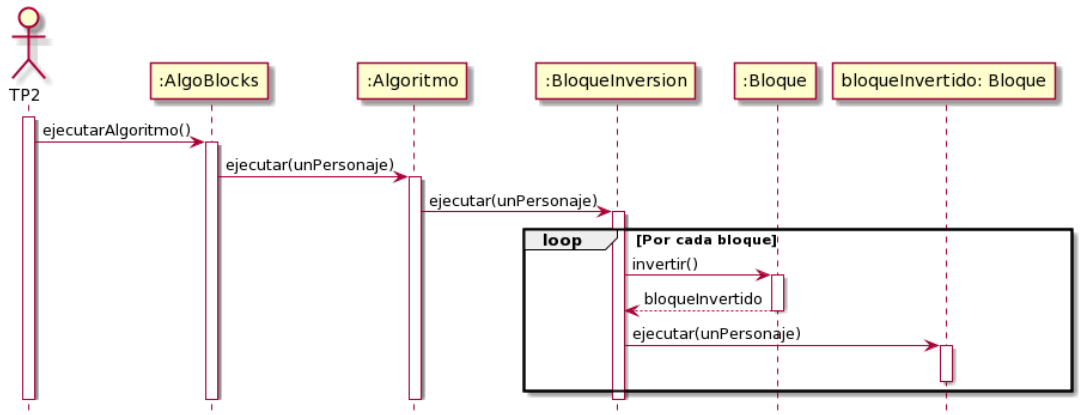

Figura 8: Diagrama de Secuencia - Ejecutar bloque inversión.

Las figuras 9 y 10 hacen referencia a casos que pueden ocurrir como iteraciones del ciclo del diagrama de secuencia anterior (Figura: 8). El primero muestra el caso de invertir un bloque repetición, que como resultado se invertiría el orden y cada bloque dentro del mismo.
El segundo representa la inversión de un bloque inversión.

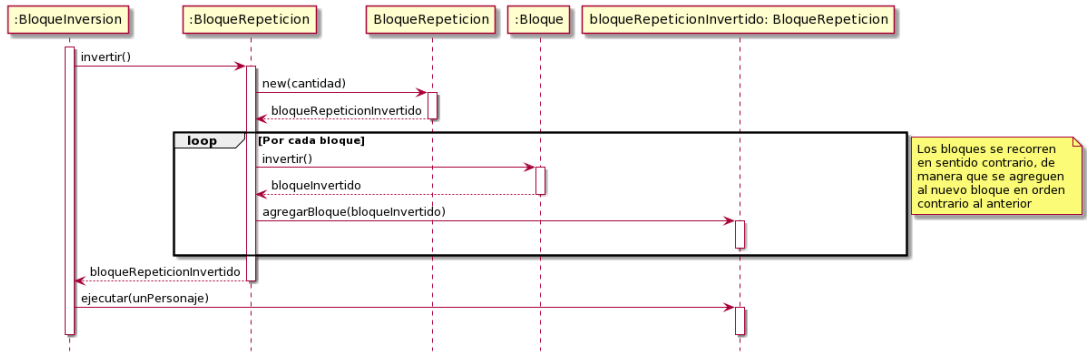

Figura 9: Diagrama de Secuencia - Invertir un bloque repetición.

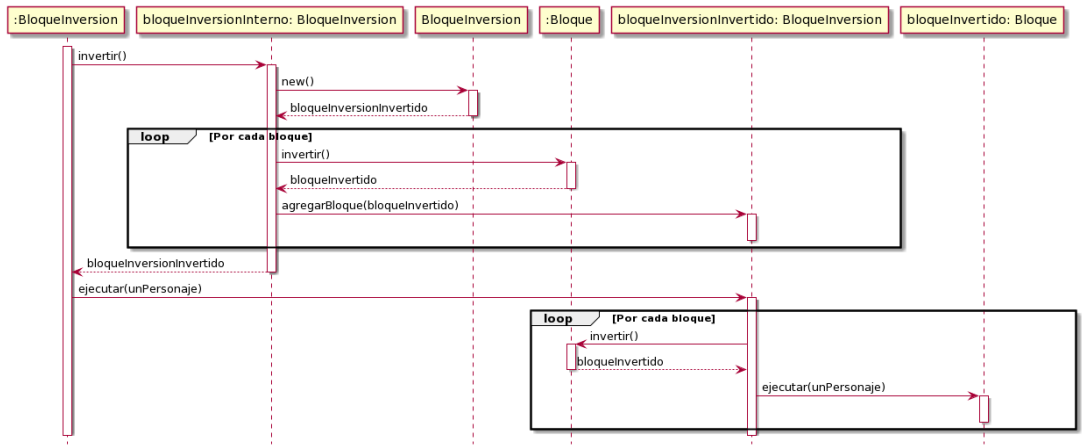

Figura 10: Diagrama de Secuencia - Invertir un bloque inversión.

5. Diagramas de paquetes

En la figura 11 se puede observar el acoplamiento entre los paquetes del trabajo.

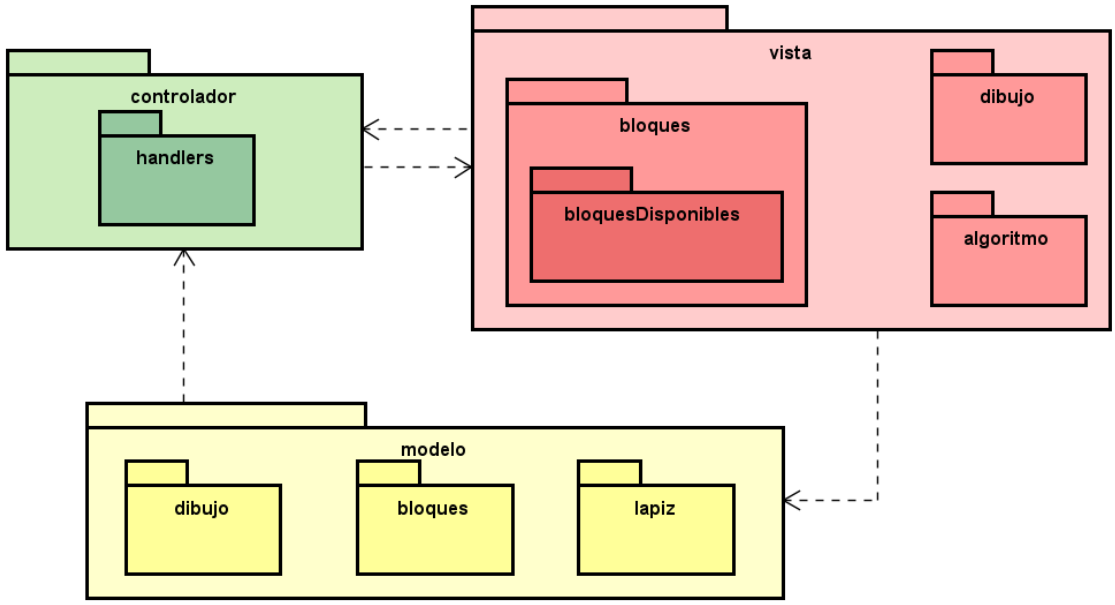

Figura 11: Diagrama de Paquetes

6. Diagramas de estado

La única entidad del modelo en la que se utiliza el patrón de diseño *State* es **Lapiz**.

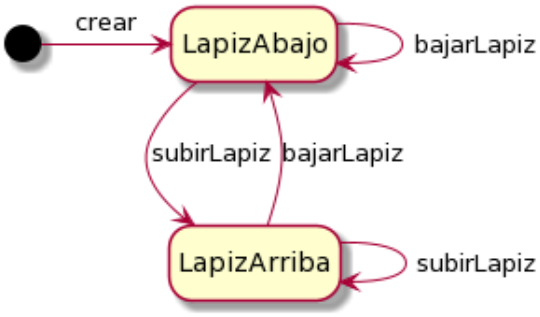

Figura 12: Diagrama de Estado - Lápiz

7. Detalles de implementación

Aunque resulta claro al ver el diagrama de paquetes de la sección 5 se destaca el uso del patrón Modelo-Vista-Controlador utilizado para la implementación de la aplicación.

7.1. Detalles del Modelo

A la hora de implementar el modelo se plantearon alternativas sobre como debería comportarse el programa ante ciertas situaciones. En esta sección se mencionan las más relevantes.

7.1.1. Condición del lápiz y su relación con el dibujo

No se considera un problema que un **Lapiz** reciba el mensaje `bajar()` si su condición es **LapizAbajo**. Similarmente, no es un problema que reciba `subir()` si su condición es **LapizArriba**. Además, de llamar al método `dibujarLinea(origen: Posicion, destino: Posicion, dibujo: Dibujo)` se crea una instancia de la clase **Linea** en el objeto `dibujo` cuya visibilidad depende de la condición del lápiz. Si esta arriba la línea creada es invisible, caso contrario, es visible.

Al cambiar la condición de un **Lapiz** se dibuja, en el dibujo, una línea con una misma posición como origen y destino, cuya visibilidad depende de la nueva condición del lápiz. Esta es consistente a lo comentado en el párrafo anterior.

Se puede ver con facilidad que la interfaz **CondicionLapiz** utiliza el patrón de diseño *State* siendo los dos posibles estados del lápiz **LapizArriba** y **LapizAbajo**.

7.1.2. Ejecución del Algoritmo

En **AlgoBlocks** tras ejecutar el **Algoritmo**, este mantiene su secuencia de Bloques, no los elimina. Y de volver a ejecutarlo, el personaje se desplaza desde su última posición y las líneas se agregan al mismo dibujo.

7.1.3. Contenedor de Bloques

Se implementó una clase abstracta llamada *ContenedorDeBloques* de la cual heredan las clases **Algoritmo**, **BloqueInvertir** y **BloqueRepeticion** para evitar la repetición de código al encontrar comportamientos similares en ellas.

7.1.4. Bloque Personalizado

Para guardar el algoritmo actual en una instancia de **BloquePersonalizado** se decidió implementar, para los bloques, la función `clone()`, la cual devuelve una copia del bloque. Es decir, un **BloquePersonalizado** almacena copias de los bloques que hay, al momento de llamar al método `guardarAlgoritmo()` en el algoritmo. Esto fue necesario para evitar que de agregarse bloques a un **BloqueContenedor** previamente almacenado en un **BloquePersonalizado**, éste se vea modificado.

7.1.5. AlgoBlocks

AlgoBlocks fue implementado utilizando el patrón *Facade*. Esto se puede corroborar con los diagramas de clase de la sección 3, y los de secuencia de la 4 donde se evidencia que **AlgoBlocks** no es más que una fachada entre el usuario y las distintas clases que componen al modelo.

7.2. Detalles de la Interfaz

También se tomaron decisiones a la hora de implementar la aplicación con la que interactúa el usuario. Aquí se comentan las más importantes.

La animación del algoritmo ingresado por el usuario siempre realizará los movimientos indicados en él. Incluso cuando esto genere que el personaje salga de los límites visibles de su región.

Al usuario no se le permite ejecutar dos veces el algoritmo, si antes reiniciarlo. De todos modos tras ejecutarlo, sigue teniendo la opción de guardarlo como un nuevo bloque.

Para que el botón de guardar el algoritmo actual aparezca cuando el algoritmo no está vacío, se utilizó un patrón *observer*, que notifica a sus observadores cuando se llama al método `agregarBloque(Bloque unBloque)` de **AlgoBlocks** y a `reiniciar()`. De manera similar, **LayoutAlgoritmo** observa a la instancia de **AlgoBlocks** para reiniciarse cuando el algoritmo vuelve a estar vacío.

8. Excepciones

- **DireccionDesplazamientoInvalidaException** Surge al solicitar la dirección de un desplazamiento, que tiene valores no nulos en ambos ejes.
- **GuardarAlgoritmoVacioException** Surge al querer guardar un algoritmo personalizado cuando no hay bloques.