FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Curso 2
Segundo cuatrimestre de 2020

Alumno 1 :	Araujo, Luis
Número de padrón: Email:	102112 laraujo@fi.uba.ar
Alumno 2 :	Arroyo, Sebastian
Número de padrón: Email:	100728 larroyo@fi.uba.ar
Alumno 3 :	San Martin, Nicolas
Número de padrón: Email:	104320 nsanmartin@fi.uba.ar
Alumno 4 :	Branko Tintilay Tacacho, Ivan
Número de padrón: Email:	102479 btintilay@fi.uba.ar

Índice

1. Introducción

En el presente informe se documentará la implementación del trabajo práctico 2 de la materia Algoritmos y Programación III, el cual consiste en desarrollar un juego orientado al aprendizaje de los conceptos básicos de la programación, mediante el armado de algoritmos usando una serie de bloques con distintas funcionalidades que se verán reflejadas a través de un dibujo en el tablero. El lenguaje de programación que se utilizó es Java, siguiendo el paradigma de la Programación Orientada a Objetos (POO).

2. Supuestos

En esta sección se explicitarán aquellas características del Juego que consideraremos esenciales para el desarrollo del programa, mismas las cuales no se encuentran cubiertas por el enunciado del trabajo o mismas las cuáles nosotros hemos decidido implementar para realizar un trabajo acorde a lo especificado.

1. En el caso en que el usuario quiera borrar el dibujo que estaba realizando, le permitimos reiniciar tanto el dibujo como el personaje a su posición inicial, como si no hubiese estado utilizando previamente la aplicación, mediante un botón que lleva el icono de una goma de borrar.

2. En el caso en que el usuario quiera borrar los bloques que seleccionó para ejecutar en la zona de Algoritmo, le permitimos eliminar todos los bloques seleccionados, como si no hubiese estado utilizando previamente la aplicación mediante un botón que lleva el icono de un tacho de basura.

3. El personaje podrá salirse de los limites del dibujo mediante la ejecución de bloques de movimiento, cuando este cruce un borde, simplemente no se reflejara en pantalla ni su existencia, ni su dibujo, hasta que vuelva a estar presente dentro del dibujo observado por el usuario.

4. El personaje puede volver a estar presente en la pantalla observada de la misma manera en que este salió de la misma utilizando bloques de movimiento o con el botón que reinicia el dibujo y al personaje (con icono de goma de borrar).

5. El Bloque Invertir podrá invertir la funcionalidad de todos los bloques incluyendo, bloques simples (bloques de movimiento de personaje y bloques que cambian la posición del lápiz) y bloques contenedores (bloques algoritmo, bloques repetidores dobles y triples y también otros bloques inversores). Para todos los bloques contenedores, se invertirá la funcionalidad de los bloques simples contenidos dentro de ellos.

6. El usuario podrá elegir si agregarle bloques directamente al sector algoritmo o a algún bloque contenedor. Para darle libertad de elegir, el usuario simplemente deberá hacerle click al bloque contenedor al cuál le quiera agregar bloques o click al primer icono dentro del sector algoritmo para volver a agregarle bloques a este.

3. Modelo de dominio

En cuanto al diseño implementado, se consideró como clases principales a los sectores, SectorAlgoritmo, SectorDibujo y SectorBloques.

El SectorAlgoritmo se encargará de ejecutar los bloques que le fueron agregados, instruyéndole al personaje con estos mismos, las acciones que debe llevar adelante. Estas acciones pueden ser de movimiento o de uso del lápiz, herramienta con la cuál podrá representar trazos en el SectorDibujo (solamente si el lápiz esta hacia abajo). También se pueden guardar los bloques agregados al SectorAlgoritmo en bloques algoritmos personalizados, los cuáles se almacenaran para usarse de la misma que los otros bloques dentro del SectorBloques.

4. Diagramas de Clase

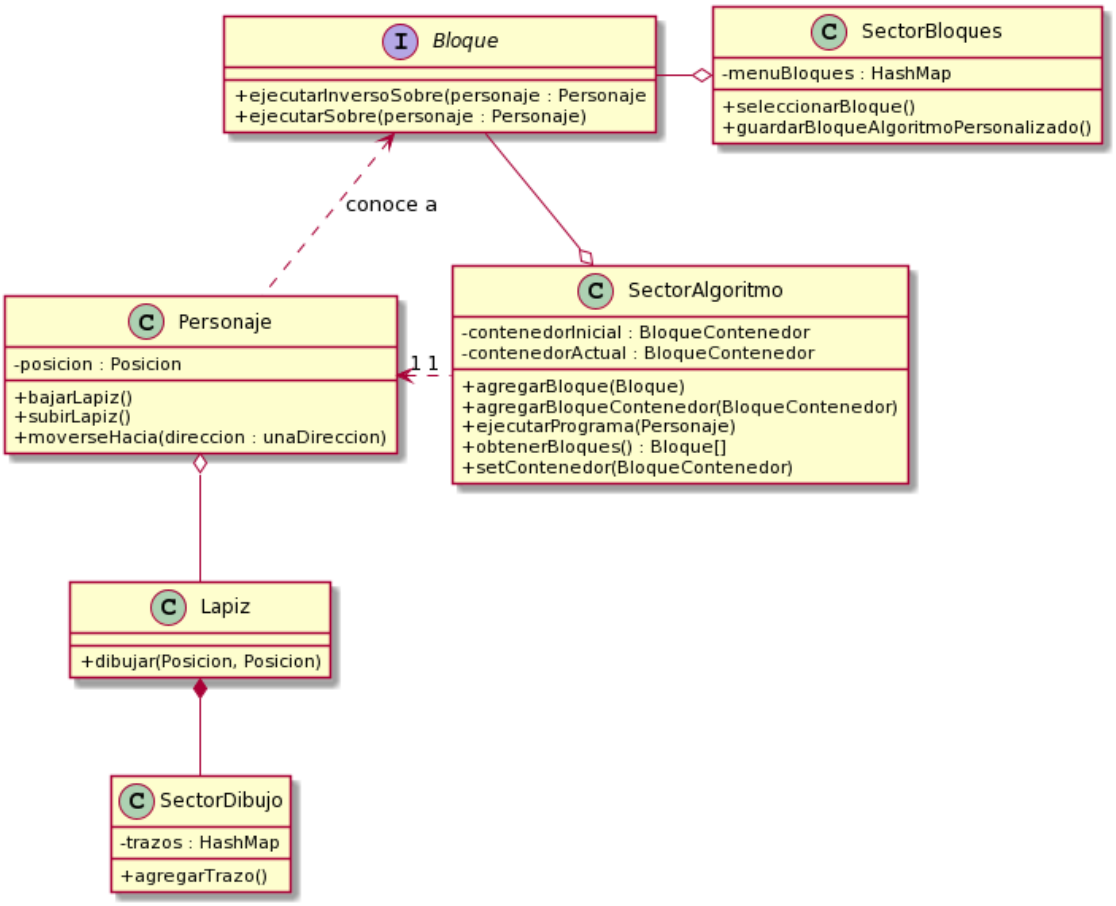

Figura 1: Diagrama sobre la relación de los sectores y sus relaciones con otras clases.

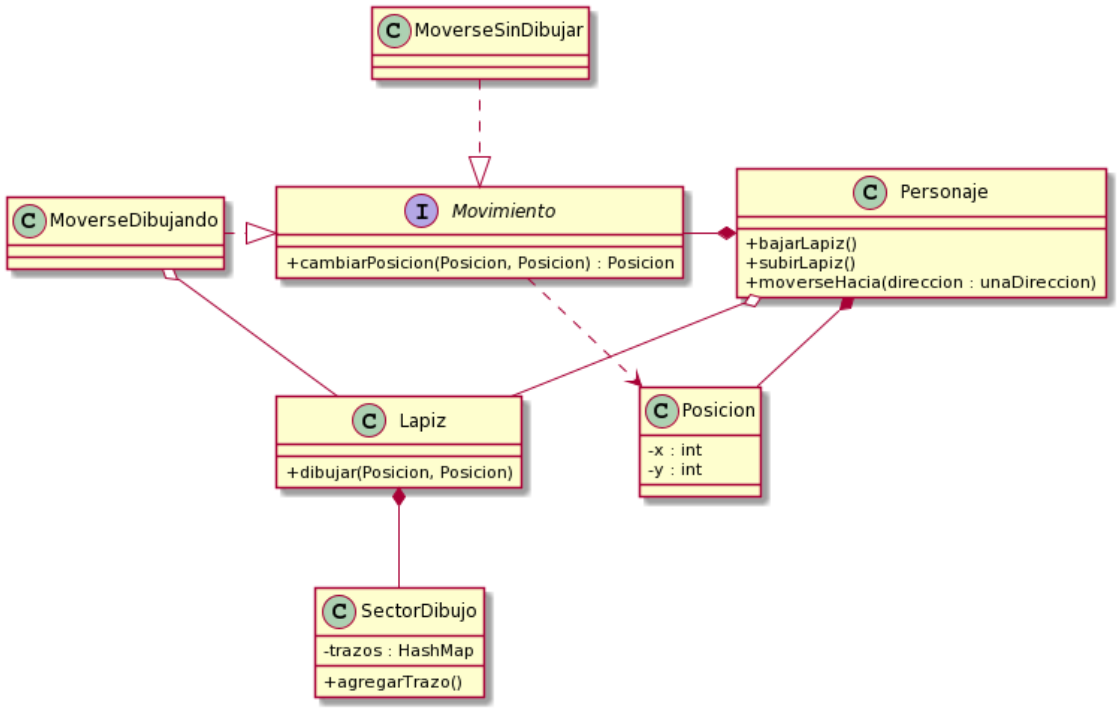

Figura 2: Diagrama sobre la relación del personaje y su forma de dibujar.

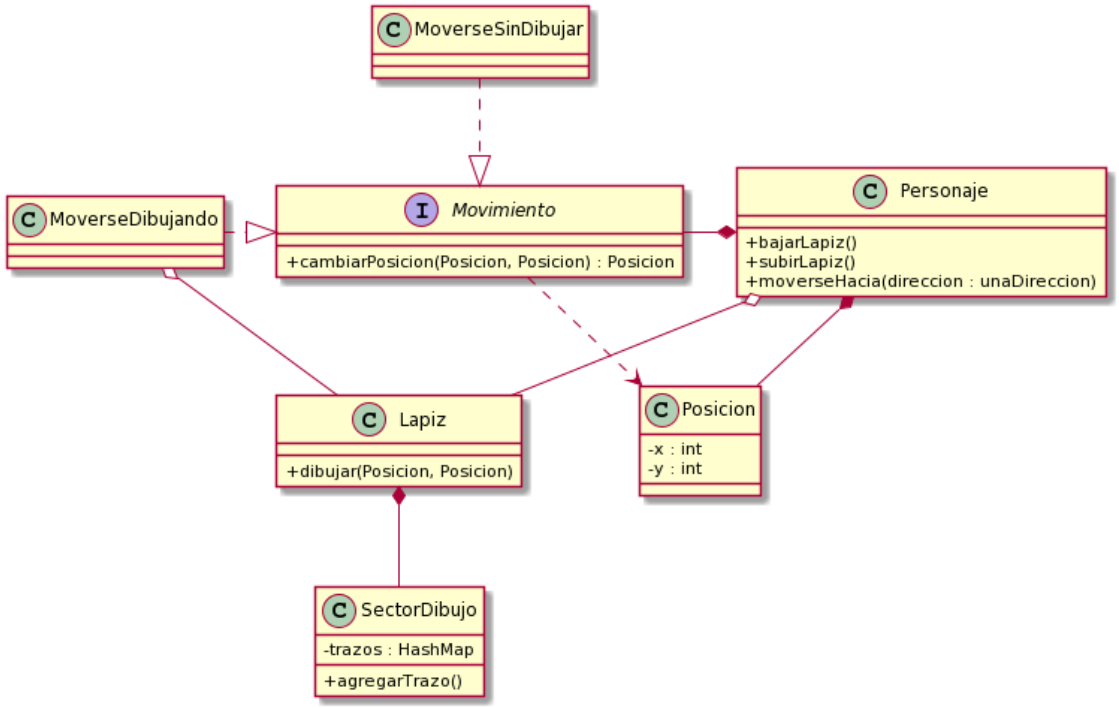

Figura 3: Diagrama sobre la relación del personaje y su movimiento.

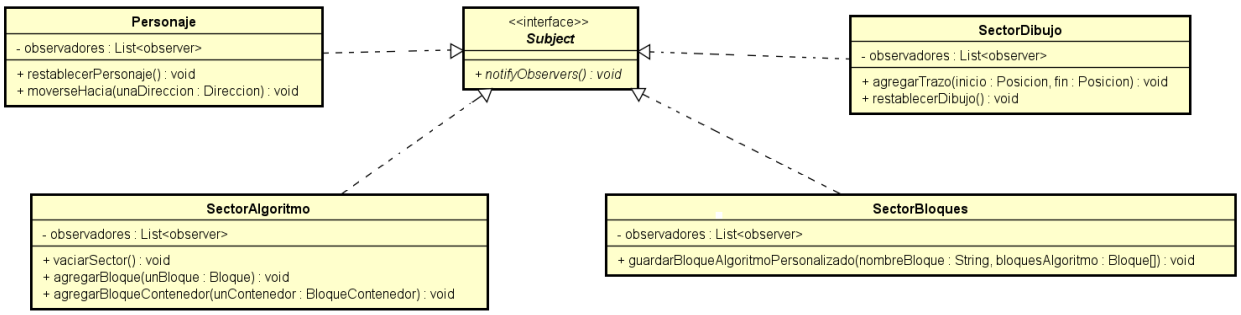

Figura 4: Diagrama sobre la relación de la interfaz Subject con las clases que la implementan.

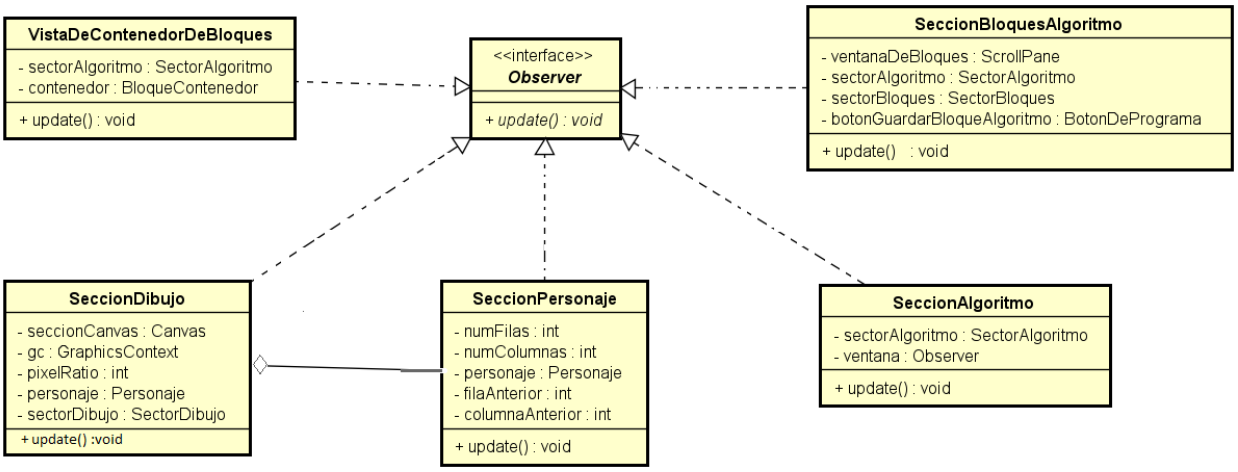

Figura 5: Diagrama sobre la relación de la interfaz Observer con las clases que la implementan.

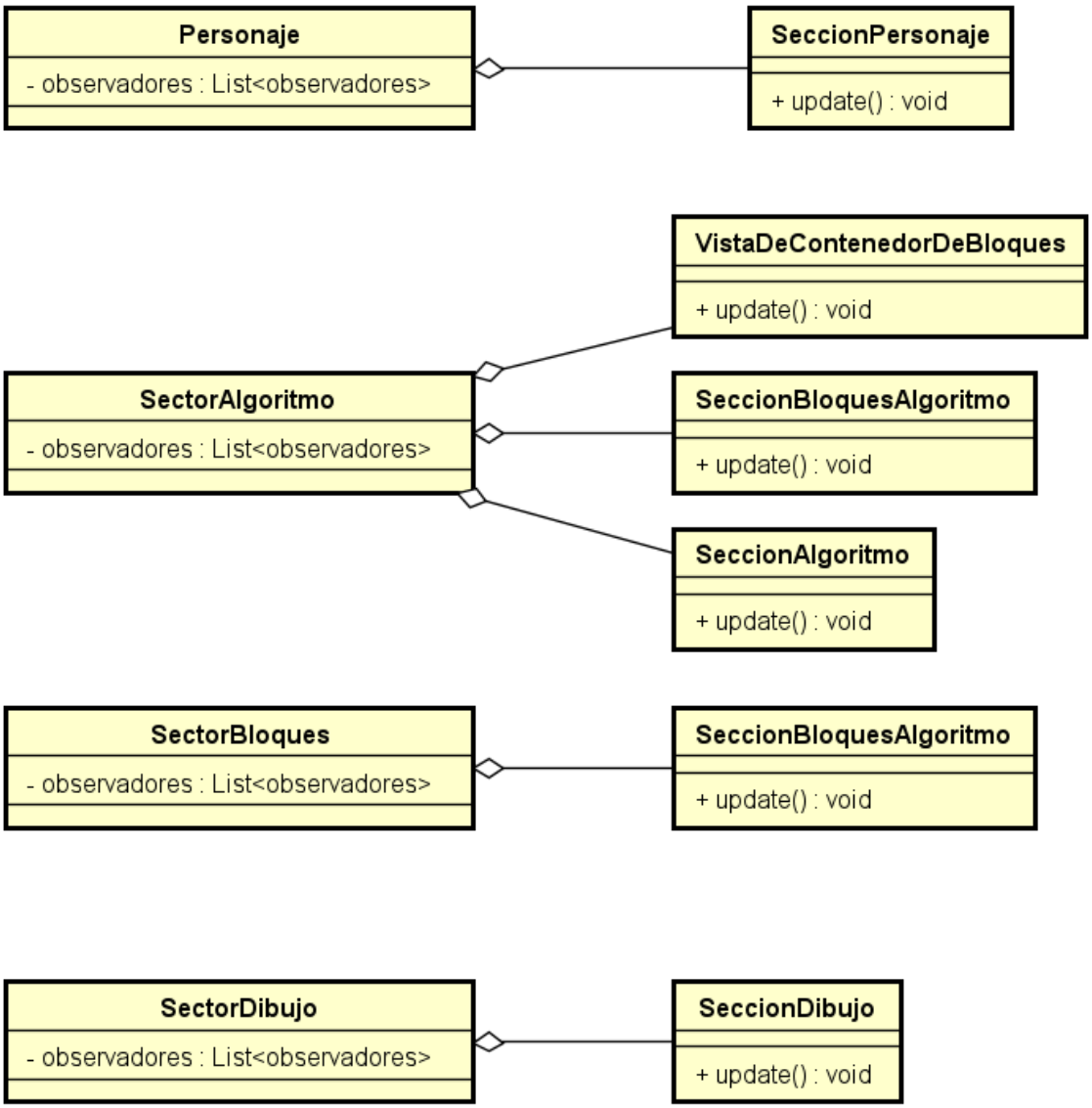

Figura 6: Diagrama sobre la relación de cada clase sujeto con sus clases observadoras.

5. Diagramas de Secuencia

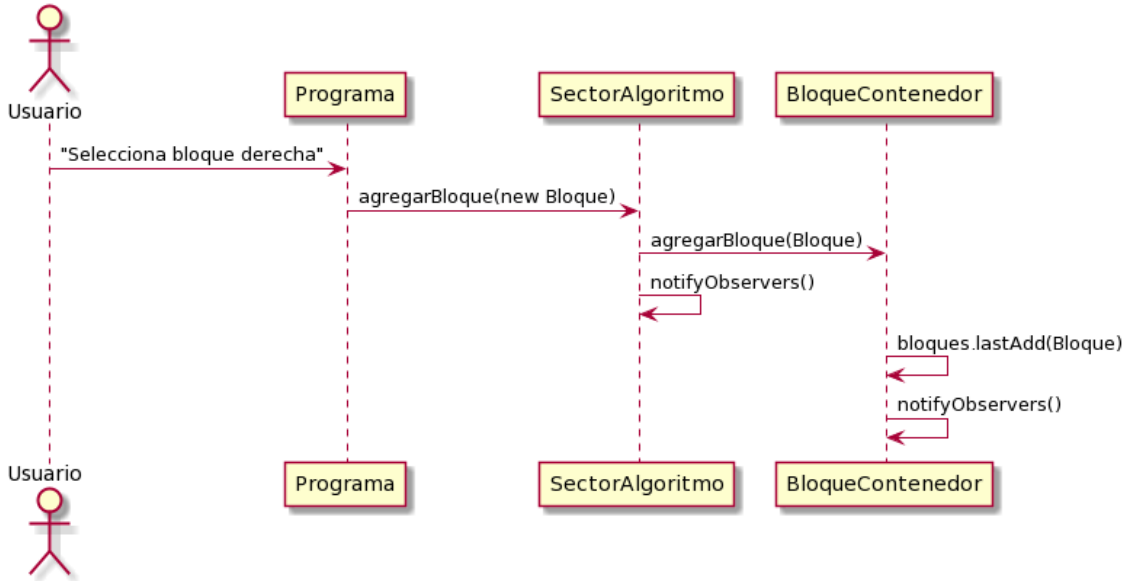

Figura 7: Diagrama sobre la selección por parte de un usuario de un bloque simple.

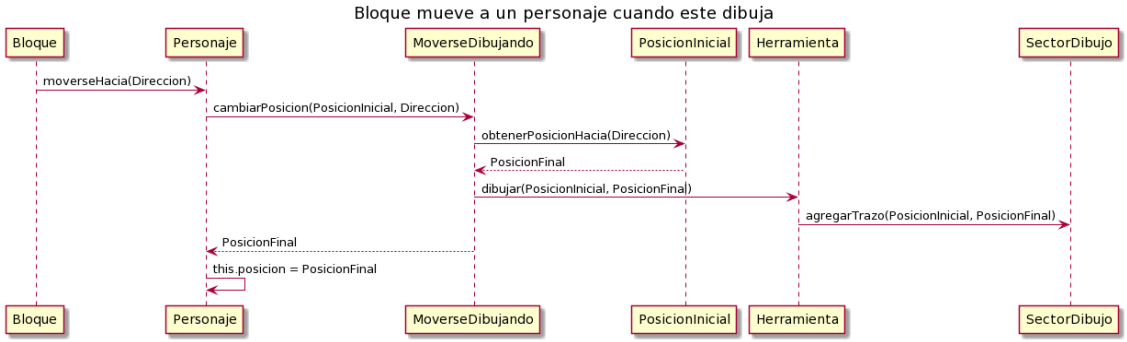

Figura 8: Diagrama sobre la relación de un Bloque de movimiento con el Personaje cuando tiene el Lapiz abajo.

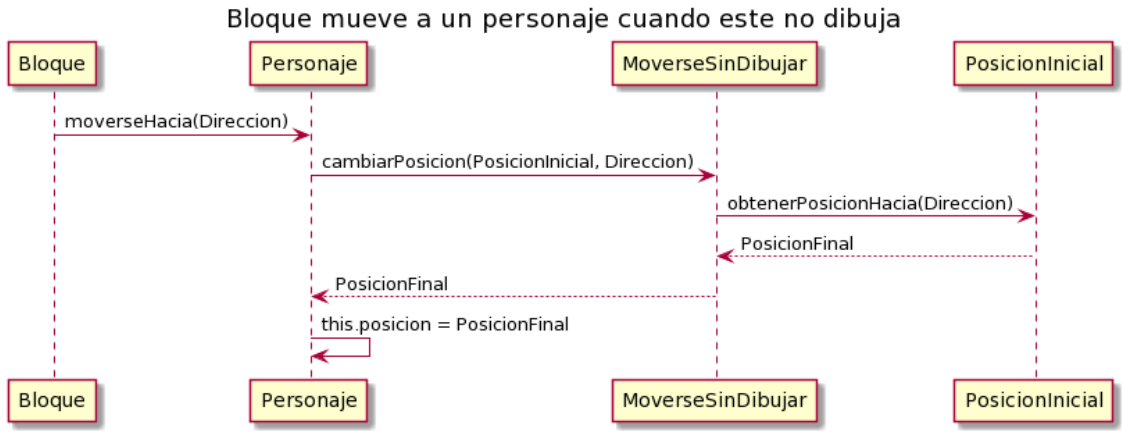

Figura 9: Diagrama sobre la relación de un Bloque de movimiento con el Personaje cuando no tiene el Lapiz abajo.

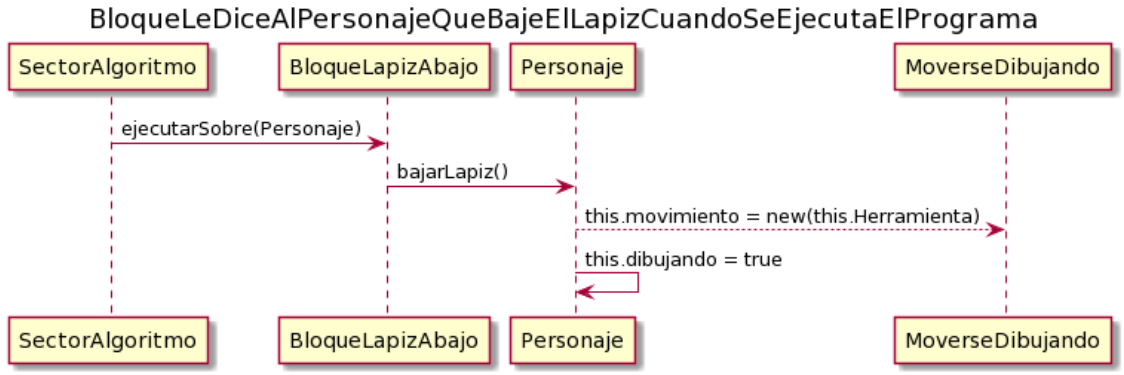

Figura 10: Bloque le dice al personaje que baje el Lapiz cuando se ejecuta el programa.

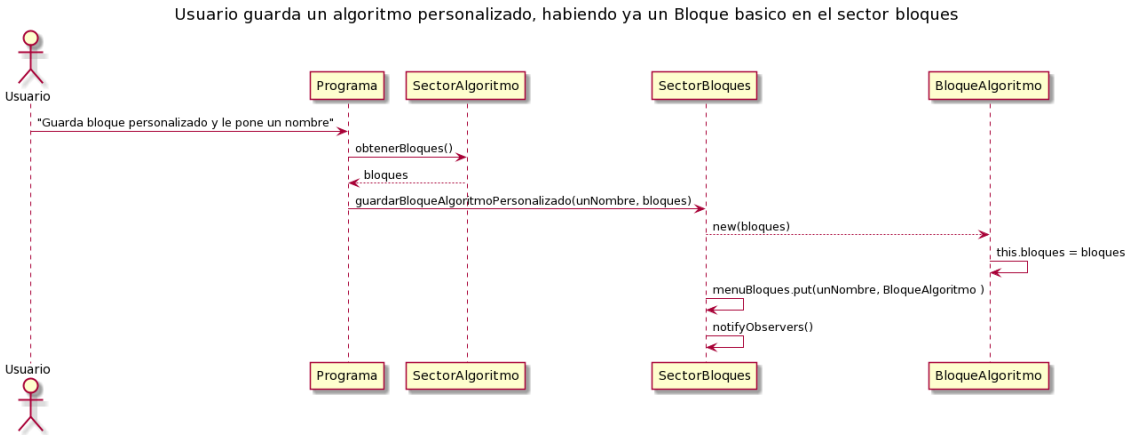

Figura 11: Usuario guarda un Algoritmo Personalizado con un bloque basico ya previamente seleccionado.

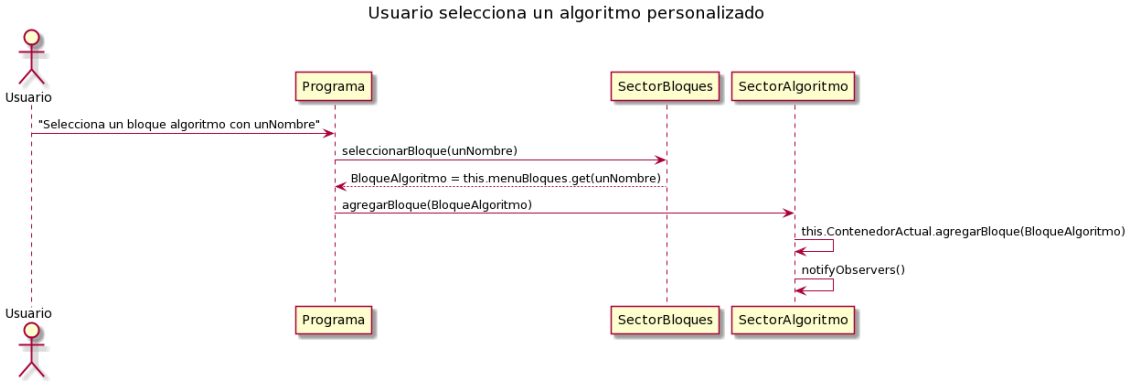

Figura 12: Usuario selecciona un Algoritmo Personalizado que ya habia guardado previamente.

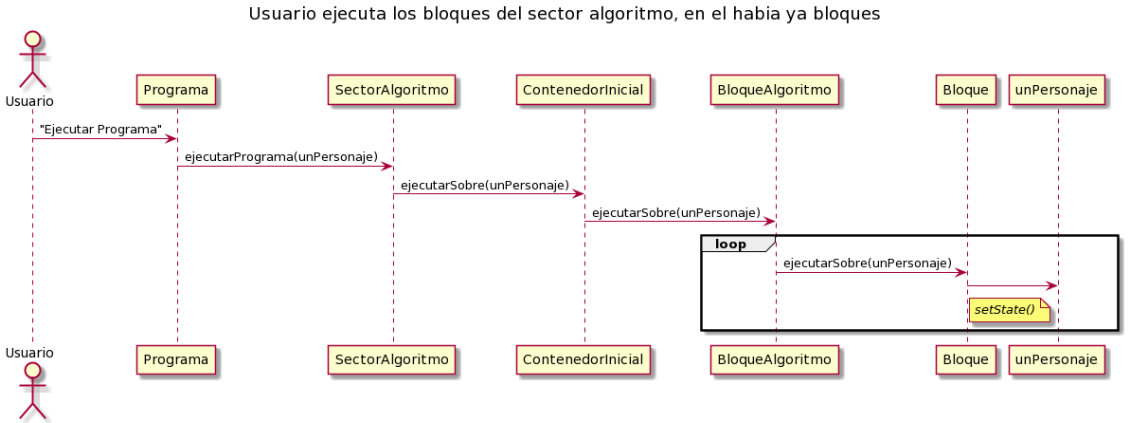

Figura 13: Usuario ejecuta el sector algoritmo, dentro de el hay un Bloque Algoritmo Personalizado.

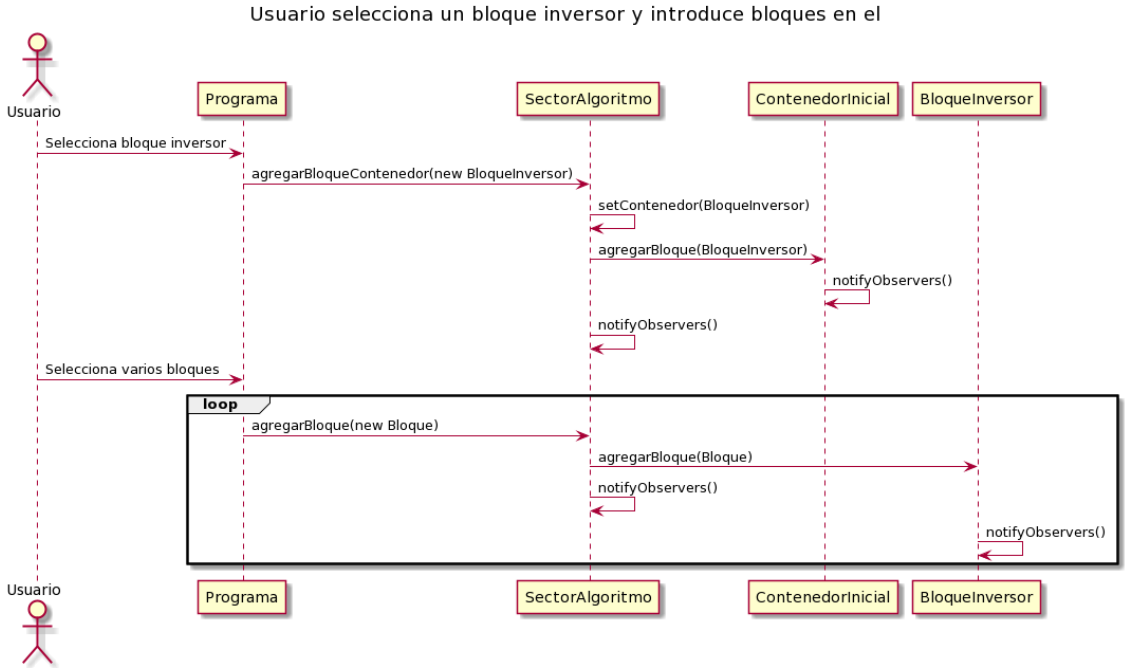

Figura 14: Usuario selecciona un bloque inversor y introduce bloques dentro de el.

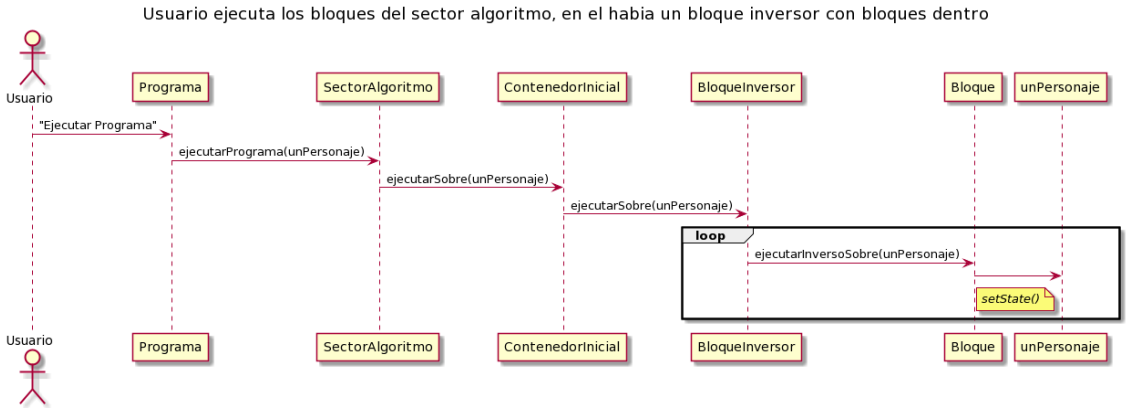

Figura 15: Usuario ejecuta el sector algoritmo, dentro de el hay un Bloque Inversor con bloques dentro.

6. Diagrama de Estado

Figura 16: Diagrama de Estado que refleja la selección de bloques.

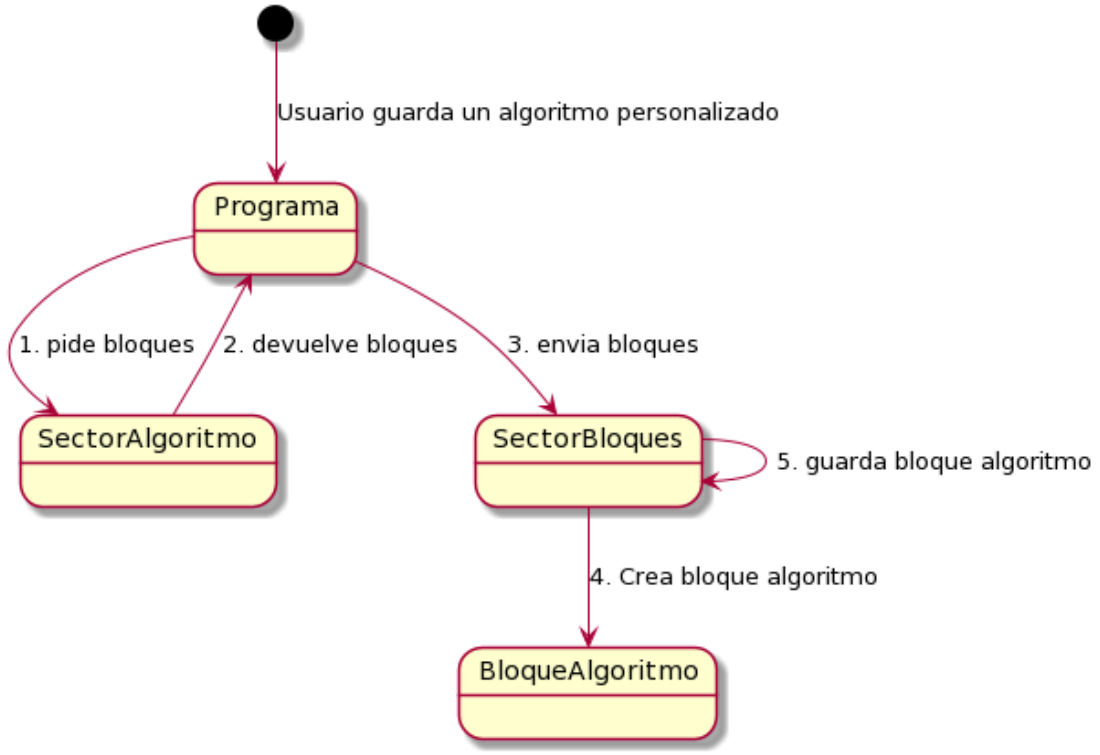

Figura 17: Diagrama de Estado que refleja el guardado de un Bloque Algoritmo Personalizado.

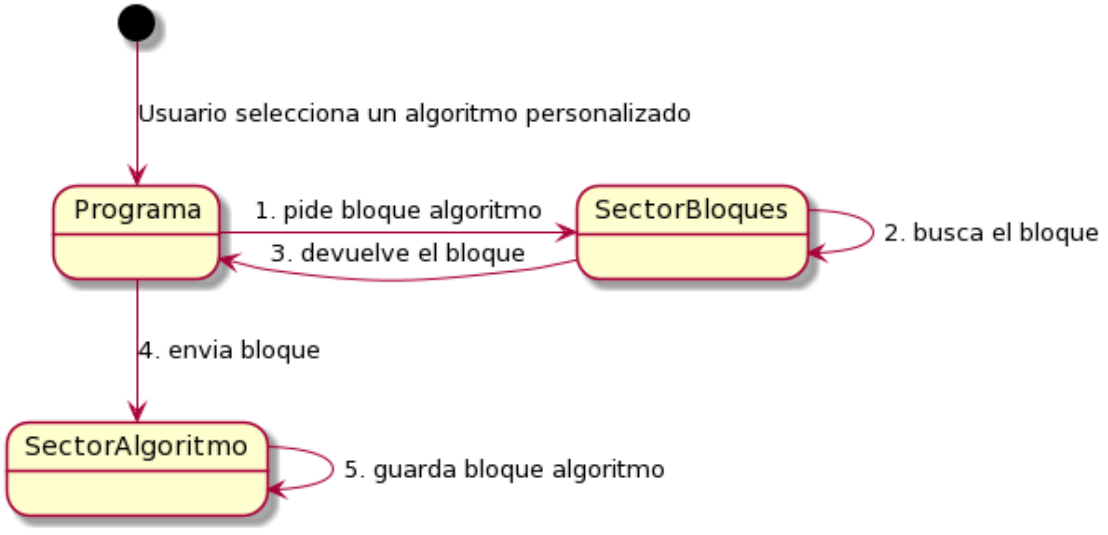

Figura 18: Diagrama de Estado sobre la selección de un Bloque Algoritmo Personalizado por parte del usuario.

7. Diagrama de Paquetes

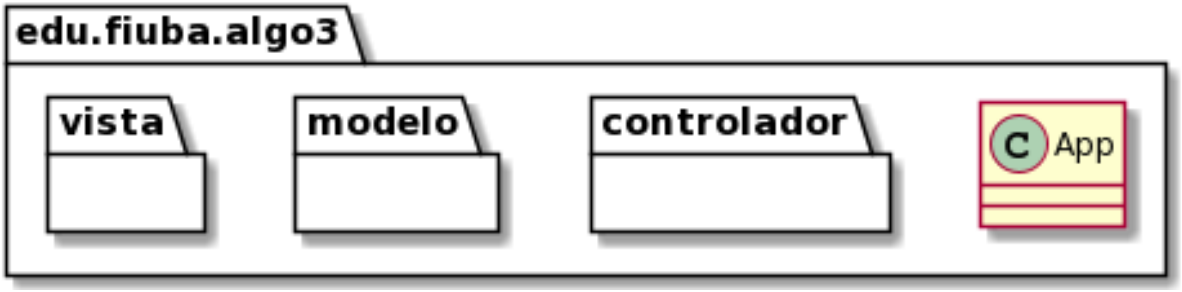

Figura 19: Vista del Paquete edu.fiuba.algo3

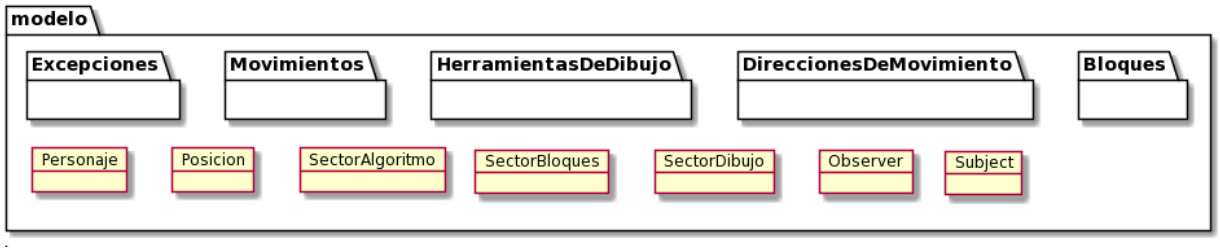

Figura 20: Vista del Paquete Modelo.

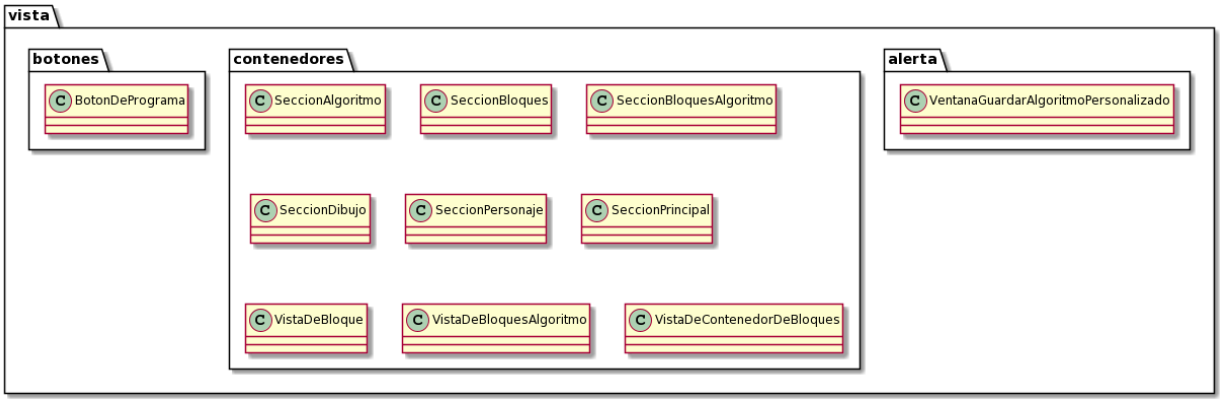

Figura 21: Vista del Paquete Vista.

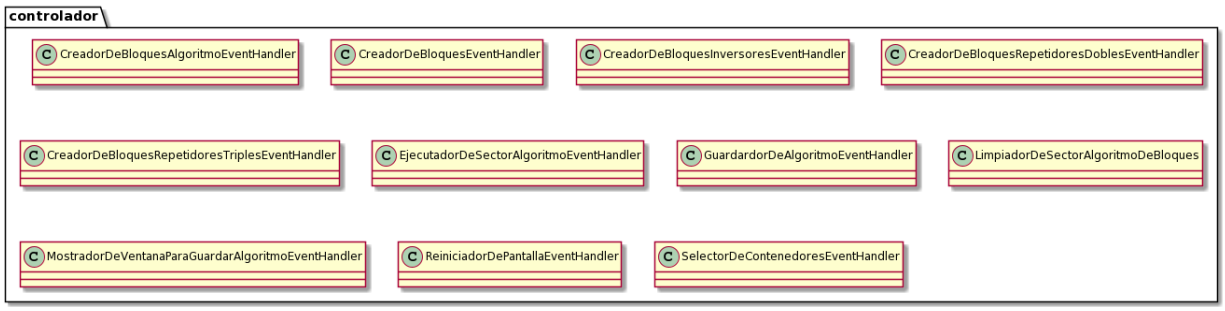

Figura 22: Vista del Paquete Controlador.

8. Detalles de Implementación

8.1. Principios de diseño

8.1.1. Principio de Responsabilidad Única

Verificamos que todas las clases que implementamos a lo largo del trabajo cumplan con este principio, manteniendo cada una, únicamente una responsabilidad, logrando que el programa sea mucho más fácil de mantener ante posibles cambios.

8.1.2. Principio Abierto-Cerrado

Hacemos uso de este principio puntualmente en el caso del Personaje, el cuál utiliza la interfaz Movimiento, misma que es implementada por las clases MovimientoSinDibujar y MovimientoDibujando. La primer clase estando encargada de desplazar al personaje sin dibujar y la segunda encargándose tanto de mover al personaje como de dibujar en el SectorDibujo. Es así que garantizamos de esta forma que cada una de nuestras clases tengan un comportamiento propio, quedándose cerradas para modificaciones, pero abiertas gracias al uso de la interfaz Movimiento, permitiendo al mismo tiempo implementar en un futuro si lo necesitáramos, otros tipos de movimientos posibles para el personaje.

8.2. Patrones de diseño

8.2.1. Patrón Strategy, Movimiento, MovimientoDibujando y MovimientoSinDibujar

Implementamos la interfaz *Movimiento*, la cual sirve de utilidad para que el Personaje, internamente pueda adquirir diferentes comportamientos de movimiento. Para esto, la interfaz posee la firma del método cambiarPosicion el cuál las clases MovimientoDibujando y MovimientoSinDibujar implementan cada una de forma diferente. El MovimientoDibujando, utilizará el Lápiz, que interactúa con el SectorDibujo y le devolverá una nueva posición al Personaje para que actualice su posición. El MovimientoSinDibujar hará lo mismo que la clase anterior, sólo que no utilizará el Lápiz. Para hacer uso de esta idea, nos basamos en el patrón Strategy, el cuál busca tener un comportamiento que sea solucionado en tiempo de ejecución, ya que a lo largo de la vida útil del personaje este deberá utilizar o no su lápiz según se le indique. Estas indicaciones serán llevadas a cabo por la ejecución de los bloques BloqueLapizArriba y BloqueLapizAbajo.

8.3. Diseño en profundidad

8.3.1. Bloques

Para el armado y uso de bloques, decidimos crear una interfaz que lleva el nombre Bloque. La cuál es implementada por varias clases, estos bloques cuentan con funcionalidades únicas y se pueden utilizar mediante el llamado a dos métodos, los cuáles deben recibir al Personaje como parámetro.

8.3.2. Bloques del Lápiz y el Personaje

Para que el Personaje pueda hacer uso a futuro del Lápiz, es condición necesaria que este lo reciba por parámetro en su constructor de clase. Al mismo tiempo, se deberá recurrir al uso y ejecución de los bloques BloqueLapizArriba y BloqueLapizAbajo. Estos bloques se comunican directamente con el personaje mediante métodos, los cuales le indican que tipo de movimiento deberá utilizar en adelante. De esta forma, el Lápiz podrá ser utilizado o no, dependiendo únicamente del tipo de movimiento utilizado en tiempo de ejecución por el personaje. Estas clases de movimiento, poseen cada una un comportamiento especifico el cuál esta relacionado particularmente con el SectorDibujo, al momento de dibujar sobre él. Como última condición, para que el Personaje pueda dibujar con el Lápiz, se le debe indicar que debe moverse con los bloques que poseen esta funcionalidad.

8.3.3. Bloques de movimiento y el Personaje

El movimiento del Personaje está dado por la ejecución de cualquiera de los siguientes bloques, BloqueArriba, BloqueAbajo, BloqueDerecha y BloqueIzquierda, estos le envían al Personaje la dirección de movimiento en la cuál debe moverse, la cuál entiende mediante el uso de un método. El personaje, al recibir la dirección a la cual debe moverse llamara al tipo de movimiento utilizado, con la posición y la dirección indicada y actualizará la posición en la que se encuentra.

8.3.4. Bloque Contenedor de Bloques

Creada para representar la capacidad de un Bloque de contener otros Bloques (de ahora en mas llamado Bloque Contenedor).

Dentro del modelo reconocimos Bloques que compartian la capacidad de contener otros Bloques (Bloque Inversor y Bloque Repetidor) y que el Sector Algoritmo tambien necesitaba comportamientos similares.

Para cumplir este rol implementamos la clase BloqueContenedor con los siguiente metodos:

- agregarBloque(Bloque unBloque) : Agrega un Bloque al Bloque contenedor.
- removerBloque() : Remueve el ultimo Bloque agregado al Bloque Contendor, si no hay Bloques a remover no hace nada.
- vaciarContenedor( ): Remueve todos los Bloques agregados al Bloque Contendor.

Al implementar la interfaz Bloque, define los metodos de la siguiente manera:

- ejecutarSobre(Personaje personaje) : Ejecuta el metodo ejecutarSobre() de todos los Bloques que contiene, en orden, sobre el personaje dado.
- ejecutarInversoSobre(Personaje personaje) : Ejecuta el metodo ejecutarInversoSobre() de todos los Bloques que contiene, en orden, sobre el personaje dado.

Esta clase es heredada por los bloques Inversor y Repetidor, e incluida en el Sector Algoritmo para administrar los Bloques.

8.3.5. Sector Algoritmo

Nuestra implementacion del Sector Algoritmo cumple mas la funcion de administrador de Bloques Contenedores.

Al crearse, inicia con un BloqueContenedor interno y se guarda como Contenedor actual

- agregarBloque(Bloque unBloque) : Agrega el Bloque dado al Contenedor sobre el que se este trabajando actualmente
- agregarBloqueContenedor(BloqueContenedor unBloqueContenedor) : Agrega el BloqueContenedor dado al Contenedor sobre el que se este trabajando actualmente y lo guarda como Contenedor actual.

Para cambiar el Contenedor sobre el que se este trabajando sin tener que agregar uno nuevo se utiliza

- establecerContenedor(BloqueContenedor unBloqueContenedor) : Guarda el BloqueContenedor como Contenedor actual.

9. Excepciones

9.1. SinBloquesADevolverException

Representa la falta de bloques para devolver en un Bloque Contenedor

Se eleva cuando se quieren obtener los bloques de un Bloque Contenedor al que no se le agregaron otros Bloques