

Trabajo Práctico II — JAVA

AlgoBlocks

[7507/9502] Algoritmos y Programación III

Curso 2

Segundo cuatrimestre de 2020

Alumno	Número de padrón	Email
Diaz, Luz M.	105122	ldiazc@fi.uba.ar
Torres, Lucas G.	97819	lgtorres@fi.uba.ar
González, Matías Ignacio	104913	maigonzalet@fi.uba.ar

Índice


1. Introducción	2
2. Instrucciones	2
3. Supuestos	3
4. Diagramas de clase	3
5. Diagrama de estado	6
6. Detalles de implementación	6
6.1. Patrón Observer SectorAlgoritmo	6
6.2. Implementación de la vista	7
6.3. Implementación bloques de movimiento (Modelo)	7
6.4. Implementación Lápiz (Modelo)	7
6.5. Implementación de ejecución del bloque opuesto (Modelo)	8
7. Excepciones	8
8. Diagramas de secuencia	9
9. Diagramas de Paquetes	12
10. Imágenes de la aplicación	14

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 un juego que permite aprender los conceptos explicados a lo largo de la materia, haciendo énfasis en la POO, generando algoritmos utilizando bloques visuales. Los algoritmos permitirán a un personaje moverse por la pantalla mientras dibuja con un lápiz, logrando realizar diferentes diseños en un tablero implementado en el juego.

2. Instrucciones

Este es un nuevo tipo de desafío llamado Artista.
Jugá con los bloques y dibujá algo divertido!



Cada bloque en el sector de Bloques Disponibles tiene una función.
Experimentá con cada uno y diseñá tus propios algoritmos!
Podés arrastrar cada bloque al sector Algoritmo.
Luego podés ejecutar tu algoritmo presionando el botón 'Ejecutar'.

Shortcuts:
M: Activar/Desactivar Musica
E: Ejecutar Algoritmo
D: Limpiar Algoritmo
G: Guardar Algoritmo
I: Mostrar Instrucciones

Figura 1: Instrucciones.

3. Supuestos

Se debe tener en cuenta lo siguientes supuestos:

Supuesto I: No se puede insertar el bloque en una posición predeterminada, siempre se aplica al final de la zona donde se lo suelte (sea adentro del contenedor madre, o adentro de un contenedor de repeticion/inversión).

Supuesto II: Para guardar/ejecutar un algoritmo, es necesario al menos 1 bloque en la zona de trabajo.

Supuesto III: El personaje desaparece si excede el Rango de su sector. Vuelve a aparecer si retorna al sector dibujo.

Supuesto IV: No se pueden borrar bloques de un algoritmo personalizado que ya fue guardado.

Supuesto V: El personaje inicia con el lapiz levantado (sin la capacidad de dibujar), es necesario agregar un bloque para que apoye el lapiz.

4. Diagramas de clase

Nuestro proyecto utiliza el patrón MVC (Model-View-Controller) para organizar las clases creadas. En los diagramas a continuación, mostramos la relación entre ellas:

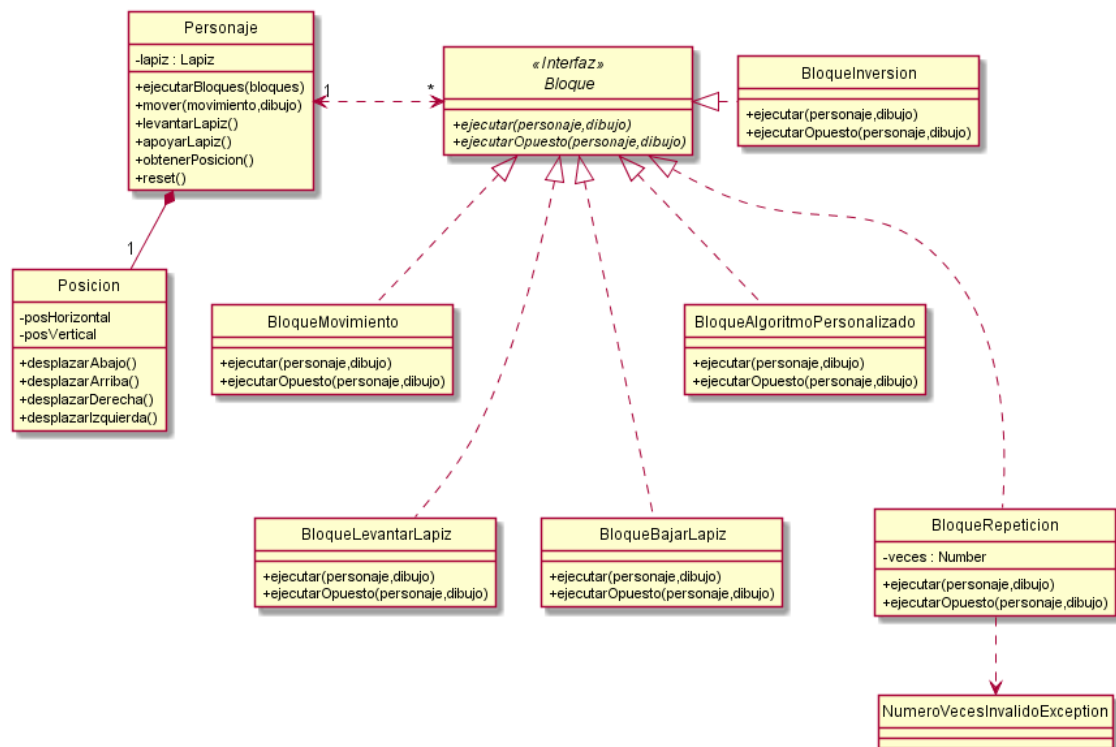


Figura 2: Bloque y Personaje.

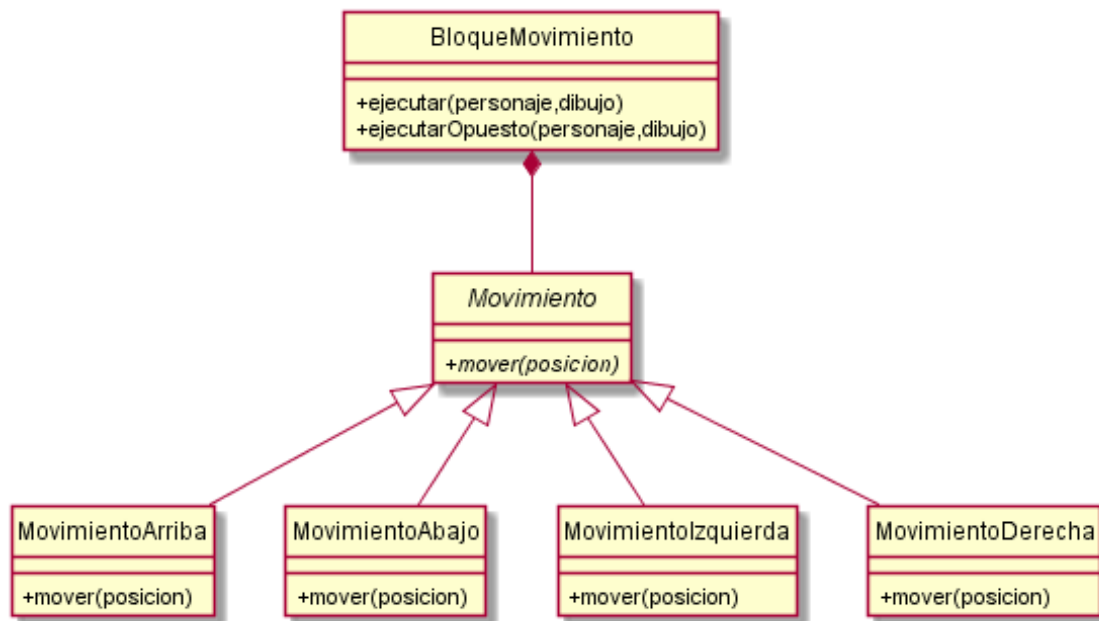


Figura 3: BloqueMovimiento.

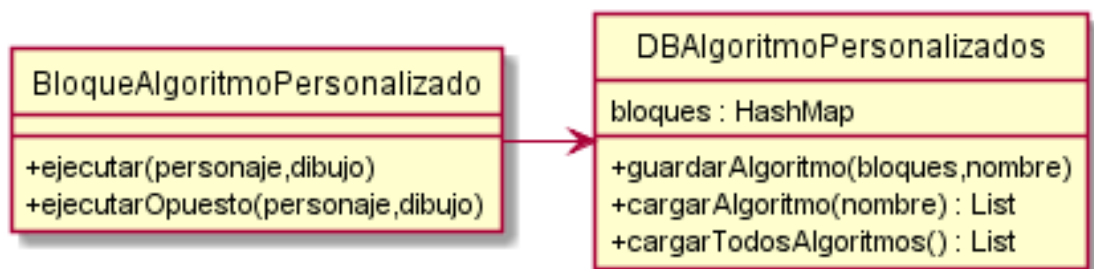


Figura 4: BloqueAlgoritmoPersonalizado.

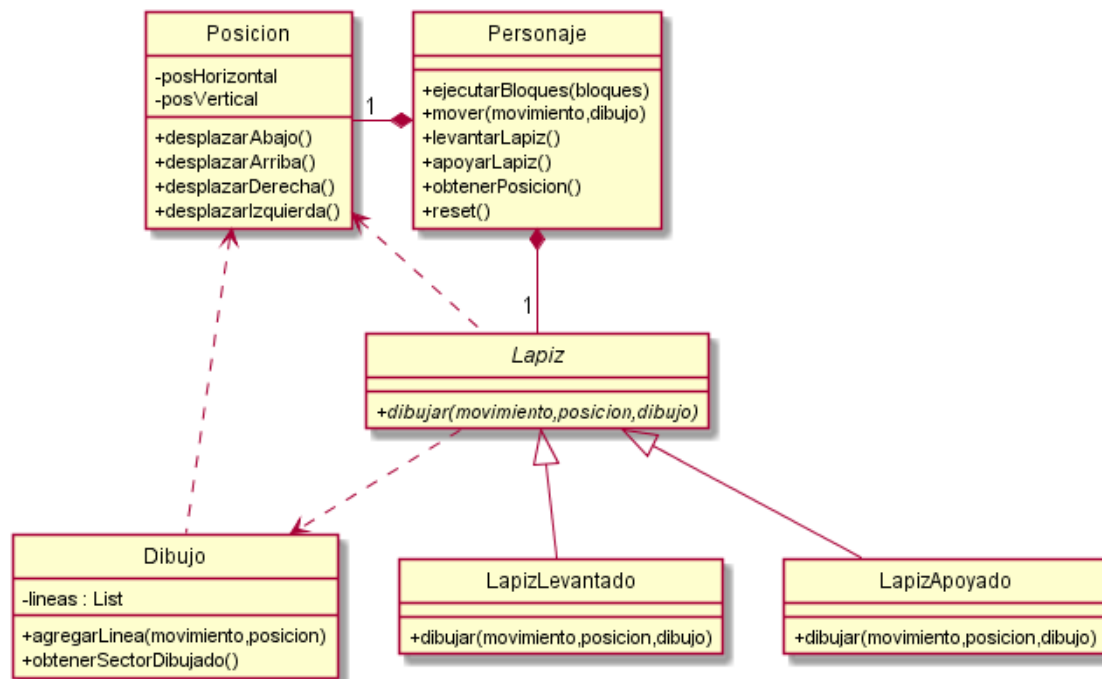


Figura 5: Personaje.

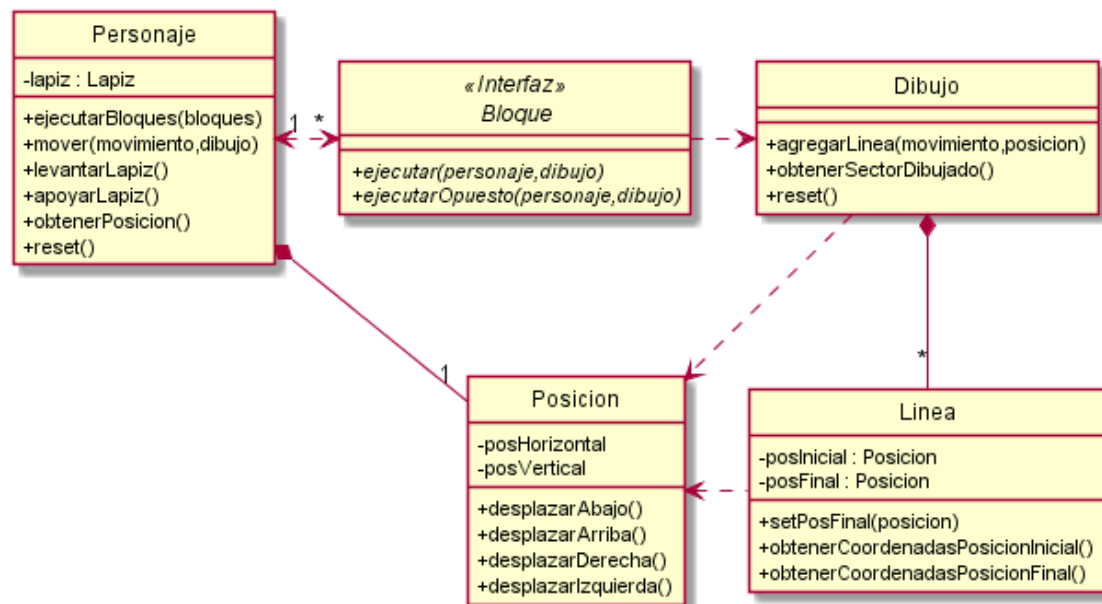


Figura 6: Dibujo.

5. Diagrama de estado

El siguiente diagrama de estado muestra como el juego avanza desde que se lo ejecuta, hasta que finaliza el mismo.

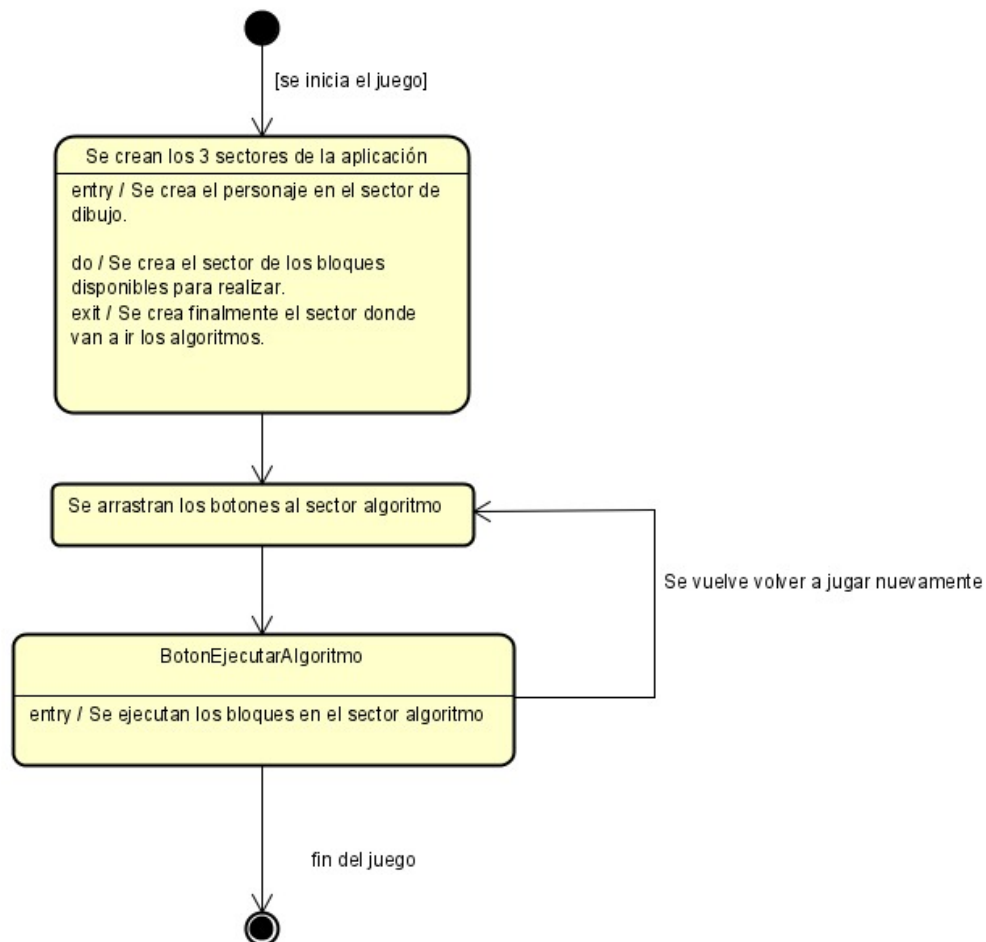


Figura 7: Diagrama para funcionamiento del juego.

6. Detalles de implementación

6.1. Patrón Observer SectorAlgoritmo

Este patrón es fundamental porque conecta las 3 zonas de trabajo del juego. Conecta el sector de bloques disponibles con el Sector algoritmo, y conecta el sector Dibujo con el sector algoritmo. Cada vez que se agrega un bloque en el sector algoritmo, automáticamente se actualiza el estado del "Botón Guardar Algoritmo", y se actualiza la lista de bloques a ejecutarse cuando se presione el botón Ejecutar! en sector Dibujo.

6.2. Implementación de la vista

Como dicho anteriormente se usa el patrón MVC, de modo tal que se tienen dos representaciones del estado en el modelo, VistaPersonaje y VistaLinea. Vista Personaje es realizado a través del Patrón Observer, de modo tal que VistaPersonaje es el objeto observador, mientras que la clase Personaje es el objeto observado, un cambio de estado del Personaje en el modelo será notificado a VistaPersonaje, tal para poder moverse, además teniendo en cuenta su sentido de giro.

Para VistaLinea se implementa el mismo Observer, cada vez que este es notificado por la clase Dibujo

```
public void agregarLinea(Movimiento movimiento, Posicion posicion) {  
    Linea linea = new Linea(posicion);  
    movimiento.mover(posicion);  
    linea.setPosFinal(posicion);  
    lineas.add(linea);  
    notificarObservador();  
}
```

nos da los datos que guarda la lista de Lineas para poder representarlo en el SectorDibujo.

6.3. Implementación bloques de movimiento (Modelo)

Esta parte del modelo se implementa a partir de dos principales clases: BloqueMovimiento y Movimiento donde la primera contiene como atributo a la segunda.

Cuando se llama a ejecutar del BloqueMovimiento, este informa al personaje que 'debe' moverse y le manda su propio Movimiento que le dirá 'como' modificar su posición:

En la clase BloqueMovimiento:

```
@Override  
public void ejecutar(Personaje personaje, Dibujo dibujo) {  
    personaje.mover(movimiento, dibujo);  
}
```

En la clase MovimientoArriba:

```
@Override  
public void mover(Posicion posicion){  
    posicion.desplazarArriba();  
}
```

6.4. Implementación Lápiz (Modelo)

La principal responsabilidad de la clase Lápiz es decidir si al moverse el personaje se crea una nueva Linea en el tablero o no. Esta responsabilidad se implementa a través de dos subclases: LápizApoyado y LápizLevantado.

En la clase LápizLevantado: (No dibuja)

```
@Override  
public void dibujar(Movimiento movimiento, Posicion posicion, Dibujo dibujo){  
    movimiento.mover(posicion);  
}
```

En la clase LápizApoyado: (Si dibuja)

```
@Override  
public void dibujar(Movimiento movimiento, Posicion posicion, Dibujo dibujo){  
    dibujo.agregarLinea(movimiento,posicion);  
}
```


6.5. Implementación de ejecución del bloque opuesto (Modelo)

El juego permite la oportunidad de hacer ejecuciones inversas, es decir, si se arrastra un bloque de movimiento hacia la derecha en el sector algoritmo, el personaje ejecutará un movimiento hacia la izquierda. Para solucionar este problema manteniendo las reglas de POO dadas por la cátedra, evitando enviar información por parámetros y/o condicionales, implementamos un método en la interfaz Bloque, llamado ejecutarOpuesto, el cual permite ejecutar el opuesto que le corresponde, como mostramos a continuación.

Clase interfaz Bloque:

```
public interface Bloque {
    void ejecutar(Personaje personaje, Dibujo dibujo);
    void ejecutarOpuesto(Personaje personaje, Dibujo dibujo);
}
```

Método ejecutarOpuesto en BloqueLevantarLapiz:

(Apoya el lapiz en el sector y todos los bloques siguientes deberían permitir hacer un dibujo en el Sector Dibujo):

```
@Override
public void ejecutarOpuesto(Personaje personaje, Dibujo dibujo) {
    personaje.apoyarLapiz();
}
```

Método ejecutarOpuesto en BloqueRepetición:

```
@Override
public void ejecutarOpuesto(Personaje personaje, Dibujo dibujo){
    for( int j = 0; j < this.veces_repeticion; j++ ) {
        bloques.forEach( bloque -> bloque.ejecutarOpuesto(personaje, dibujo) );
    }
}
```

Método ejecutarOpuesto en un BloqueMovimiento:

```
@Override
public void ejecutarOpuesto(Personaje personaje, Dibujo dibujo) {
    personaje.mover(movimiento.opuesto(), dibujo);
}
```

Esta implementación de la clase Bloque permite a futuro agregar nuevas ejecuciones de forma sencilla, y creando los movimientos que sean necesarios y los opuestos que le correspondan.

7. Excepciones

NumeroVecesInvalidoException Se lanza esta excepción si se desea crear un bloque de repetición en algún rango que no pertenezca entre [2,3].

8. Diagramas de secuencia

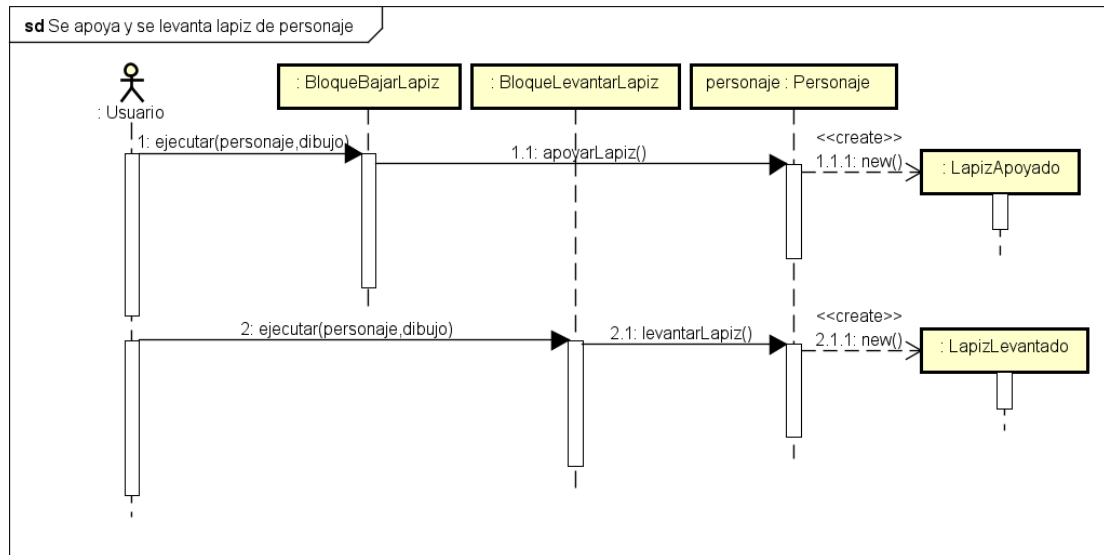


Figura 8: Se apoya y se levanta Lapiz.

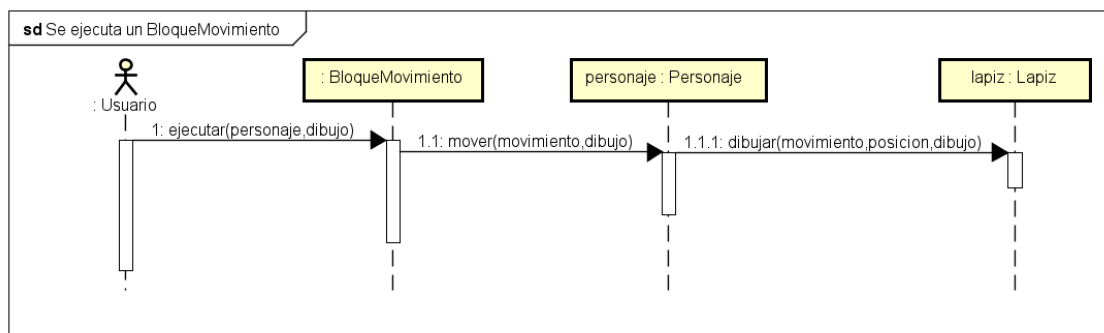


Figura 9: Se ejecuta BloqueMovimiento.

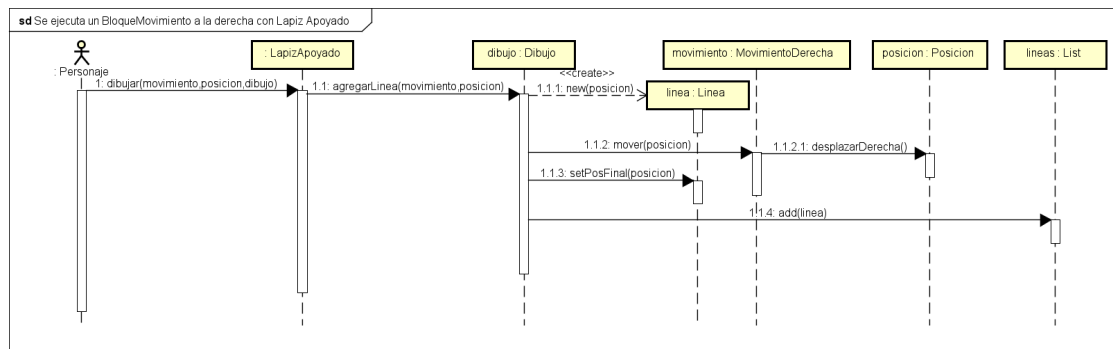


Figura 10: Ejecucion BloqueMovimiento con LapizApoyado.

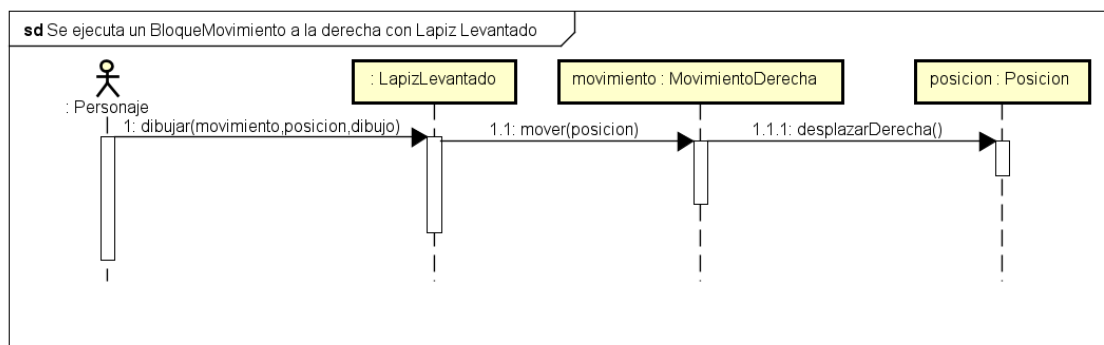


Figura 11: Ejecucion BloqueMovimiento con LapizLevantado.

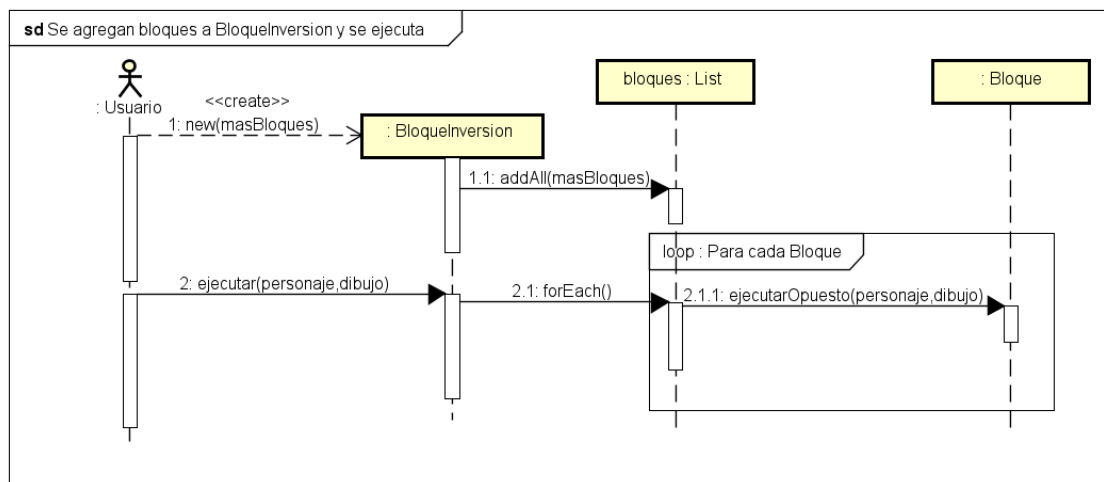


Figura 12: Ejecucion BloqueInversion.

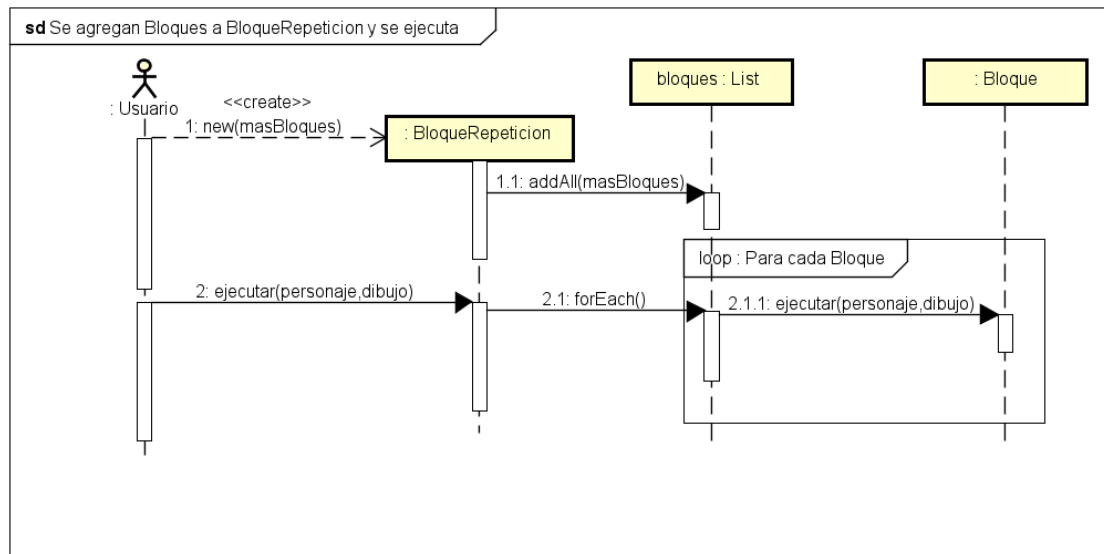


Figura 13: Ejecucion BloqueAlgoritmoPersonalizado.

9. Diagramas de Paquetes

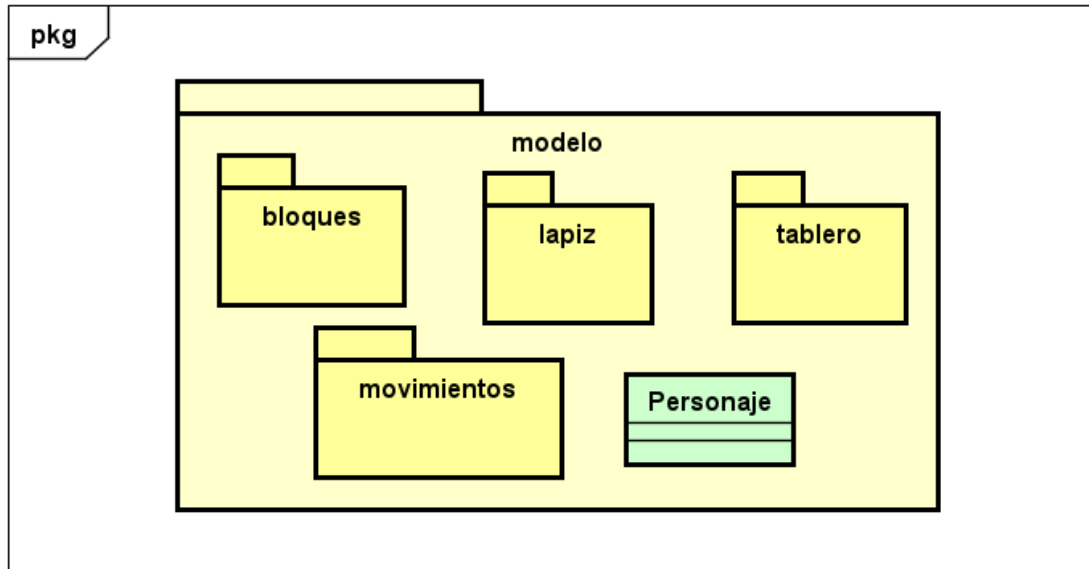


Figura 14: Diagrama de paquetes general del modelo

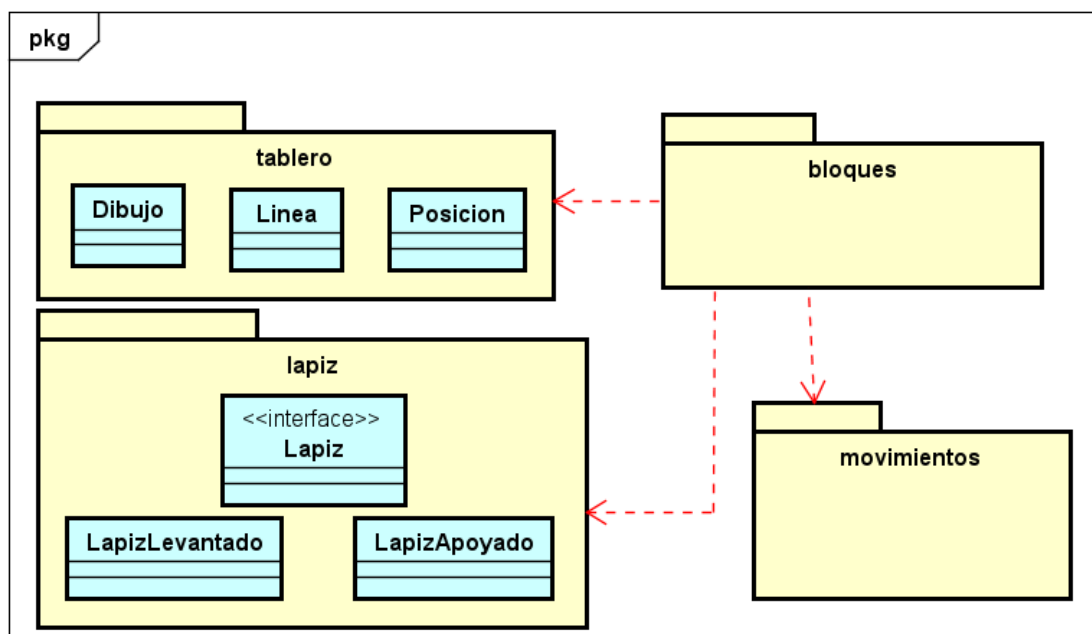


Figura 15: Diagrama de paquetes en detalle del modelo

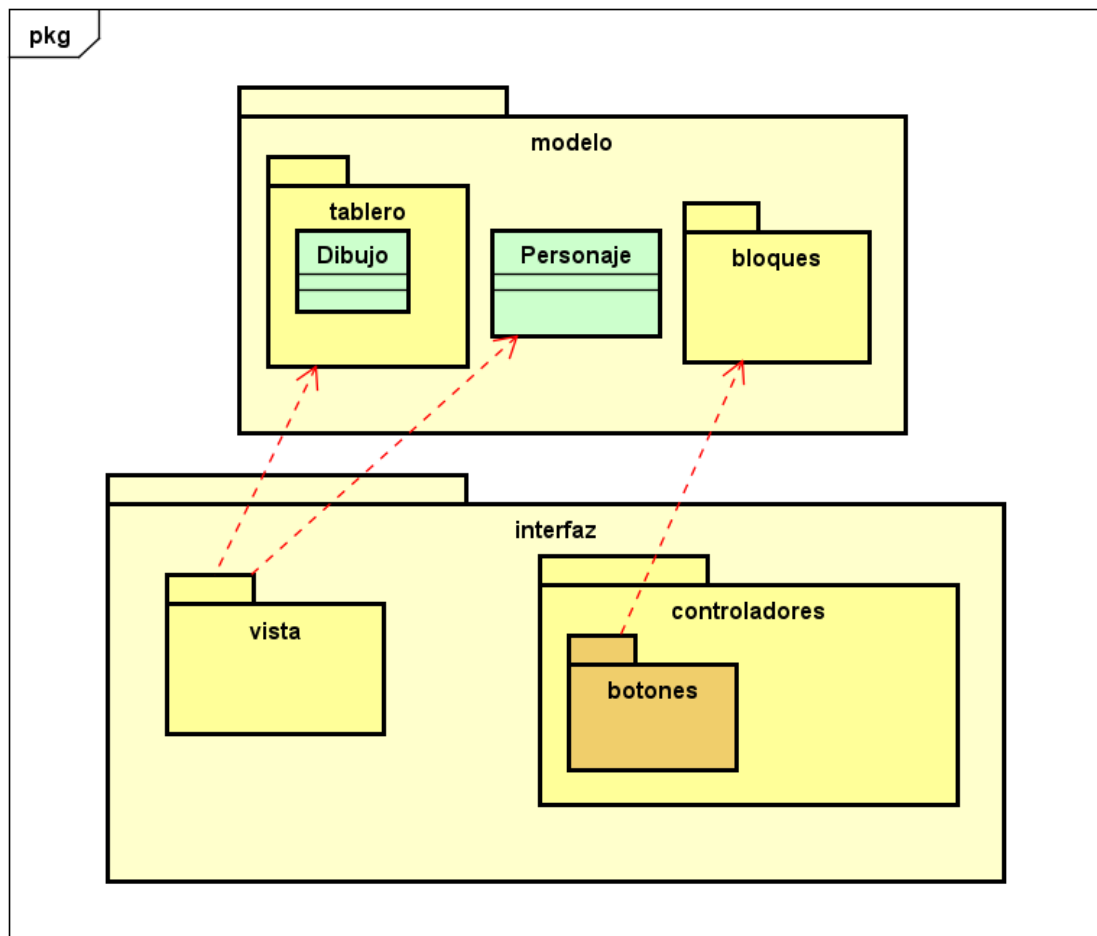


Figura 16: Diagrama de paquetes general del modelo vista e interfaz

10. Imágenes de la aplicación

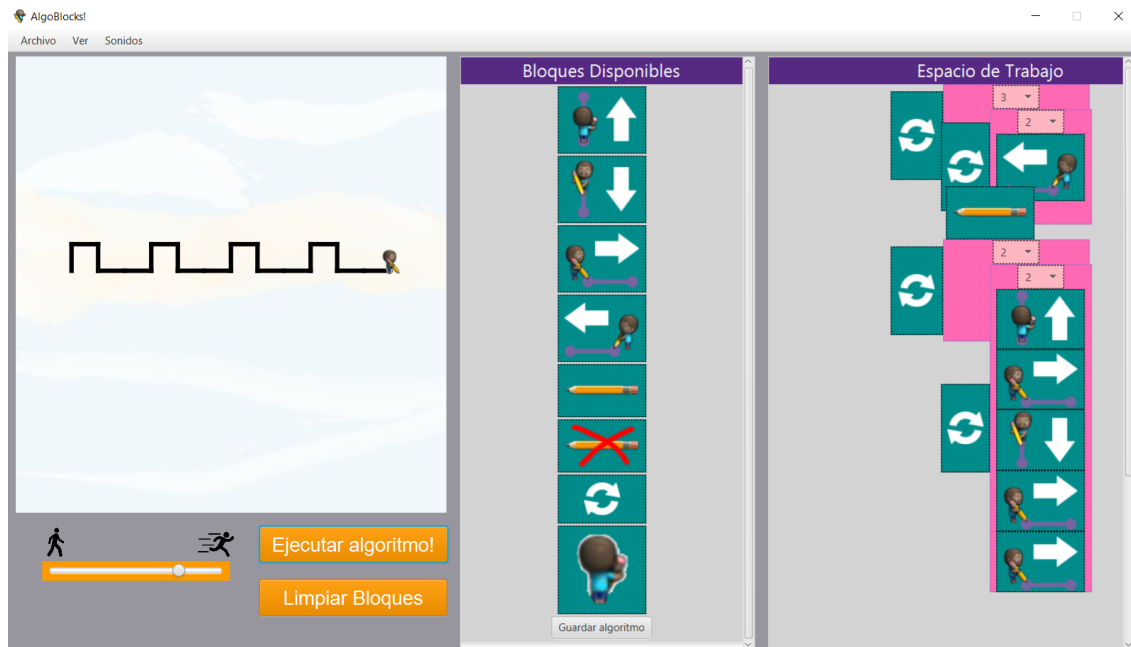


Figura 17: Señal cuadrada con ciclo de trabajo 33 %

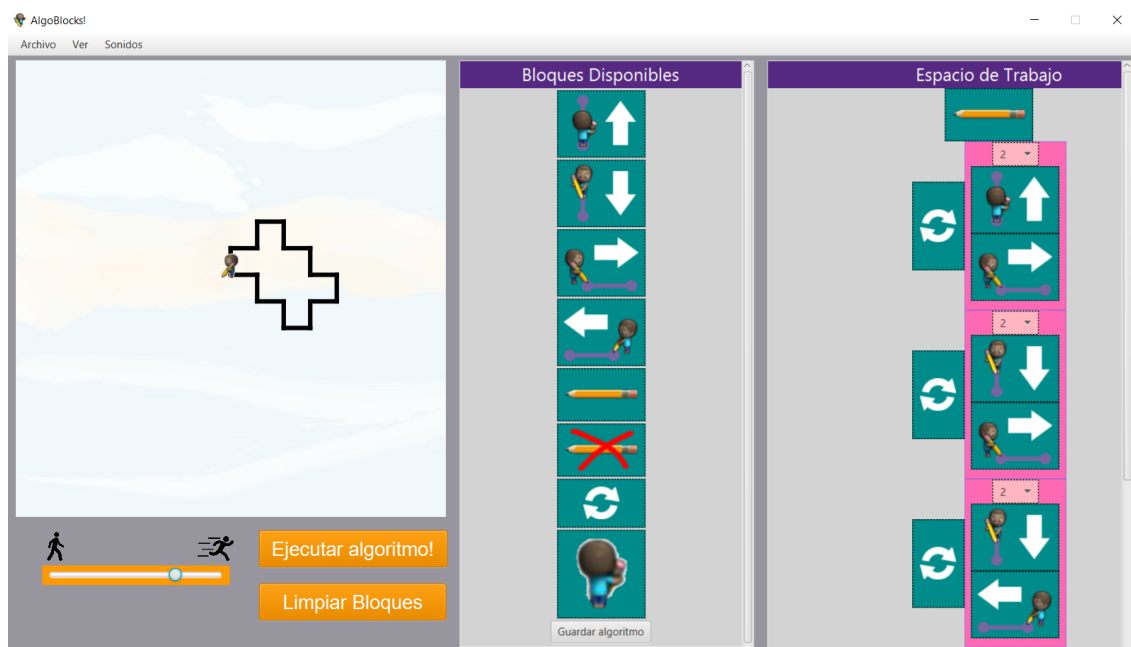


Figura 18: Huesito