Trabajo Práctico 2 — Java

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

Alumno	**Padrón**	**Mail**
Carlocchia, Camila Belén	104225	ccarlocchia@fi.uba.ar
Curetti, Santiago	105133	scuretti@fi.uba.ar
Escalante, Luis Enrique	105204	lescalante@fi.uba.ar
Ramirez, José	93751	jose5deoctubre@gmail.com
Zambrano, Andrés	105500	azambrano@fi.uba.ar

Índice

1. Introducción

El presente informe reúne la documentación del segundo trabajo práctico de la materia Algoritmos y Programación III donde se desarrolló un juego similar a el siguiente ejemplo visual en *Java* utilizando los conceptos del paradigma de la orientación a objetos vistos en el curso, fue implementado usando *Test-driven development* y la interfaz gráfica fue desarrollada usando *JavaFx*. En la aplicación, el usuario interactúa con un *algoritmo* conformado por *bloques*, un *personaje* y su *sector de dibujo*, en donde el fin es crear un dibujo mediante la selección de determinados bloques.

2. Supuestos

A medida que se avanzó en el modelo del trabajo, surgieron los siguientes supuestos:

* Como el personaje no tiene limites de movimiento, implementamos una función en personaje llamada reiniciar posicion donde vuelve al personaje al inicio. Tampoco se lanzará ningún tipo de excepción sí es que el personaje llega a salir de la zona de dibujo.

* Tomamos como supuesto el hecho de que la responsabilidad sobre los tipos de nombres de los bloques personalizados recae sobre el jugador.

* No se pueden anidar bloques de secuencia: esto es, adentro de un bloque de secuencia (bloque de repetición o de inversión) solo pueden ir bloques, más no otro bloque de secuencia. Esto se puede implementar en el modelo, sin embargo, por cuestiones visuales y prácticas, la vista lanza una excepción sí es que se intenta hacer lo descrito anteriormente.

3. Modelo de dominio

A continuación se detallaran brevemente las responsabilidades y razones por las que fue hecha cada parte del modelo.

* **Clase Algoritmo** Representa una secuencia de bloques. Su responsabilidad es guardar ordenadamente los bloques que seleccione el usuario, para luego ejecutarlos de manera correcta. A su vez, sí al usuario le gustó mucho una secuencia de bloques y la desea guardar, el algoritmo es el que se encarga de crear el bloque personalizado, vaciando la lista de bloques en el acto.

* **IObservado e IObservador** Estas interfaces son luego implementadas por el SectorDeDibujo y por el controladorSectorDeDibujo. Implementando este **Patrón Observer**, el sector puede notificar a sus observadores que fue modificado y actuen correspondientemente.

* **ValidaAlgoritmo y ValidaNombre** Son dos clases con un único método que se encarga de revisar un parámetro y devolver un booleano en caso de que sea correcto o lanzar una excepción en caso de que sea inválido. En vez de dejar la responsabilidad a la clase que los convoca (Algoritmo), se decidió crear una clase cuya responsabilidad sea validar dichos parámetros. Por lo que, sí el día de mañana se cambia la forma de validar un nombre o de revisar sí un algoritmo está vacío, se modifican directamente estas clases, sin afectar el comportamiento general del modelo.

Paquete "Bloque"

* **Interfaz Bloque** La interfaz bloque implementa los bloques de movimiento, repetición, activación, desactivación, personalizado y de inversión. La interfaz tiene como métodos ejecutar(Personaje), además de inverso(), método que regresa un Bloque. Por lo tanto, todos los bloques del modelo tienen que implementar dichos mensajes.

* **Bloques de Activación** Las clases *BloqueActivarLapiz* y *BloqueDesactivarLapiz* se encargan de cambiar el estado correspondiente del lápiz. El inverso de activar lapiz es el desactivar lapiz, y viceversa.

* **Bloques de Movimiento** Los bloques de movimiento cuentan con una clase abstracta que tiene implementado el método ejecutar. Hay cuatro tipo de movimientos, a cada uno le corresponde una clase *BloqueMoverAbajo, BloqueMoverArriba, BloqueMoverDerecha, y BloqueMoverIzquierda*, esta clase cuenta con una *dirección*. Al llamar al método inverso del bloque que mueve hacia arriba, devuelve un bloque que mueve hacia abajo).

* **Interfaz de Dirección** La interfaz dirección cuenta con cuatro clases que la implementan, *dirección arriba*, *dirección abajo*, *dirección derecha* y *dirección izquiera*. Estas clases responden cuál es la posición en la que quedará el personaje al moverse en esa dirección.

* **Bloques de Secuencia** La interfaz *BloquesDeSecuencia* define un método que agrega bloques, ya que las clases *BloqueInvertir y BloqueRepeticion* tienen la capacidad de guardar varios bloques, y ejecutarlos todos secuencialmente. Estos bloques tienen un algoritmo a la que se le van añadiendo los bloques dados, cada uno con su respectiva lógica. El método ejecutar no es distinto, ya que cada uno lo hace según su clase.

* **BloquePersonalizado** A diferencia de los bloques de movimiento y de activación, pero igual a los bloques de secuencia, el bloque personalizado se guarda un algoritmo. Sin embargo, no se le pueden añadir bloques una vez creado. Además, este bloque también posee un nombre, ya que como no tiene un comportamiento en específico (solo el dado por el jugador), se le puede diferenciar de esta manera de los otros bloques personalizados.

Paquete "Personaje"

* **Clase Personaje** El personaje es quien crea la clase *lápiz* y la clase *posición* en la ubicación (x=0,y=0). Tiene como responsabilidad de saber cambiar su posición, lo demás lo delega a las clases que tengan la capacidad de cumplir los métodos.

* **Clase Posición** Esta clase cuenta con 2 coordenadas. Tiene un mensaje de clase por cada dirección, por ejemplo Posicion.derechaDe(Posicion) es un mensaje que utiliza la clase DireccionDerecha para saber la próxima posición que va a ocupar el personaje. A su vez, una instancia de posición no se puede comparar a sí misma con otra posición. Para esto está el mensaje de clase ComparaPosiciones(Posicion, Posicion) que devuelve true sí las posiciones tienen las mismas coordenadas o false en otro caso.

* **Clase Lápiz** La clase Lápiz cuenta con un *estado* y un *sector de dibujo*, es responsable de manejar los cambios de estado y dar la orden de dibujar al estado.

* **Interfaz Estado de Lápiz** Esta interfaz es implementada por las clases *estado activado* y *estado desactivado*. Estas clases van a saber cómo comportarse cuando se les pida dibujar: el estado activado se comunica con el sector de dibujo, mientras que el estado desactivado no hace nada cuando se invoca este método.

* **Clase Sector de Dibujo** Su responsabilidad es la de, cuando reciba la orden, guardar el par de posiciones (desde, hasta) que se van a dibujar. Esta acción es llevada a cabo almacenando en una tupla la posición en la que estaba el personaje y la posición a la que avanzará. Una vez guardada la tupla, el sector se comunica con sus observadores (en este caso el controlador del sector de dibujo).

4. Diagramas de clase

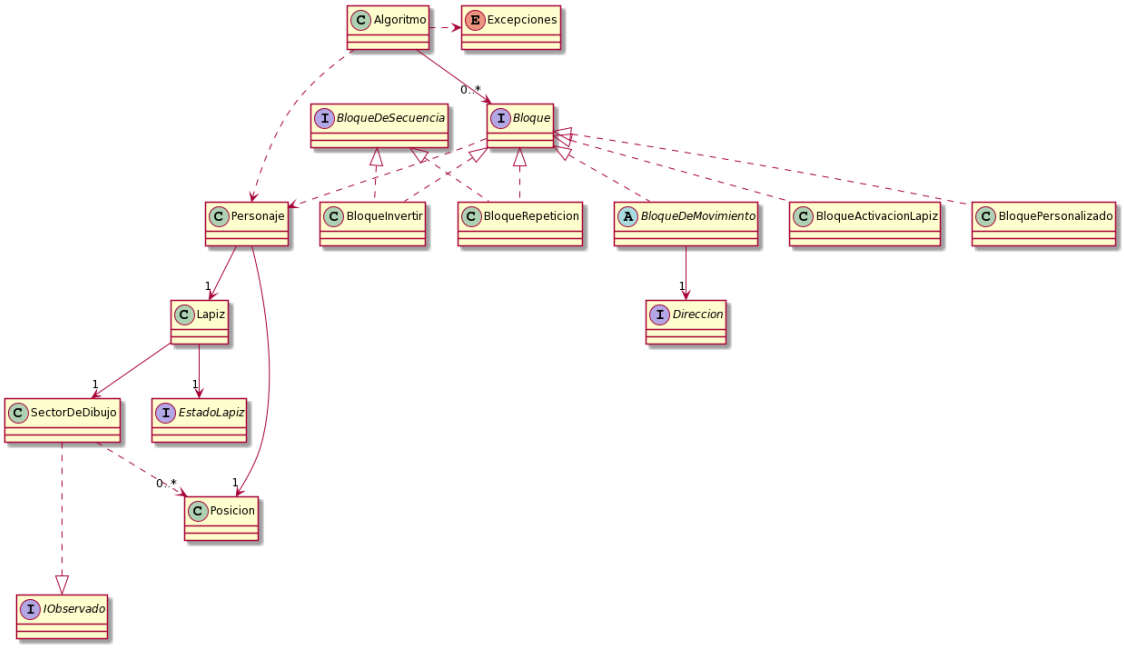


Figura 1: Diagrama general

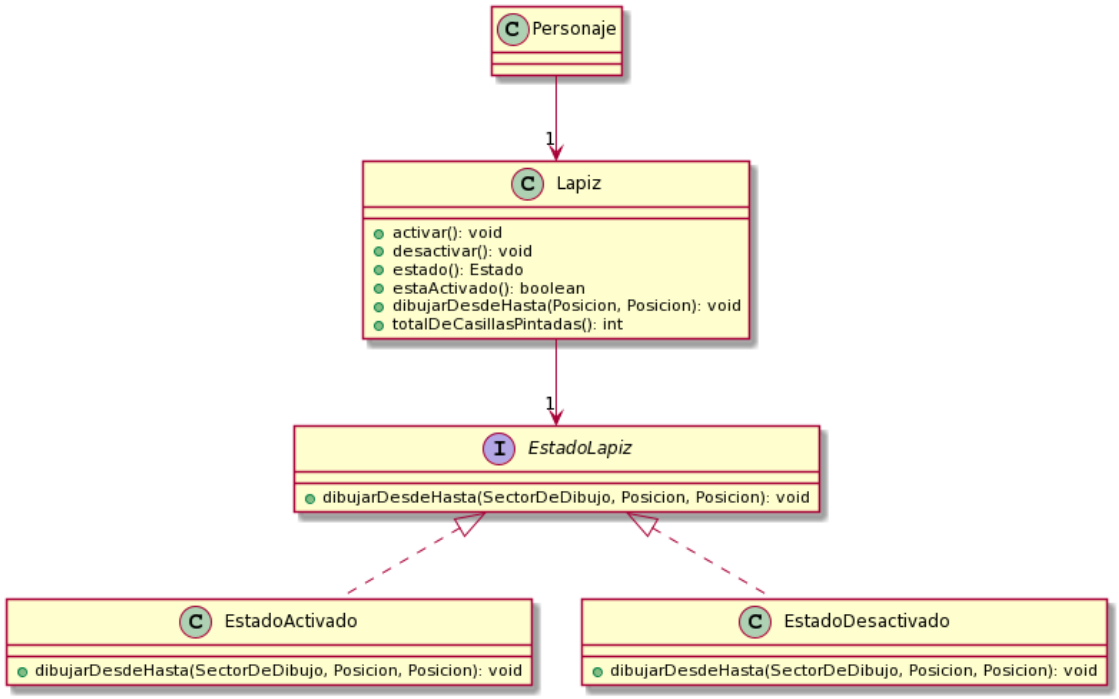


Figura 2: Diagrama de Personaje y lapiz

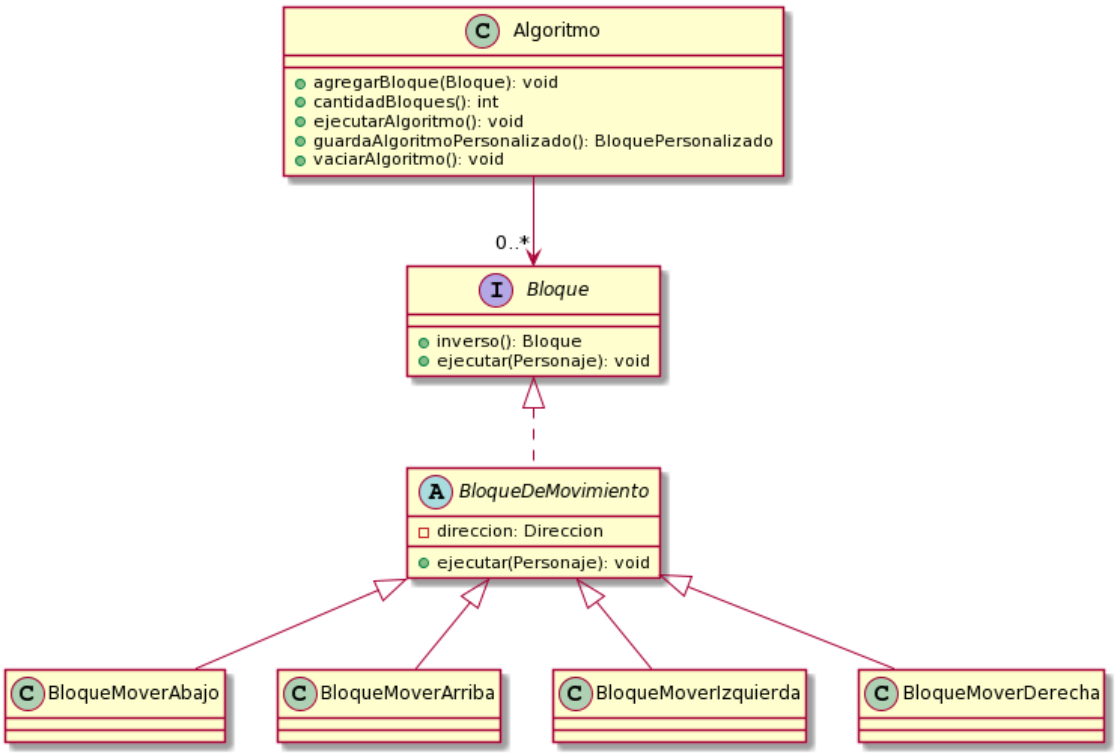
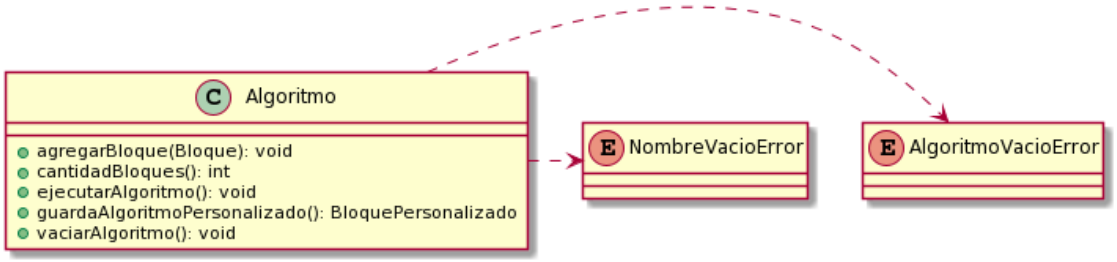


Figura 3: Diagrama Algoritmo y relacion con los Bloques



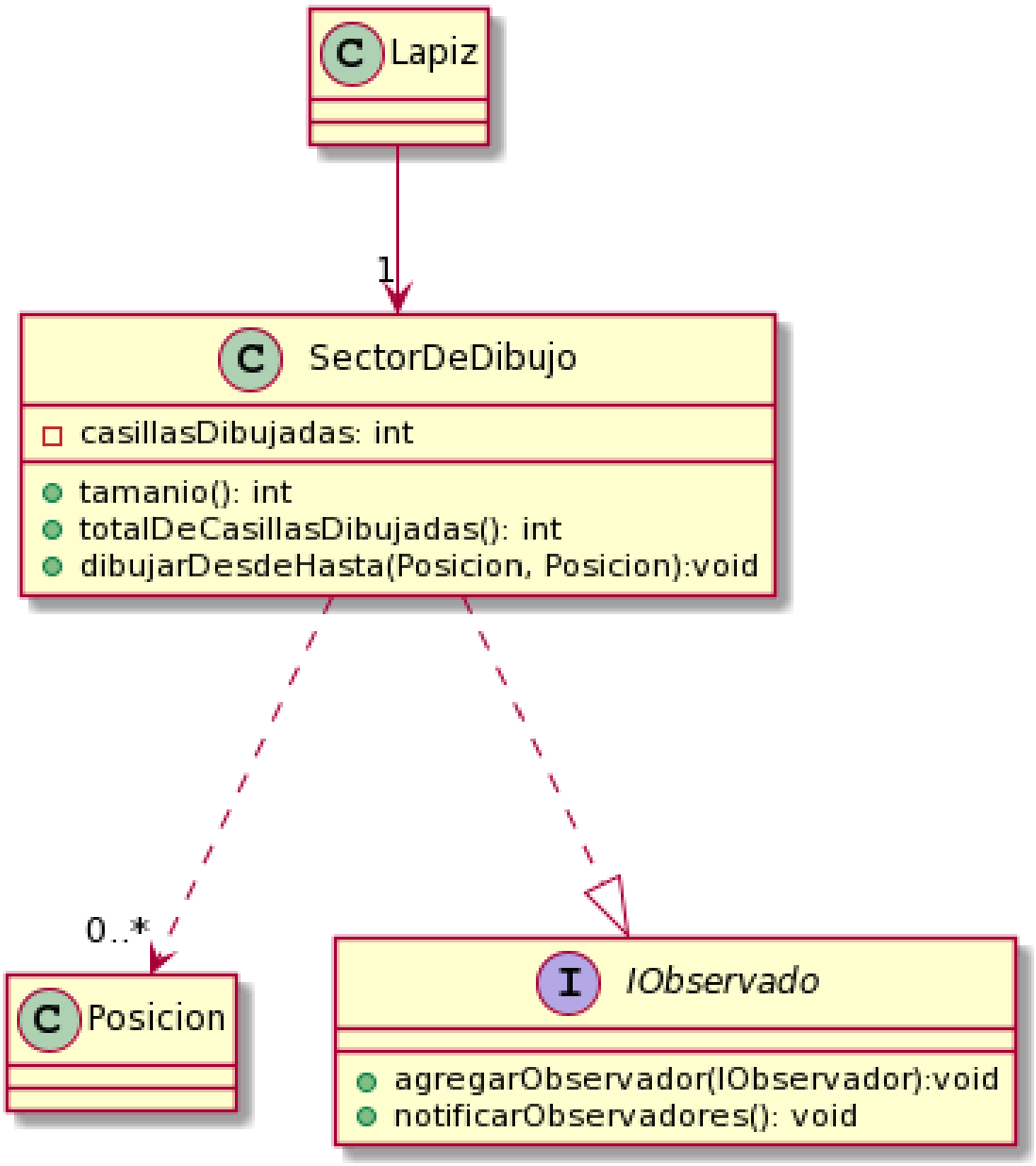


Figura 4: Diagrama de lapiz

.

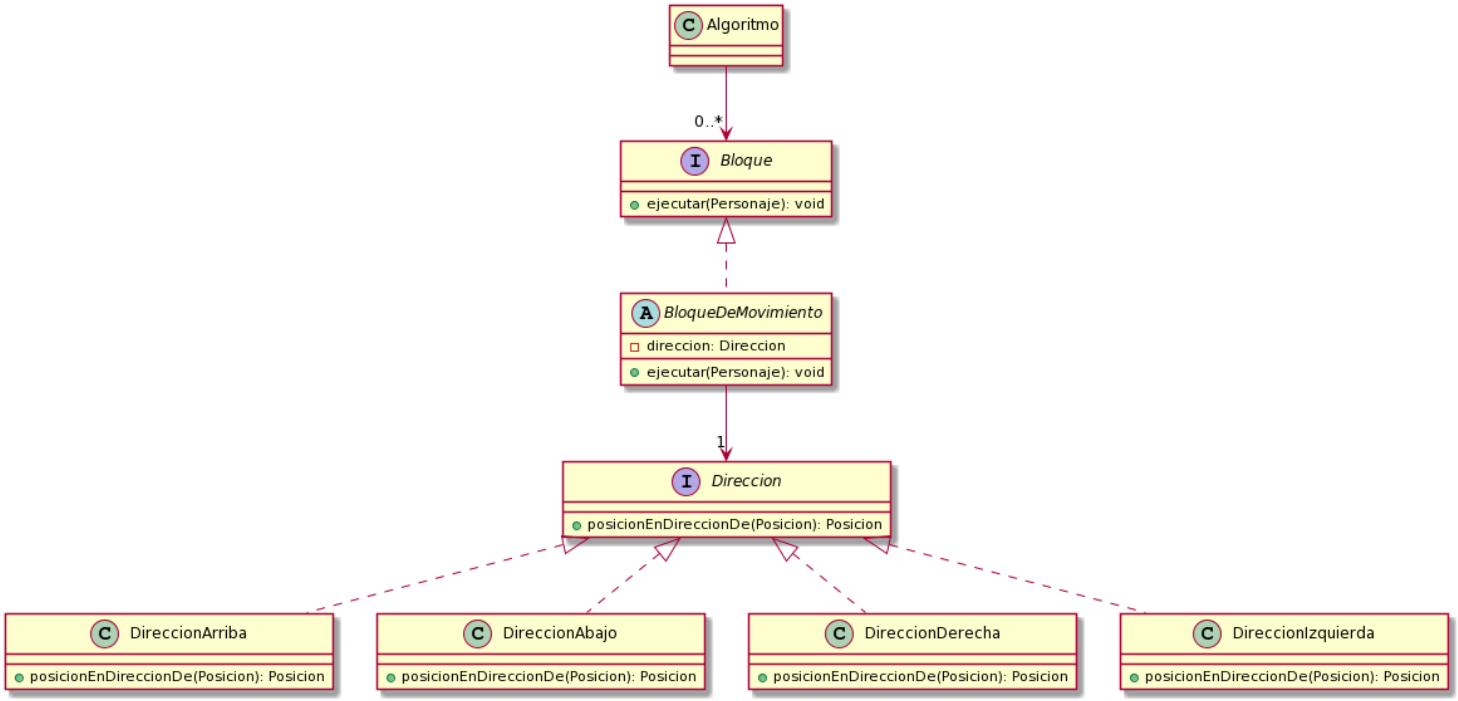


Figura 5: Diagrama de algoritmo interfaz bloque y direcciones

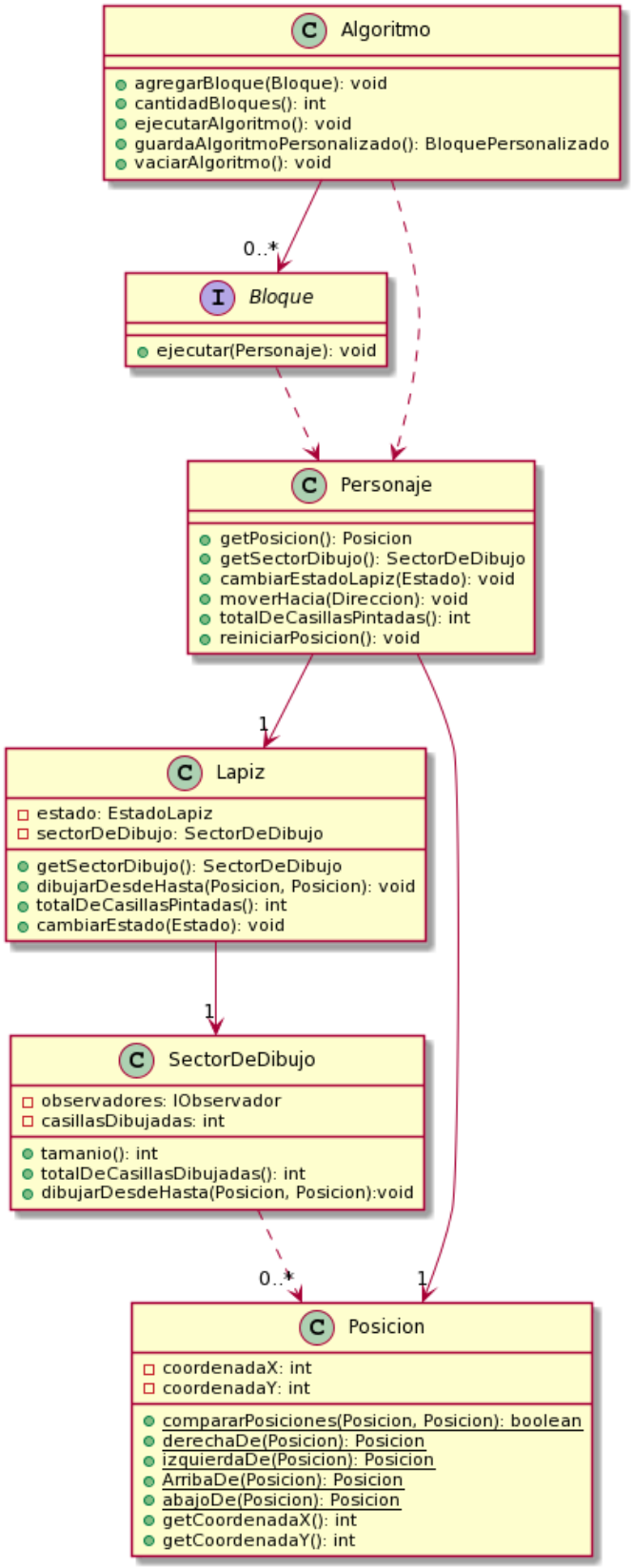


Figura 6: Diagrama de Algortimo y su relacion con Personaje

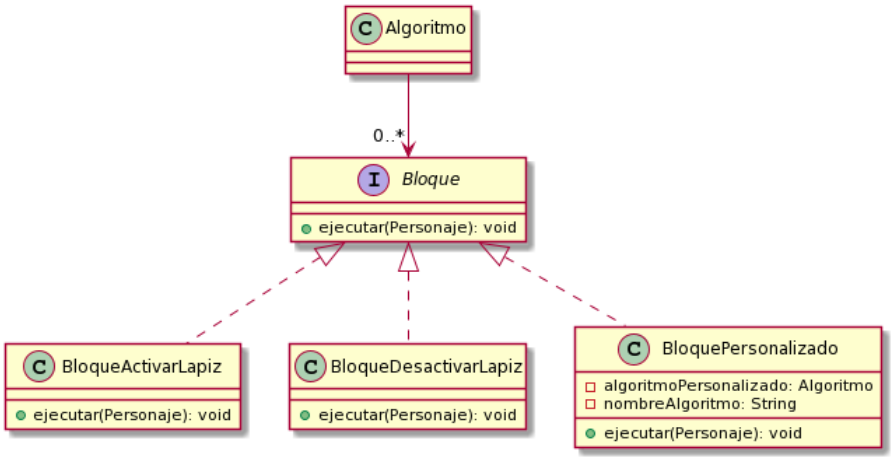


Figura 7: Diagrama de algoritmo y bloque personalizado, y bloque de activar y desactivar lapiz.

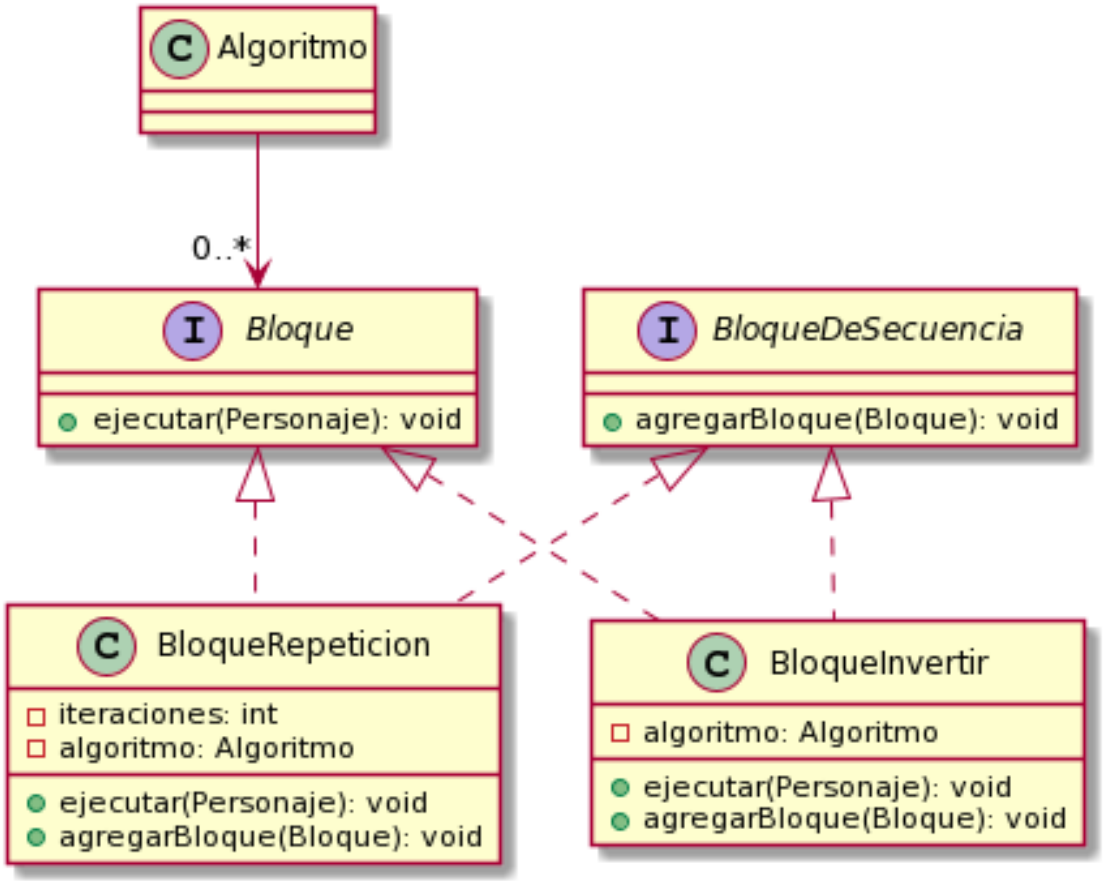


Figura 8: Diagrama de bloques de secuencia

5. Diagramas de Paquetes

Mostramos a continuación un diagrama de paquetes del paquete modelo.

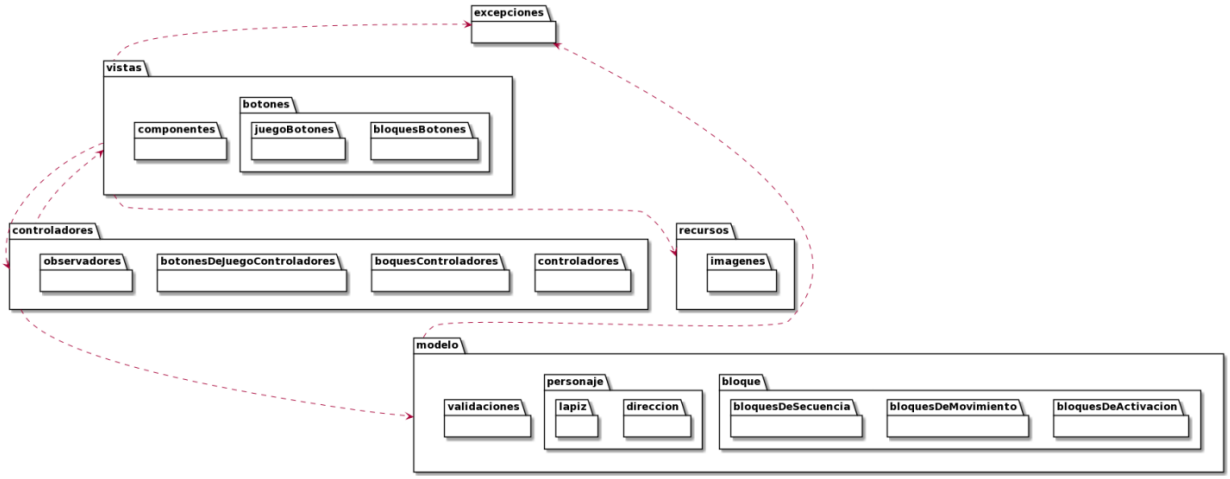


Figura 9: Diagrama de paquetes del modelo

6. Detalles de implementación

6.1. Bloques

Los bloques son una parte crucial del modelo diseñado, en total fueron 12 las implementaciones necesarias entre interfaces, clases y clases abstractas para poder modelar los diferentes tipos de comportamientos.

Los métodos que todos tienen en común son ejecutar e inverso:

* El primero se encarga de ejecutar el bloque, ya sea la acción que este indica o en caso de los bloques de secuencia y el bloque personalizado, de ejecutar los bloques que contiene.

* Con inverso sucede algo similar, el bloque al que se llama devuelve su inverso, ya sea esto un bloque con el efecto opuesto o un algoritmo invertido.

Los bloques de secuencia tienen ademas un método que les posibilita guardar más bloques en su algoritmo.

6.2. Personaje

El personaje al ser instanciado, crea un nuevo lápiz y una nueva posición ubicada en la coordenada (0,0). Luego durante la ejecución del juego delegará responsabilidades a estas clases.

Los métodos que implementa son moverse hacia una nueva dirección, actualizando su posición, y comunicándole a lápiz que dibuje.

También tiene la responsabilidad de reiniciar la posición cuando el usuario lo pide, reemplazándola por una nueva instancia de la clase posición en las coordenadas (0,0).

6.3. Posición

Posición es a quien personaje delega la responsabilidad de saber cómo actualizarse cuando es llamada por un bloque de movimiento.

Cuenta con dos coordenadas, una en x y la otra en y. En sus métodos van a ser actualizadas independientemente la una de la otra, nunca en simultáneo.

6.4. Dirección

Dirección es una interfaz implementada por cuatro clases, todas comparten un único método que recibe una posición y devuelve la posición que está en la dirección del movimiento que se quiere hacer.

Por ejemplo en la clase DireccionAbajo:

```
@Override
    public Posicion posicionEnDireccionDe(Posicion posicion){
        return Posicion.abajoDe(posicion);
    }
```

6.5. Dirección

Se podría concluir, entonces, que una Direccion solo se encarga de saber cuál mensaje de clase enviarle a la clase Posicion, para después regresarle la nueva posición al personaje y ser actualizada.

6.6. Lápiz

Esta clase se encarga de instanciar un nuevo estado y un nuevo sector de dibujo cuando es instanciada. Su responsabilidad es delegar ciertas acciones a estos, y actualizar sus referencias cuando es debido.

EstadoLapiz

Esta interfaz es implementada por dos clases, quienes implementan el método *dibujarDesdeHasta*, en el caso del lápiz activado dibuja y en el caso de que el lápiz se encuentre desactivado no hace nada.

```
//la clase EstadoActivado
public class EstadoActivado implements EstadoLapiz {


    public void dibujarDesdeHasta(SectorDeDibujo sectorDeDibujo, Posicion posicionVieja, Posicion

        sectorDeDibujo.dibujarDesdeHasta(posicionVieja, posicionNueva);

    }
//la clase EstadoDesactivado
public class EstadoDesactivado implements EstadoLapiz {


    public void dibujarDesdeHasta(SectorDeDibujo sectorDeDibujo, Posicion posicionVieja, Posicion


    }
```

6.7. SectorDeDibujo

Esta clase tiene la responsabilidad de almacenar desde donde hasta donde se deberá dibujar.

Las posiciones desde donde hasta donde se quiere dibujar son guardadas en una tupla, ya que no es deseado que se pueda modificar esta información. Luego estas tuplas son guardadas en una lista, ya que se quiere poder seguir agregando tuplas. Por ende el constructor está compuesto por una lista de tuplas de posiciones.

Tambien cuenta con una lista de los observadores a quienes notificará cuando se dibuje.

```
    public void dibujarDesdeHasta(Posicion posicionVieja, Posicion posicionNueva) {
        Pair<Posicion, Posicion> tupla = new Pair(posicionVieja, posicionNueva);
        this.posicionesDibujadas.add(tupla);
        notificarObservadores();
    }
```

6.8. Algoritmo

Algoritmo está conformado por los bloques que el usuario desea usar. Su composición cuenta con una lista de todos esos bloques, y sus métodos son similares a los ya nombrados.

Por un lado se puede ejecutar un algoritmo, este método recorre la lista de bloque y va pidiendole a cada uno que se ejecute. Como está la posibilidad de que la lista de bloques esté vacía y se quiere que se ejecute igual cuenta con un try/catch para atrapar esa excepción.

```
public void ejecutarAlgoritmo(Personaje unPersonaje) {
                try{
                ValidaAlgoritmo.algoritmoValido(this.cantidadBloques());
                    }
                catch(AlgoritmoVacioError algoritmoVacioError) {
            throw algoritmoVacioError;
        }
                for (Bloque unBLoque : secuenciaBloques) {
```

```
                unBLoque.ejecutar(unPersonaje);
            }
        }
```

Dispone de otros métodos, como invertir, que lo que hace es pedirle a cada bloque se la lista que devuelva su inverso. También vaciar que vacía la lista de bloques. Y por último puede guardar un algoritmo personalizado y devolverlo en forma de bloque con el nombre que el usuario haya decidido ponerle.

```
public BloquePersonalizado guardaAlgoritmoPersonalizado(String nombre) {

    try{
        ValidaAlgoritmo.algoritmoValido(this.cantidadBloques());
        ValidaNombre.nombreValido(nombre);
    }catch(AlgoritmoVacioError algoritmoVacioError){
        throw algoritmoVacioError;
    }catch(NombreVacioError nombreVacioError){
        throw nombreVacioError;
    }
    BloquePersonalizado personalizado = new BloquePersonalizado(nombre, secuenciaBloques);
    vaciarAlgoritmo();
    return personalizado;
}
```

6.9. Patrón de Arquitectura de Software MVC

Se diseñó la aplicación siguiendo este patrón para lograr desacoplar la implementación del modelo de la interfaz gráfica, de esta forma se puede reutilizar el modelo si luego queremos usar otra herramienta de vista, ya qué, si el mismo dependiera de JavaFX, esto no podría realizarse, limitando así la usabilidad del modelo.

7. Excepciones

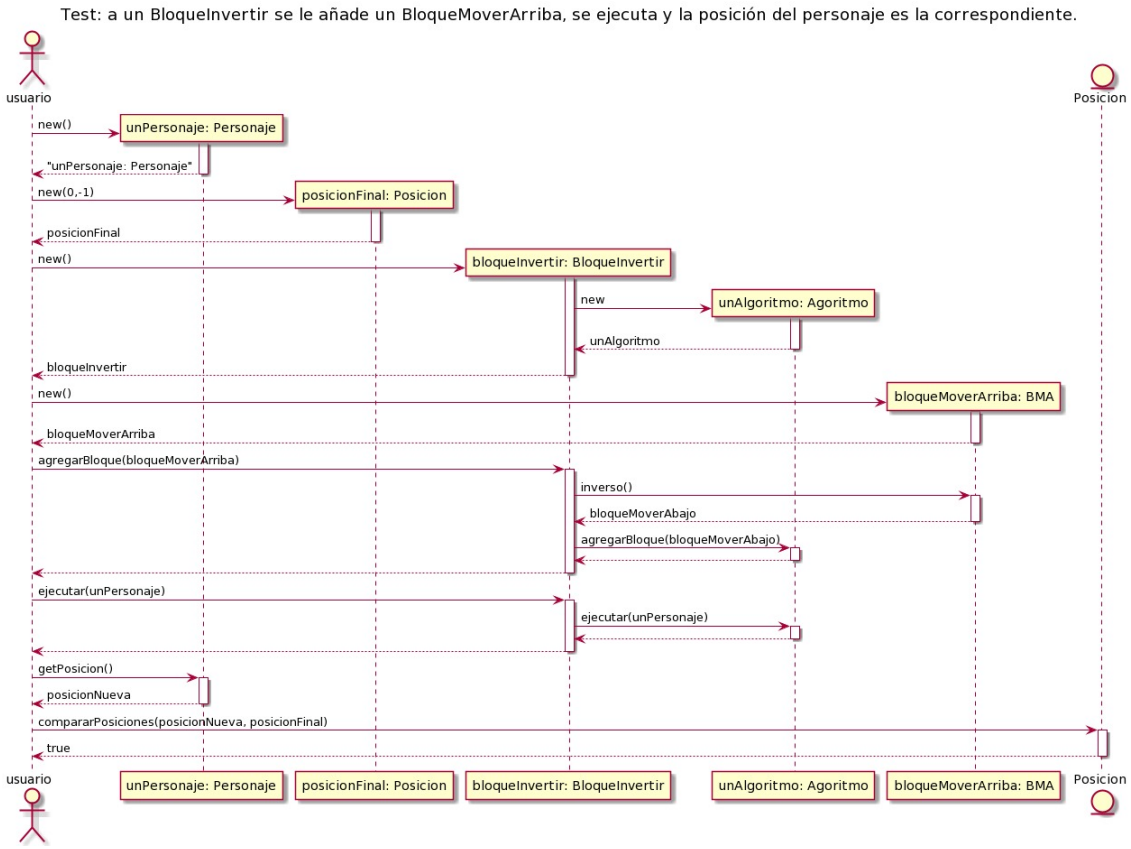
AlgoritmoVacioError Esta excepción esta para indicar que ocurrió un error al querer ejecutar los bloques del sector algoritmo y éste esté vacio. Se lanza un cartel para avisar que se gregue un bloque para ejecutar. La excepcion es lanzada por ControladorEjecutarAlgoritmo y Algoritmo.

NombreVacioError.java Esta excepción es lanzada por la clase Algoritmo, para no dejar que se guarde una secuencia de bloques personalizado sin un nombre que lo identifique. De esta forma se puede seguir jugando y hacer uso del bloque personalizado cuando se lo requiera.

BloquesDeSecuenciaAnidadosDeError Esta excepcion fue creada para evitar que se agregue un bloque de secuencia en el sector algoritmo cuando ya hay un bloque de secuencia iniciado, al lanzarce se despliega un cartel con la recomendacion para realizar la anidacion de los bloques de secuencia. El cartel recomienda que se guarde el bloque de secuencia vigente como bloque personalizado, para despues poder hacer uso del bloque personalizado en el sector algoritmo y poder meterlo dentro de un bloque de secuencia , si asi se requiere.Esta excepcion puede ser lanzada por las clases ControladorBloqueRepetirX2, ControladorBloqueRepetirX3, ControladorBloqueInvertir. Como fue detallado anteriormente, dicha funcionalidad podría haber sido implementada por el modelo, ya que no hay ningún límite al respecto adentro de este. La excepción que se encuentra en el programa surge de la vista.

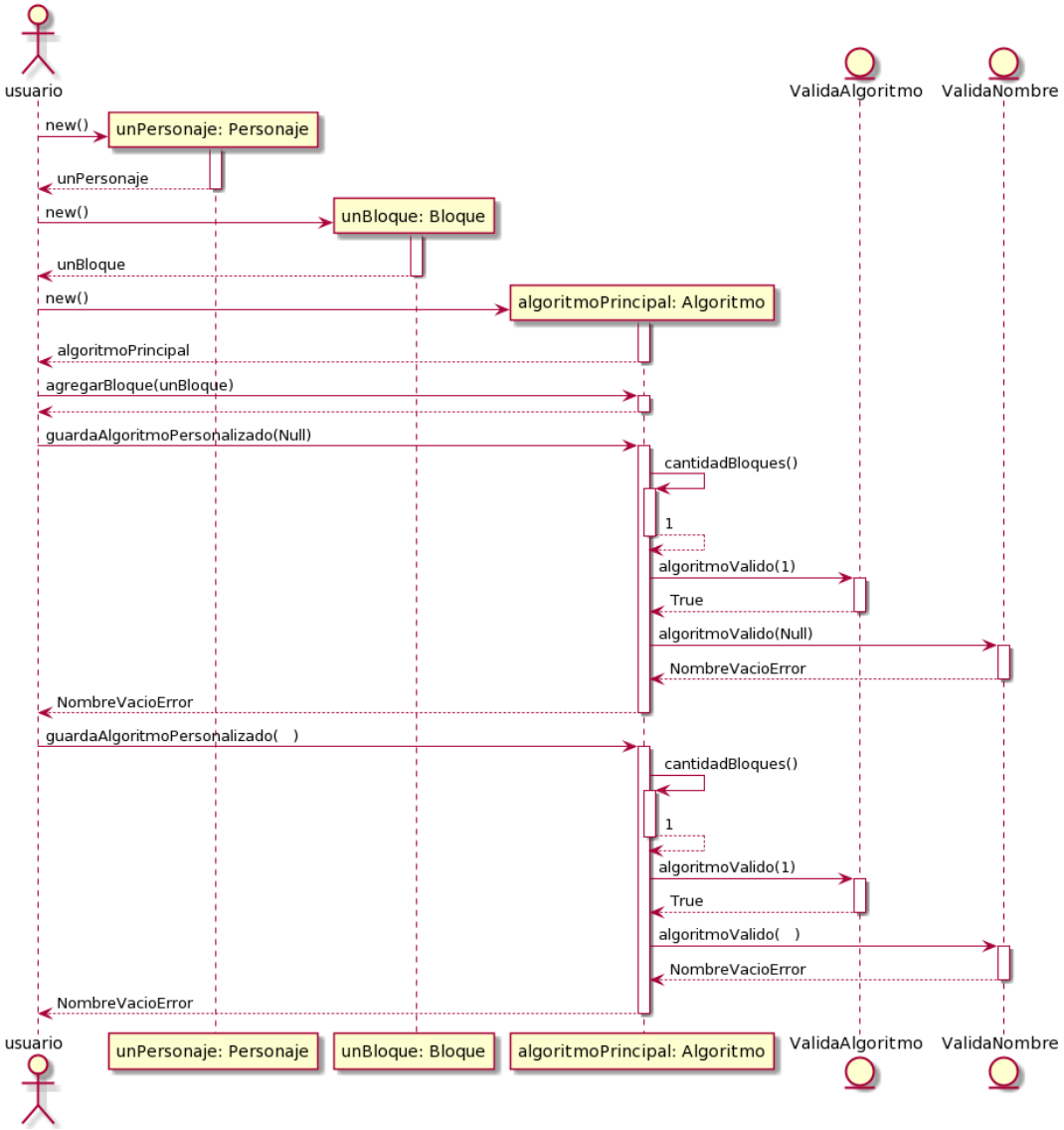
8. Diagramas de secuencia

Se muestra ahora un diagrama de un test.

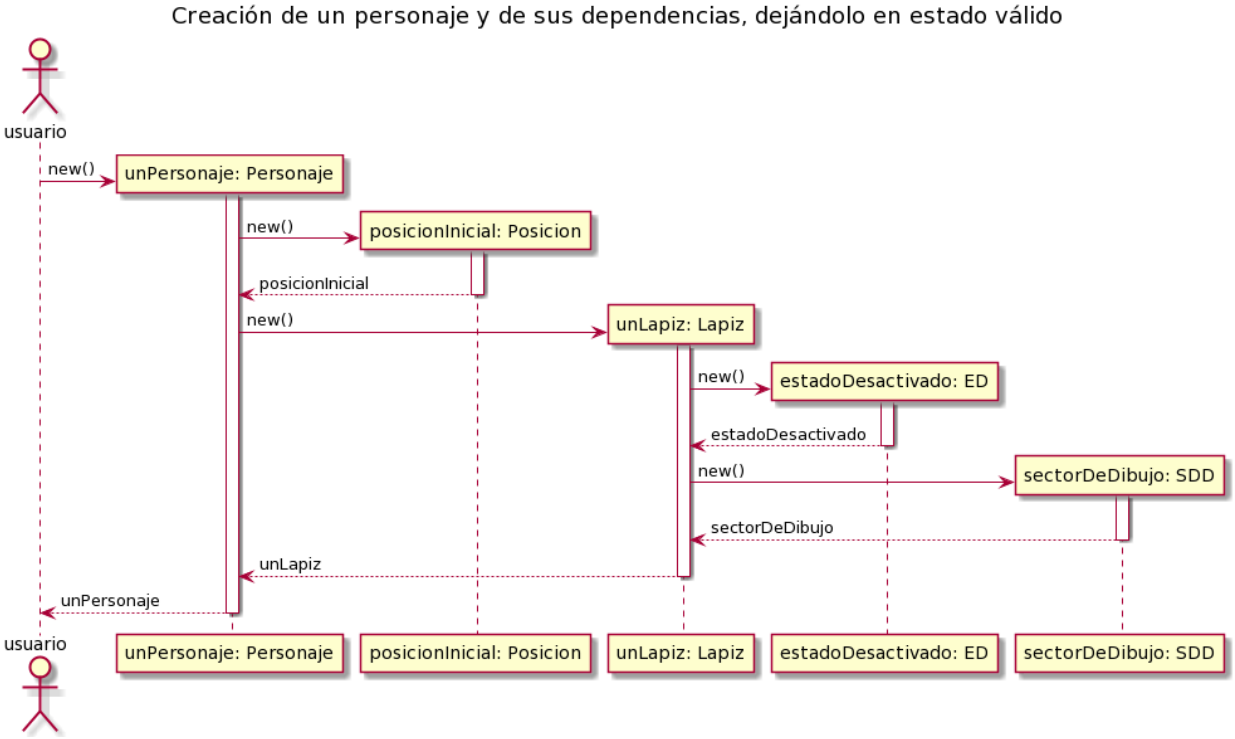


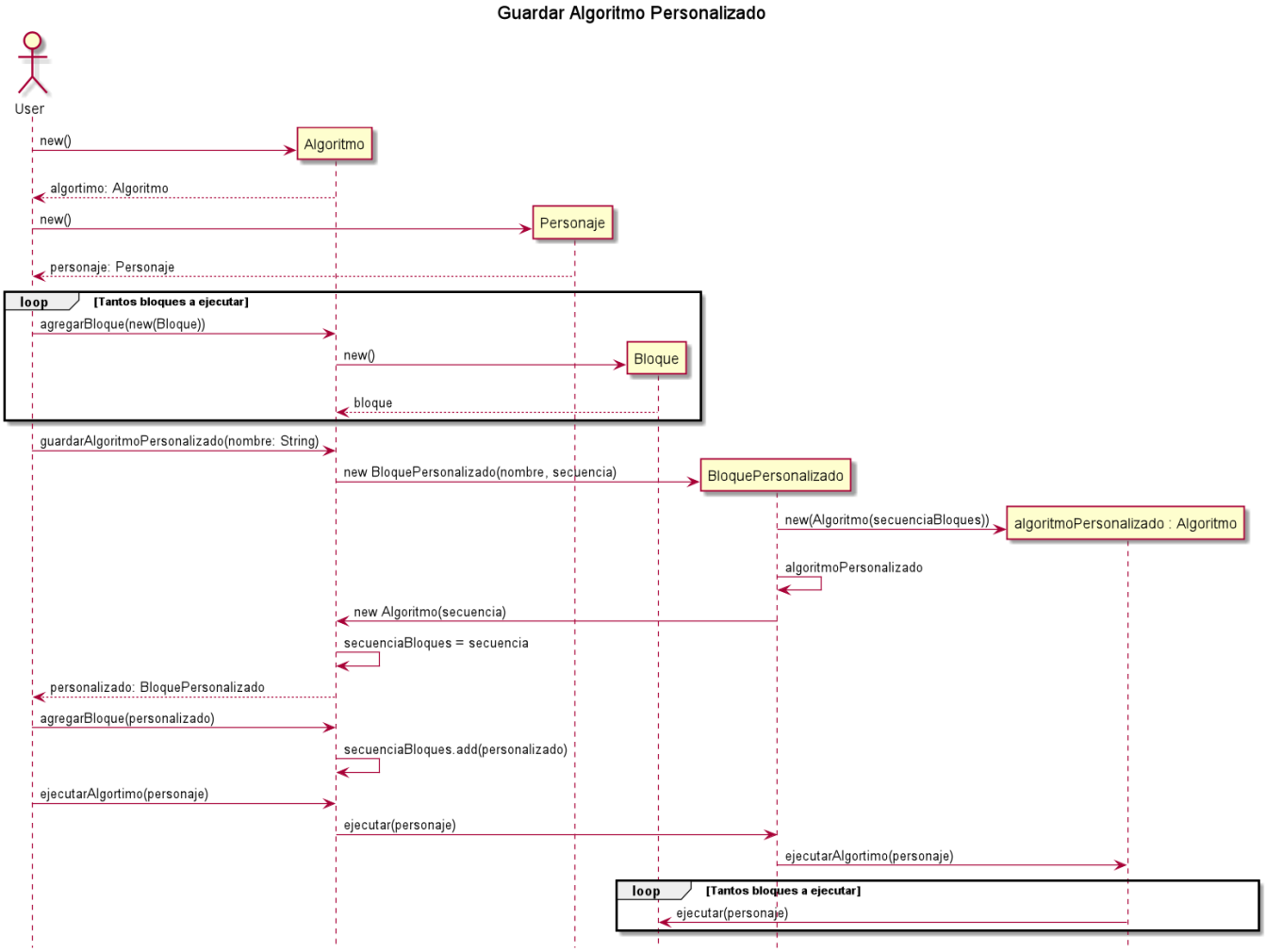
Se muestra un diagrama con las validaciones al querer guardar un bloque personalizado.

Surgen de las dos formas distintas la excepción NombreVacioError al intentar guardar un bloque personalizado



En este diagrama se muestra la creacion de un personaje y sus dependencias


Creación de un personaje y de sus dependencias, dejándolo en estado válido
usuario
new()
unPersonaje: Personaje
new()
posicionInicial: Posicion
posicionInicial
new()
unLapiz: Lapiz
new()
estadoDesactivado: ED
estadoDesactivado
new()
sectorDeDibujo: SDD
sectorDeDibujo
unLapiz
unPersonaje
usuario
unPersonaje: Personaje
posicionInicial: Posicion
unLapiz: Lapiz
estadoDesactivado: ED
sectorDeDibujo: SDD


Guardar Algoritmo Personalizado
User
Algoritmo
Personaje
Bloque
BloquePersonalizado
algoritmoPersonalizado : Algoritmo
new()
algortimo: Algoritmo
new()
personaje: Personaje
loop
[Tantos bloques a ejecutar]
agregarBloque(new(Bloque))
new()
bloque
guardarAlgoritmoPersonalizado(nombre: String)
new BloquePersonalizado(nombre, secuencia)
new(Algoritmo(secuenciaBloques))
algoritmoPersonalizado
new Algoritmo(secuencia)
secuenciaBloques = secuencia
personalizado: BloquePersonalizado
agregarBloque(personalizado)
secuenciaBloques.add(personalizado)
ejecutarAlgortimo(personaje)
ejecutar(personaje)
ejecutarAlgortimo(personaje)
loop
[Tantos bloques a ejecutar]
ejecutar(personaje)

9. Diagramas de Estados

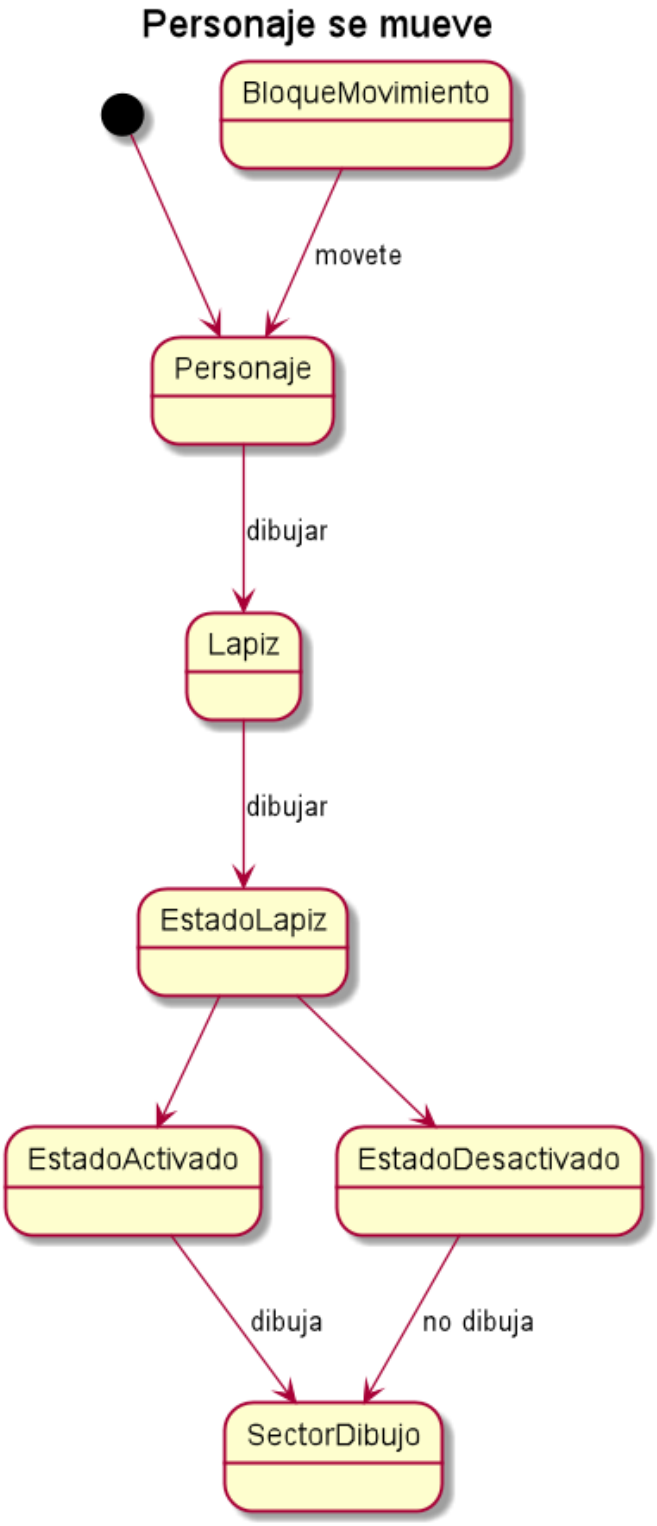


Figura 10: Diagrama de estados que muestra cuando se mueve el personaje

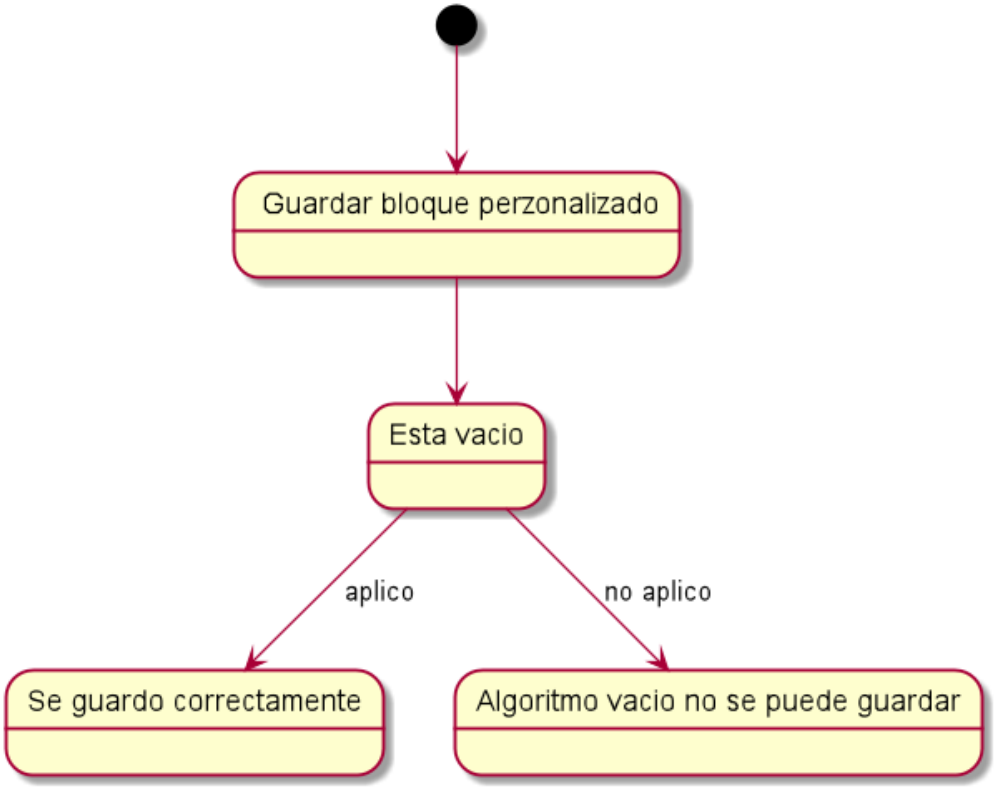


Figura 11: Diagrama mostrando la secuencia en caso de que el usuario decida guardar un bloque personalizado