Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Curso 2 - Grupo 5
Segundo Cuatrimestre de 2020

Alumno:	Filipovskis	Nieva	Aceval	Saavedra
Número de padrón:	105231	103614	104082	104302

Índice

1. Introducción

Este informe contiene la documentacion de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar un juego que permite aprender los conceptos básicos de programación, armando algoritmos utilizando bloques visuales. Usando todos los conceptos vistos en el curso, utilizando un lenguaje de tipado estático (Java) con un diseño del modelo orientado a objetos y trabajando con las técnicas de TDD e Integración Continua vistos hasta ahora en el curso.

2. Supuestos

El personaje cambiará de lado en el dibujo si pasa sus límites: El dibujo está delimitado por un área en la cuál siempre se encontrará el personaje, y cuando este sobrepase los límites, aparecerá por el lado contrario a donde se encontraba.

El botón reiniciar, reinicia todo el juego: No pueden borrarse de manera individual las posiciones que ya hayan sido dibujadas. La funcionalidad soporta borrar todo el dibujo y el algoritmo, al mismo tiempo, incluyendo bloques de algoritmos personalizados.

El botón borrar, borra el algoritmo: La forma de eliminar los bloques de un algoritmo soportada es removiendo el algoritmo entero.

Los bloques se agregan haciendo click sobre el bloque

Un algoritmo vacío no puede guardarse: Sólo se generará un bloque personalizado en caso de que se presione el botón de guardar algoritmo y éste último ya posea bloques.

3. Diagramas de clases

3.1. Modelo

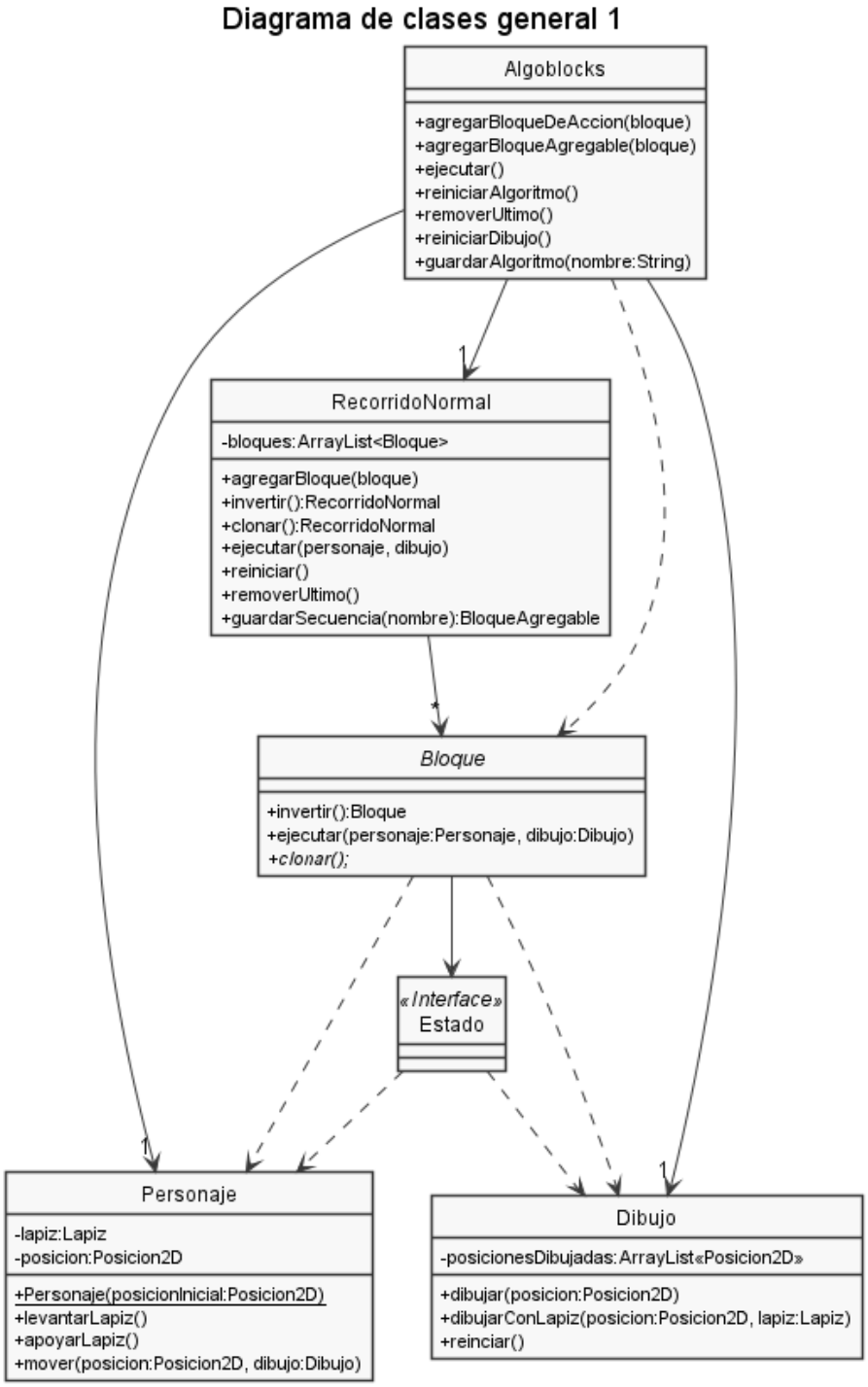


Figura 1: La clase Algoblocks recibe mensajes y luego delega según corresponda. Está compuesta por RecorridoNormal, un Dibujo y un Personaje, principalmente.

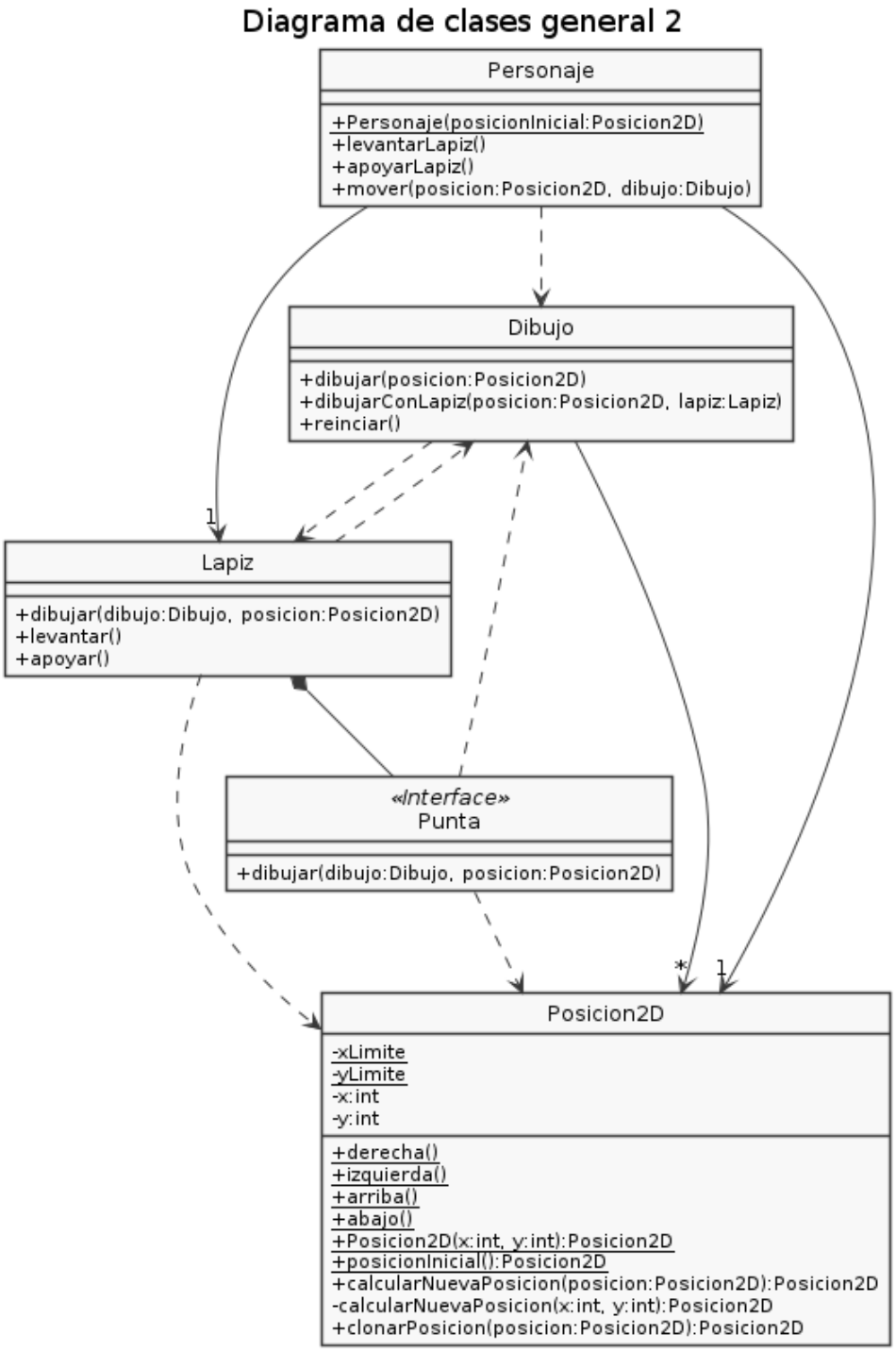


Figura 2: Por el lado del personaje, este tiene una posición, que se indica con la clase Posicion2D, también tiene un Lapiz que tendrá un tipo de punta concreta (Ver más adelante en la figura 7). El personaje a su vez conoce al Dibujo en el cual dibujará sus movimientos.

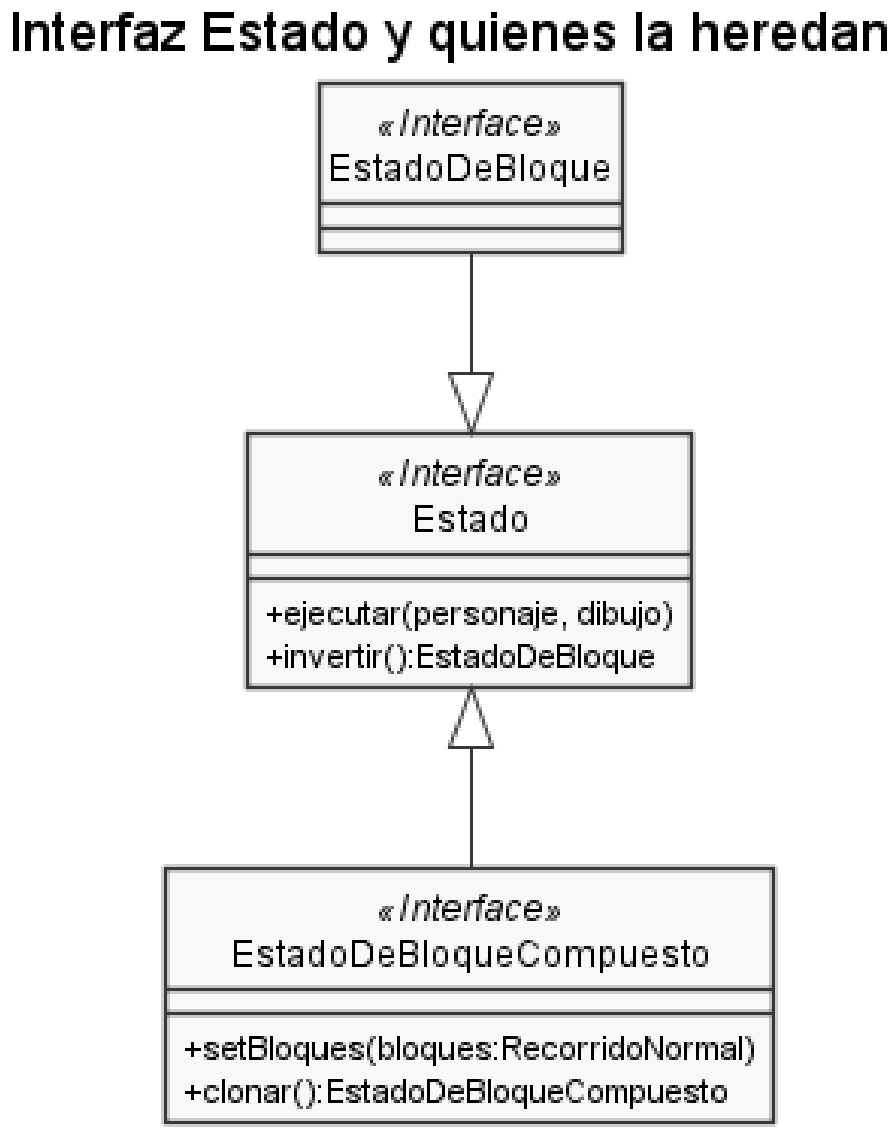


Figura 3: Relación entre las clases Estado, EstadoDeBloque y EstadoDeBloqueCompuesto.

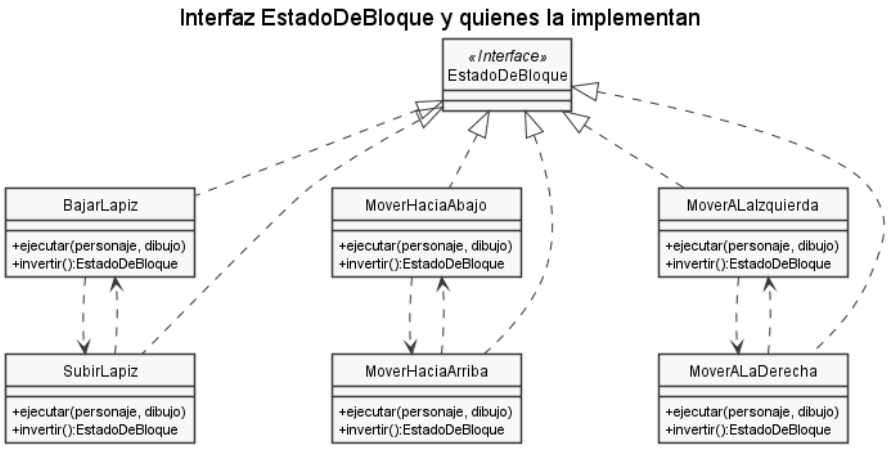


Figura 4: Clases que implementan la interfaz EstadoDeBloque, ideada como un patrón State, el cual se detalla en el apartado 6.2.

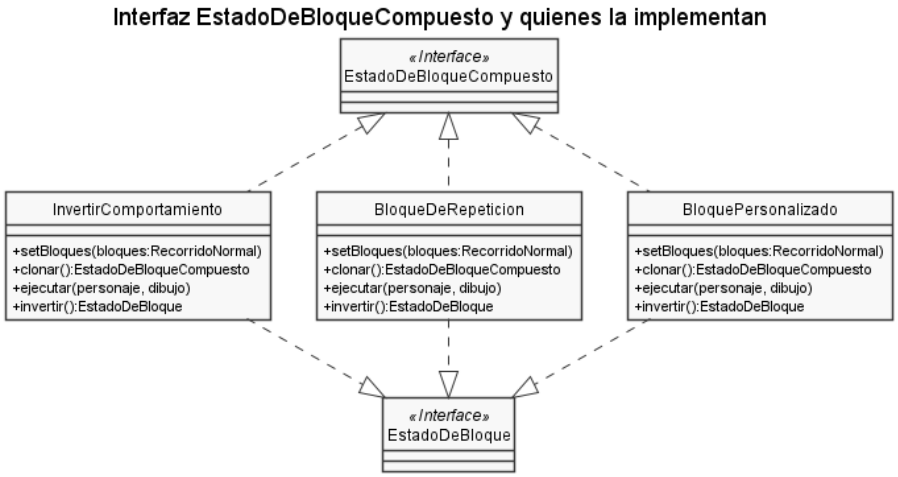


Figura 5: Clases que implementan la interfaz EstadoDeBloqueCompuesta

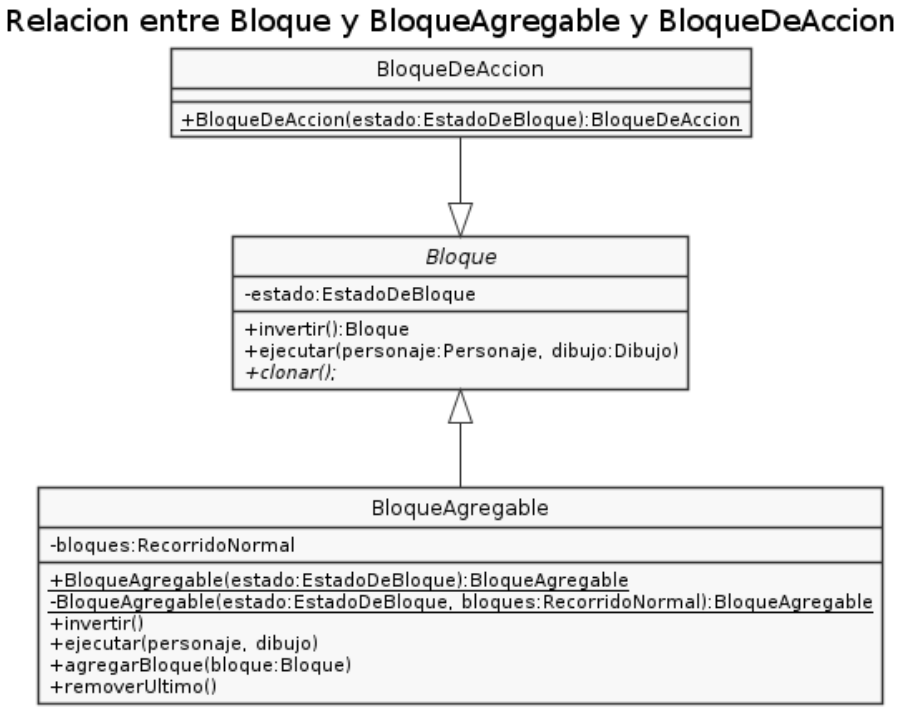


Figura 6: Clases que heredan de Bloque.

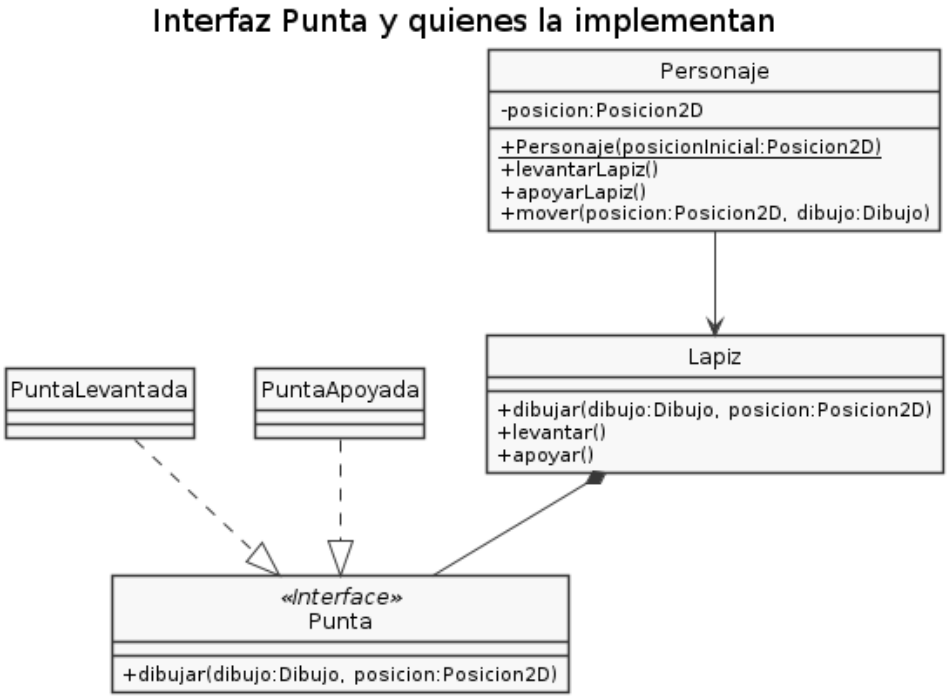
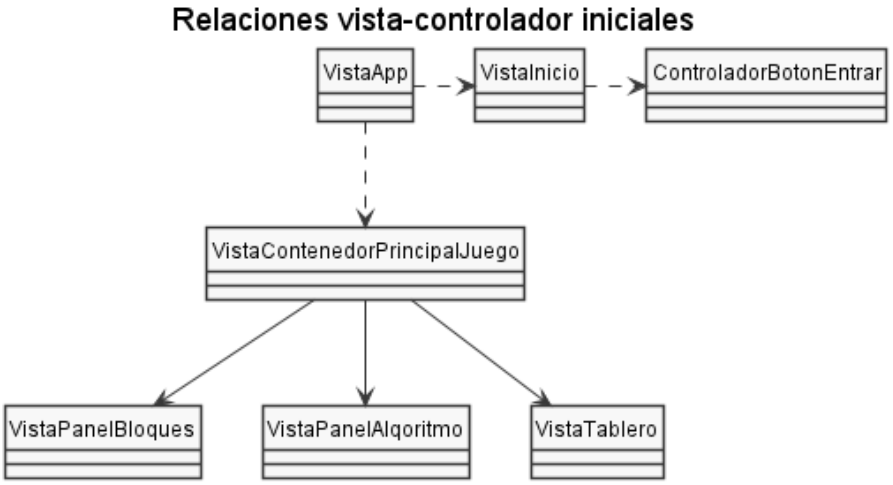
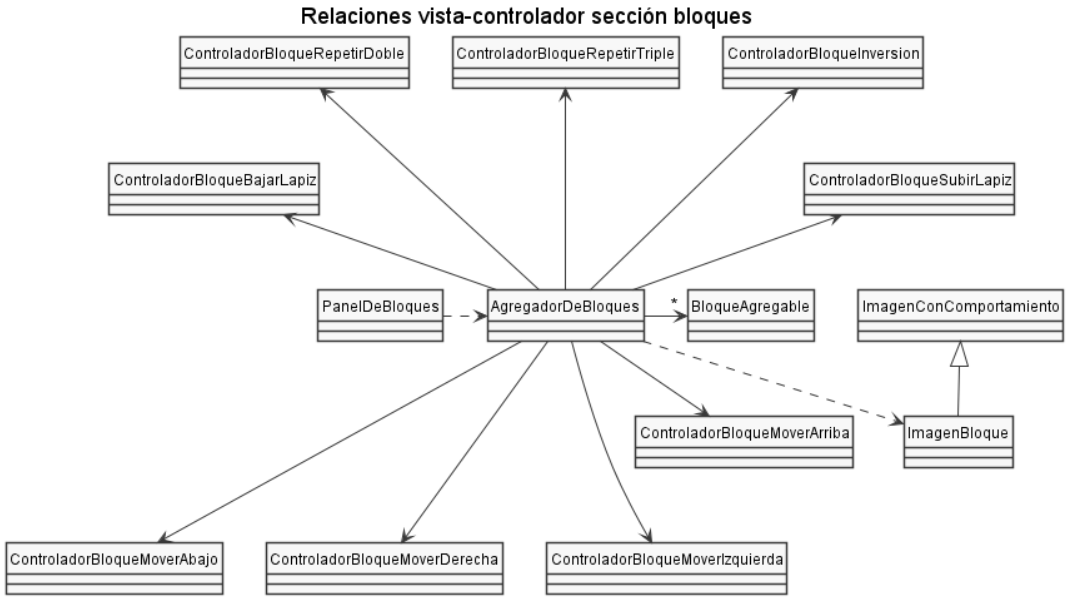


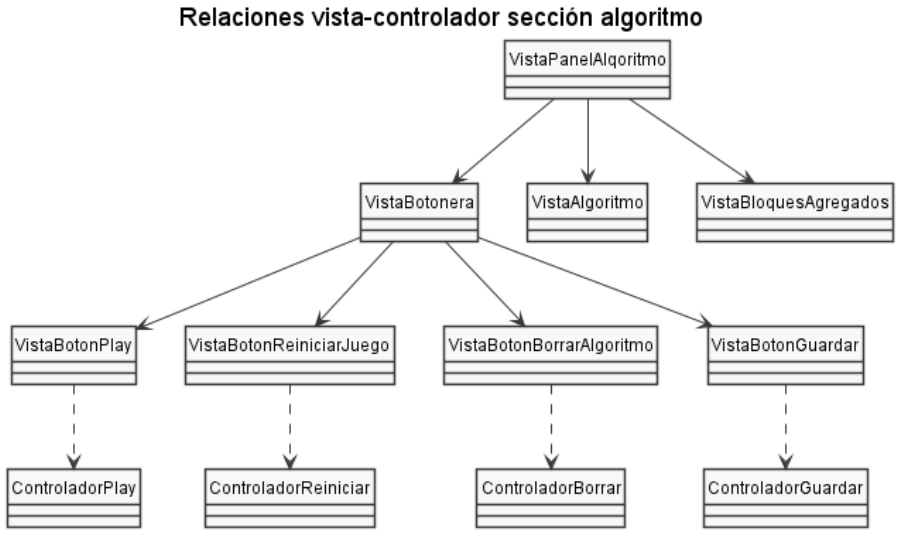
Figura 7: Las clases concretas PuntaLevantada y PuntaApoyada, junto con la interfaz Punta, son parte de un patrón Strategy.

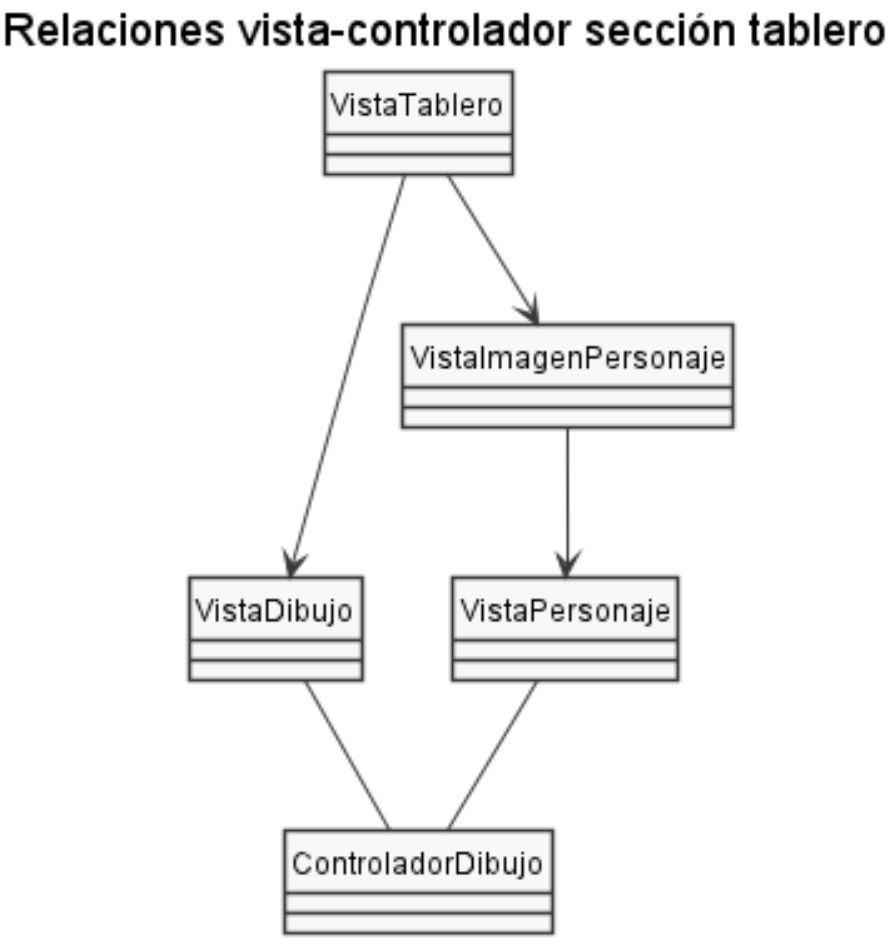
3.2. Vista-Controlador

Los diagramas Vista-Controlador a continuación, muestran un panorama de cómo se componen las vistas, y cuáles son sus respectivos controladores.

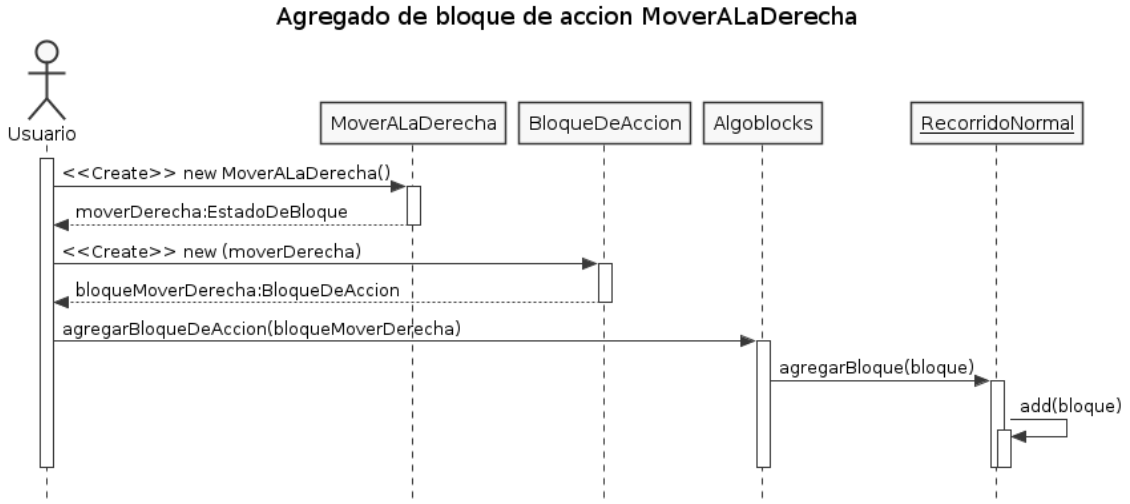


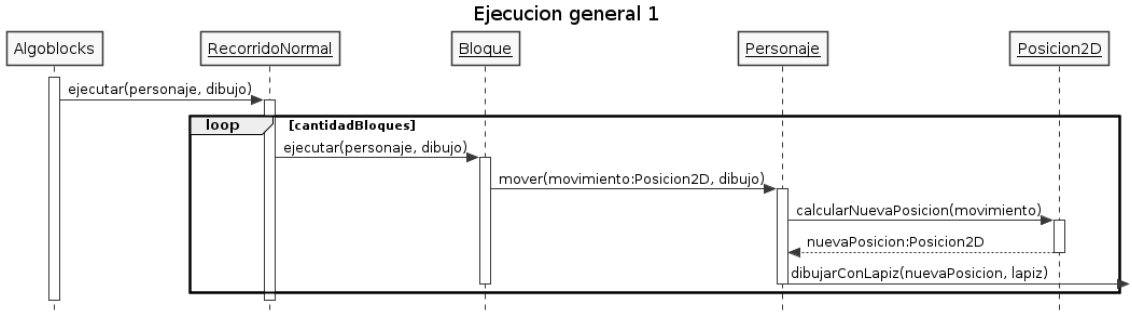


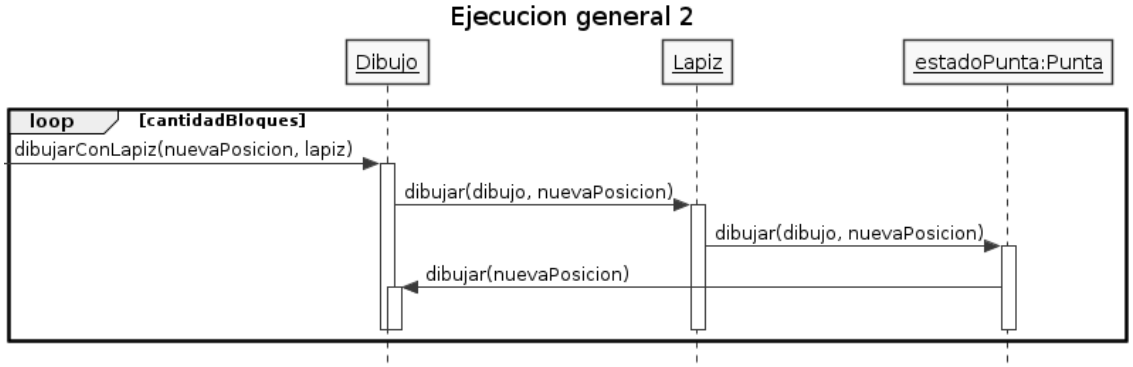

Relaciones vista-controlador sección algoritmo
VistaPanelAlgoritmo
VistaBotonera
VistaAlgoritmo
VistaBloquesAgregados
VistaBotonPlay
VistaBotonReiniciarJuego
VistaBotonBorrarAlgoritmo
VistaBotonGuardar
ControladorPlay
ControladorReiniciar
ControladorBorrar
ControladorGuardar

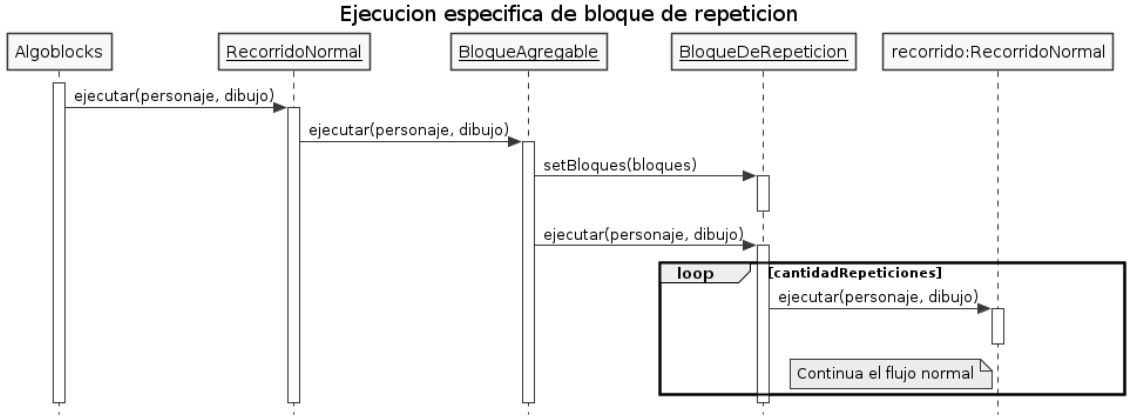

Relaciones vista-controlador sección tablero
VistaTablero
VistaImagenPersonaje
VistaDibujo
VistaPersonaje
ControladorDibujo

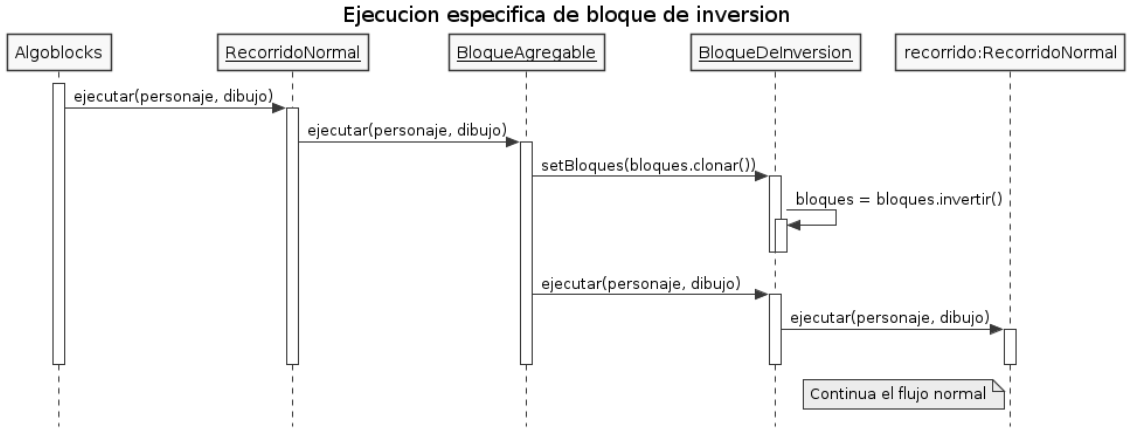
4. Diagramas de secuencia

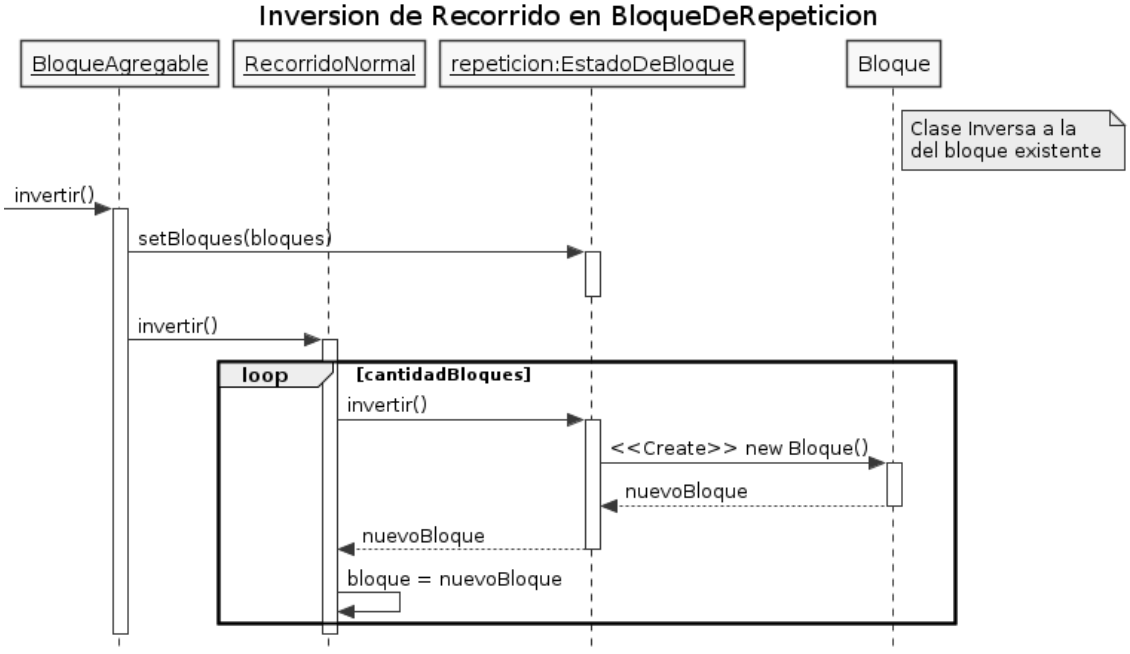


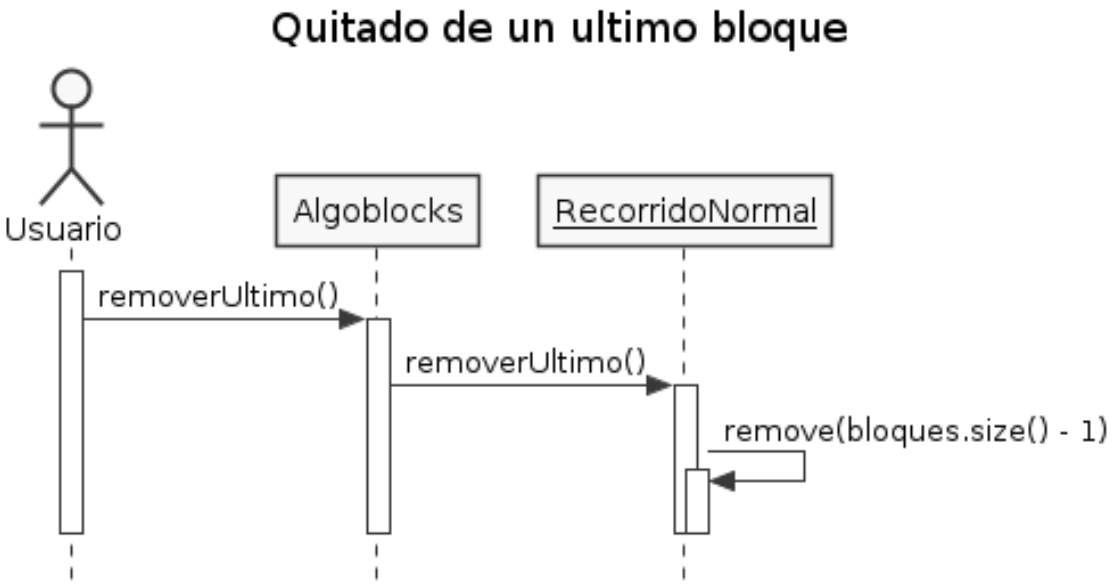





Ejecucion especifica de bloque de repeticion
Algoblocks
RecorridoNormal
BloqueAgregable
BloqueDeRepeticion
recorrido:RecorridoNormal
ejecutar(personaje, dibujo)
ejecutar(personaje, dibujo)
setBloques(bloques)
ejecutar(personaje, dibujo)
loop
[cantidadRepeticiones]
ejecutar(personaje, dibujo)
Continua el flujo normal


Ejecucion especifica de bloque de inversion
Algoblocks
RecorridoNormal
BloqueAgregable
BloqueDeInversion
recorrido:RecorridoNormal
ejecutar(personaje, dibujo)
ejecutar(personaje, dibujo)
setBloques(bloques.clonar())
bloques = bloques.invertir()
ejecutar(personaje, dibujo)
ejecutar(personaje, dibujo)
Continua el flujo normal


Inversion de Recorrido en BloqueDeRepeticion
BloqueAgregable
RecorridoNormal
repeticion:EstadoDeBloque
Bloque
Clase Inversa a la del bloque existente
invertir()
setBloques(bloques)
invertir()
loop
[cantidadBloques]
invertir()
<<Create>> new Bloque()
nuevoBloque
nuevoBloque
bloque = nuevoBloque



5. Diagramas de paquetes

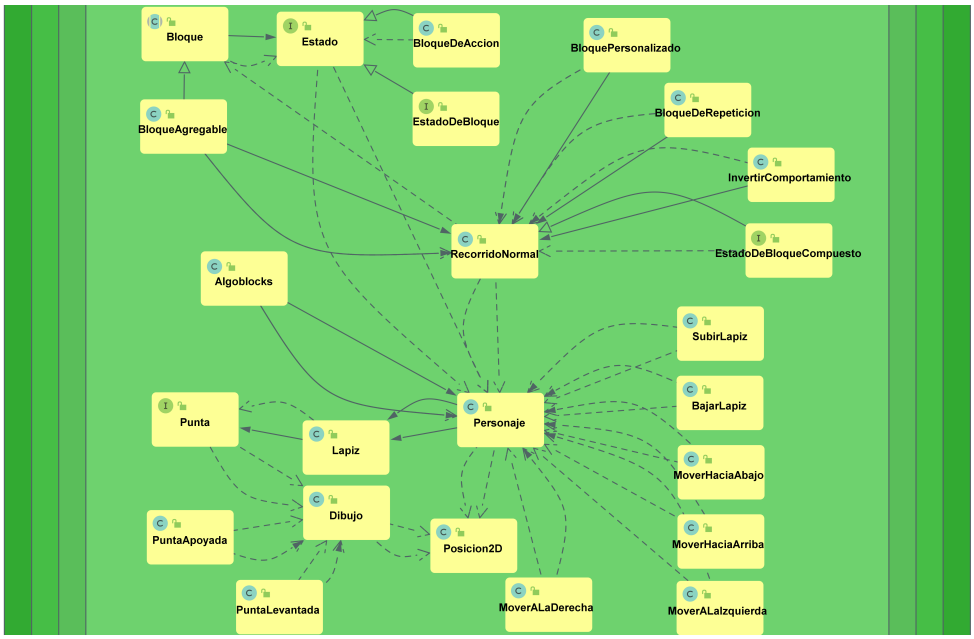


Figura 8: Diagrama del paquete edu.fiuba.algo3.modelo

6. Detalles de implementación

6.1. Estrategias utilizadas para puntos conflictivos

Manejo de posición del personaje: El objetivo principal de plantear la posición de un personaje de la manera realizada, es obtener posiciones genéricas y modificables a futuro. En

principio se tiene un personaje que tiene una posición en dos dimensiones, representada con la clase llamada Posicion2D. Las posiciones de dicha clase se ven afectadas únicamente por objetos que implementan EstadoDeBloque. Estos se encargarán de instanciar posiciones que representen movimientos, y que al ser enviadas a la posición del personaje, mediante el método mover(), permitan que se determine la nueva posición.

De la clase MoverALaDerecha, que implementa a EstadoDeBloque:

```
public void ejecutar(Personaje personaje, Dibujo dibujo) {
        personaje.mover(Posicion2D.derecha(), dibujo);
    }
```

'derecha()' es un método de estático de la clase Posición2D que devolverá aquella posición que represente un movimiento a la derecha.

De la clase Personaje:

```
public void mover(Posicion2D posicion) {
    Posicion2D nuevaPosicion = this.posicion.calcularNuevaPosicion(posicion);
    ...
    this.posicion = nuevaPosicion;
}
```

De la clase Posicion2D:

```
public Posicion2D calcularNuevaPosicion(Posicion2D posicion) {
    return posicion.calcularNuevaPosicion(this.x, this.y);
}

private Posicion2D calcularNuevaPosicion(int xActual, int yActual) {
    int nuevaX = (this.x + xActual + xLimite) % (xLimite);
    int nuevaY = (this.y + yActual + yLimite) % (yLimite);
    return new Posicion2D(nuevaX, nuevaY);
}
```

Delegación de mensajes de Algoblocks a Tablero: En la primer iteración se pensó en la clase Tablero para delegar todas las funciones que correspondían al Dibujo, sin embargo con esta implementación había un alto acoplamiento entre las clases Tablero y Dibujo. El alto acoplamiento derivó en que Tablero sea una clase anémica, sin comportamiento real. Debido a esto, se eliminó y Algoblocks pasó a tener la referencia al dibujo.

Representación del algoritmo: Inicialmente fue mediante una clase ColaDeInstrucción, donde en conjunto con la clase Recorrido, se diferenciaban si era normal o invertido y se ejecutaba la cola, pero esto se complicó al añadir la funcionalidad de eliminar último, entonces se eliminó y en cambio usamos una lista de Bloques directamente en el Recorrido.

Instanciar bloques dentro o fuera de Algoblocks: Al inicio del desarrollo se planteó crear mensajes en Algoblocks para cada uno de los bloques que se podían agregar, siendo cada tipo de bloque una clase, pero luego al poder identificar los tipos de comportamientos de los bloques, se generalizó para dos tipos, los de acción y los agregables, y a través de una refactorización hacer que al agregar un bloque, se pase la instancia del mismo.

Aplicación de herencia en la clase Bloque: Luego de plantear a la clase Bloque como una interfaz que era implementada por los diferentes tipos de bloques disponibles, se vio una fuerte relación entre la interfaz y sus implementadores. Las cuestiones que hicieron evidente este asunto fueron la reusabilidad completa de la interfaz en las clases que la implementaban,

y la existencia de atributos comunes en todas ellas. Esto último también ocasionaba repetición de código en diferentes clases, causada por la imposibilidad de establecer un atributo común a nivel interfaz. Dada esta observación, se decidió usar clases hijas que heredaban de Bloque en vez de clases que implementaban a la interfaz.

6.2. Patrones de diseño utilizados

Singleton: En la clase Algoblocks, siendo esta la única forma de acceder a la lógica del juego.

Strategy: En las clases PuntaApoyada, PuntaLevantada e interfaz Punta (figura 7), para que el Personaje siempre que dibuje con el lapiz, dependa de la clase de punta cómo dibuja.

State: En las clases que representan los bloques y la interfaz EstadoDeBloque (figura 4), para cuando el invertir tenga que cambiar qué bloque se va a ejecutar.

Observer: En las clases de la figura 9, donde la VistaImagenPersonaje le envía al Personaje el addProperty y este a su vez le envía el fireProperty a su support, así en los cambios de estado, puede avisar sus listeners, pasando los valores viejos y nuevos, entonces la VistaPersonaje puede actualizar lo que necesita dentro del propertyChange(event).

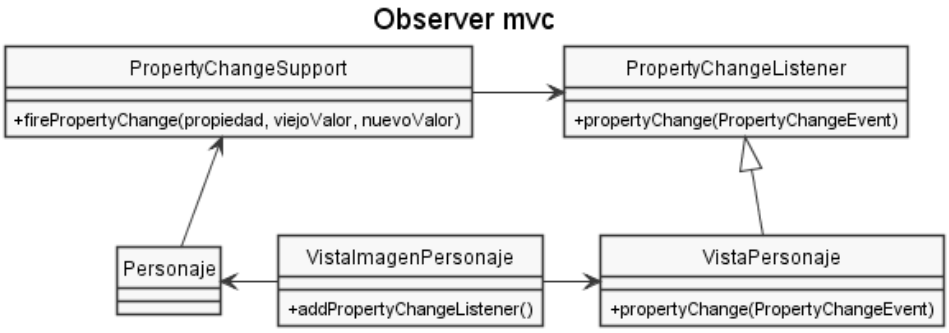


Figura 9: Diagrama de algunas clases involucradas en el patrón Observer.