FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2

(95.02 - 75.07) ALGORITMOS Y PROGRAMACIÓN III

CURSO 02 (NOCHE)

Segundo cuatrimestre 2020

Corrector:

Tomás Bustamante

Alumnos:

Bucciarelli, Trinidad (105494)

Schipani, Martin (100629)

Índice

Objetivo

Desarrollar una aplicación de manera grupal aplicando 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.

Introducción

La aplicación consiste en un juego que permite aprender los conceptos básicos de programación, armando algoritmos utilizando bloques visuales.
Los algoritmos permitirán a un personaje moverse por la pantalla mientras dibuja con un lápiz, logrando realizar distintos diseños.

La aplicación se compondrá de 3 secciones:

- **Tablero**: el tablero comenzará siendo un espacio en blanco en el cual se colocará al personaje en su posición inicial.
- **Lista de bloques**: la lista de bloques mostrará los bloques que el jugador tendrá disponibles para utilizar.
- **Algoritmo**: el algoritmo comenzará en blanco, y el jugador podrá ir colocando bloques en esta sección, que luego se ejecutarán en forma secuencial.

Una vez armado el algoritmo, el jugador podrá elegir “ejecutar” el mismo. En ese caso, cada bloque se irá procesando en forma secuencial, haciendo que el personaje realice la acción correspondiente.
El objetivo del juego es permitir al jugador experimentar con distintos algoritmos para ir aprendiendo los conceptos básicos de la programación.

Supuestos

El juego da la posibilidad de repetir e/o invertir secuencias de movimientos (bloques de movimientos). Dentro de estos bloques se podrán contener bloques de su mismo tipo. Es decir, los bloques de repetición e inversión podrán contenerse a sí mismos. Entonces, en una secuencia que se decida invertir o repetir también se podrá agregar internamente un bloque de inversión o repetición.
Por otra parte, se permite agregar bloques de movimiento sin un límite de cantidad.

Detalles de implementación

Pilares del paradigma orientado a objetos utilizados

Abstracción y Encapsulamiento

Cada objeto creado contiene cumple con una responsabilidad específica. A su vez, los objetos se relacionan entre sí, por lo que, la delegación de responsabilidades se encuentra en métodos de todos los objetos. Esto permite una mayor flexibilidad del código ante posibles cambios en el futuro. Se pretende que cada clase tenga una única razón de cambio, lo cual hace que el programa sea más adaptable a cambios dentro de esos objetos o agregación de funcionalidades dentro del programa.

Polimorfismo

- Bloque posee el mensaje ejecutar(personaje : Personaje) que es implementado en todas las clases que son bloques de movimientos. (BloqueIzquierda, BloqueDerecha, BloqueArriba, BloqueAbajo, LapizArriba, LapizAbajo, y la clase abstracta BloqueIterativo con las subclases BloqueRepetir y BloqueInverso).
- La interfaz Lápiz con el mensaje dibujar(array : int[ ]) es implementado en LapizArriba y LapizAbajo.
- La clase abstracta BloqueIterativo con los menajes recorrer(personaje : Personaje) y agregarBloque() tiene dos subclases: BloqueRepetir y BloqueInverso.

El uso de interfaces y herencia permite la reutilización de mensajes simplificando la lectura del código. Los objetos que implementan las interfaces son capaces de entender un mismo mensaje sin la necesidad de saber la diferencia en la implementación de cada objeto.
De modo tal que, los objetos que compartan la implementación, sea a través de una interfaz o sean clases hermanas, pueden ser tratados de la misma manera al ser capaces de entender un mismo mensaje. Por estos motivos, consideramos que dentro de nuestro modelo resulta conveniente el uso del polimorfismo en estos objetos, funcionalmente comparten mismos comportamientos con sus implementaciones particulares. Y también teniendo en cuenta que la partición de comportamiento en interfaces nos permite una mayor flexibilidad para que cada objeto implemente sólo aquellos métodos que necesita y no se vea forzado a depender de métodos que no utiliza.

Herencia

- La clase abstracta BloqueIterativo y sus subclases BloqueRepetir y BloqueInverso.

En este caso se decidió el uso de herencia porque, además de cumplir la relación “es un”, hay código que se repite no sólo en métodos sino también en atributos.

Patrones de diseño empleados

Strategy

La clase AlgoBlocks se dedica en sus métodos a instanciar un Bloque específico que se acumula en una lista. Posteriormente cuando se recorre la lista, se ejecuta los métodos de ejecutar(personaje : Personaje) pertenecientes a las clase que tienen implementado a la interfaz Bloque.

Singleton

Se encuentra implementado en la clase SectorDibujo que corresponde a la clase donde se guardan los puntos que se dibujan, y estos se trazan. La razón de decidir utilizar este patrón en la clase se debe a que en nuestro modelo sólo debe haber un único SectorDibujo y de esa manera evitar que se puedan instanciar más.

Composite

Se implementa en BloqueIterativo (esto incluye a sus subclases: BloqueInverso y BloqueRepetir). La responsabilidad en esta clase es acumular una lista de bloques de movimientos y poder repetir o invertir una secuencia de movimientos. Entonces, las listas que poseen estos bloques pueden contener objetos simples (bloques simples como BloqueArriba, BloqueDerecha, etc.) y objetos compuestos (como ellos mismos, bloques que contienen otros bloques), y son tratados de la misma manera sin hacer diferencia entre ellos. La ejecución de este algoritmo da como resultado un árbol.

State

La responsabilidad de escribir o no escribir dependiendo de si el lápiz está arriba o abajo, está delegada a una interfaz Lápiz que la implementan dos clases concretas LapizArriba y LapizAbajo, la interfaz tiene el método trazar(Posición inicial, Posición final) y en la clase LapizAbajo se trazará el segmento mientras que en la clase LapizArriba no. La clase Personaje tiene un estado del lápiz el cual es una instancia de LapizArriba o LapizAbajo. Este estado cambia según el lápiz que se elija. A su vez, a partir de la instancia de lápiz que tenga el Personaje se delega la tarea de escribir mediante el método dibujar().
Se decidió aplicar este patrón ya que el problema de escribir o no escribir según el lápiz que se tenga implica un cambio en el estado interno de una clase (Personaje en nuestro caso) dependiente de otros objetos (aquellos que implementen la interfaz Lápiz)

Interfaz gráfica

La implementación respeta el patrón MVC. El diseño de la interfaz no afecta al modelo. El modelo se hace independientemente de la implementación de la interfaz gráfica y la interfaz gráfica no afecta a este.

En la interfaz gráfica se encuentran:

En la paquete Vista:

- La botonera es un VBox con los botones para realizar cada movimiento.
- El tablero donde se verá dibujado los movimientos seleccionados por el usuario. La clase Tablero extiende de Canvas. Ante la interacción mediante el botón Ejecutar el Tablero se actualizará a través de VistaTablero trazando las líneas según corresponda a la clase SectorDibujo.
- ContenedorAlgoritmo: esta clase extiende de GridPane. Se irá actualizará a partir de VistaAlgoritmo. Ante la presión de algún botón de movimiento se insertará en él una imagen que representa el movimiento seleccionado.
- La clase Juego inicializa tanto el modelo como los contenedores y todo lo que hace a la estética de la aplicación.
- ContenedorLapiz extiende de Canvas y se inicializa con la imagen de un lápiz que se moverá a través de la clase VistaLapiz cada vez que se ejecute el algoritmo de los movimientos seleccionados por el usuario.
- BarraMenu: es una clase que extiende de MenuBar. Está barra se ubica en la parte superior de la ventana. A través de la barra el usuario tiene la posibilidad de salir del juego.
- Otra parte de la vista es el Titulo que extiende de Label y tiene el nombre de AlgoBlocks.
- Por último, el FondoVentana extiende de ImageView y tiene agregada una imagen la cual es el fondo de la interfaz.

En el paquete Controladores se encuentran todos los eventos de cada botón que existe en la vista.

Aplicación de patrón Observer

Este patrón está compuesto por un objeto que es observado (observable) por otros objetos observadores (observers). Ante un cambio en el objeto observado éste notifica a sus observadores y estos últimos realizan la actualización correspondiente. De manera tal que, la implementación de este patrón en una interfaz gráfica se ve reflejado en actualizaciones en la vista de los elementos en la interfaz en la medida que hayan cambios en el modelo. Estos cambios en el modelo parten de la interacción del usuario con la interfaz mediante un evento.

Hay tres actualizaciones de vista que se requieren en la interfaz de AlgoBlocks:

- El trazo en el tablero
- El lápiz del tablero
- El contenedor del algoritmo

Ante la necesidad de tener que hacer estas actualizaciones resulta útil la implementación del patrón observer.

Ventajas de la aplicación del patrón observer

La independencia que existe entre los objetos observados y los observadores:

- Permite respetar que el modelo no se vea afectado por cambios en la interfaz.
- El objeto observado es ajeno a las implementaciones o responsabilidades de los observadores.
- Respeta el patrón MVC. La implementación de los objetos observados no contiene ningún comportamiento de los observadores y viceversa.

Excepciones

NumeroDeRepeticionesInvalidoError

El BloqueRepetir recibe por parámetro en su constructor la cantidad de repeticiones que debe hacer. De esta manera, si en algún futuro se quieren agregar más repeticiones solamente debe pasarse ese número por parámetro y no requiere la implementación de otra clase.
Esta excepción se lanza si el bloque recibe un número de repeticiones menor a dos. Si el número ingresado fuese menor a dos no se estaría respetando la responsabilidad del BloqueRepetir.

CantidadInsuficienteDeBloquesError

Las posibilidades de guardar un algoritmo y/o ejecutarlo se ve limitada por el hecho de que exista al menos un bloque de movimiento para hacerlo.
Esta excepción se encuentra en la clase AlgoBlocks en el método guardarAlgoritmo(string) el cual guarda un algoritmo para crear un bloque personalizado. Y en la clase Algoritmo en el método ejecutar() que se encarga de recorre la lista de bloques y delegar su ejecución a cada bloque.

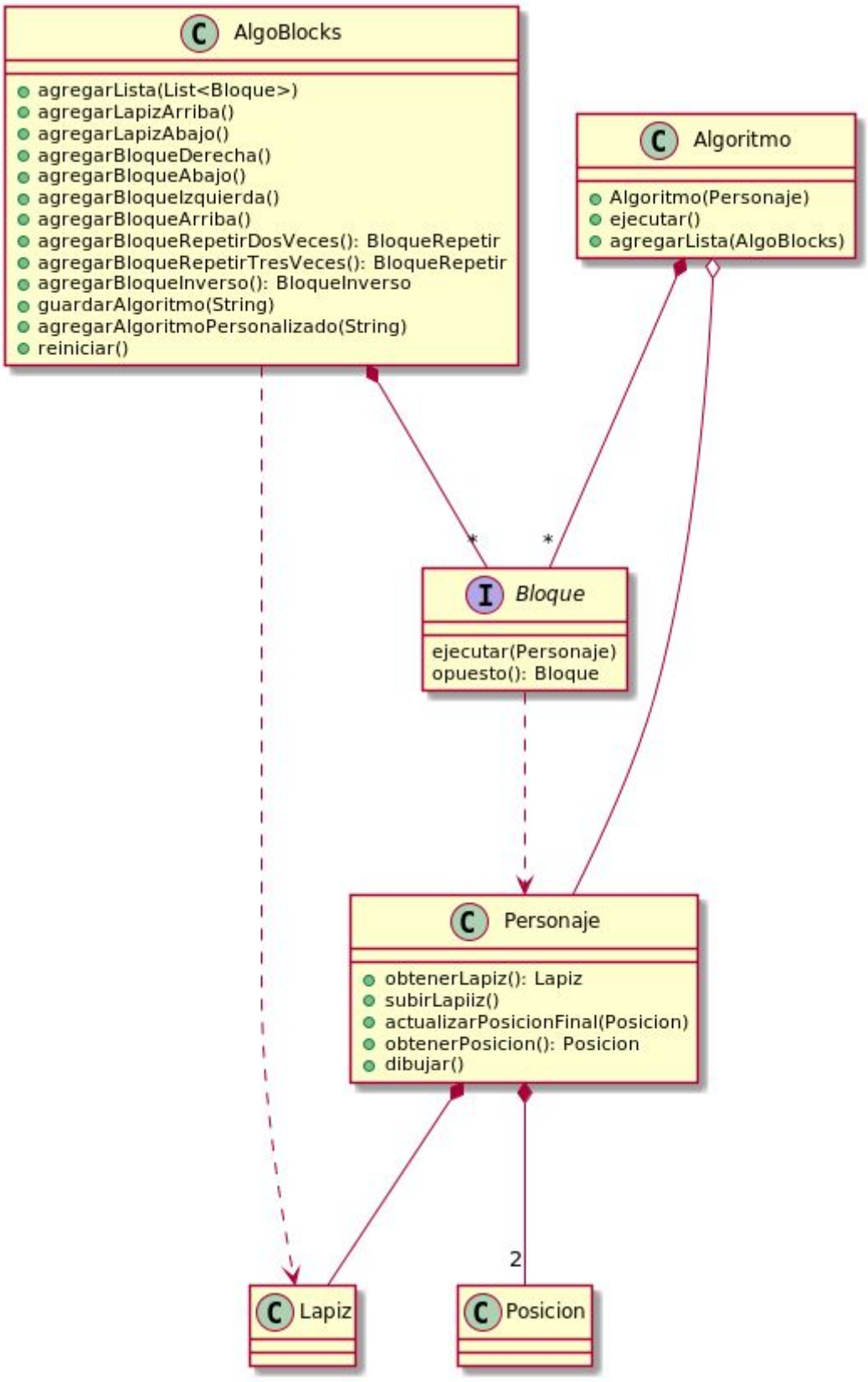
NombreInvalidoError

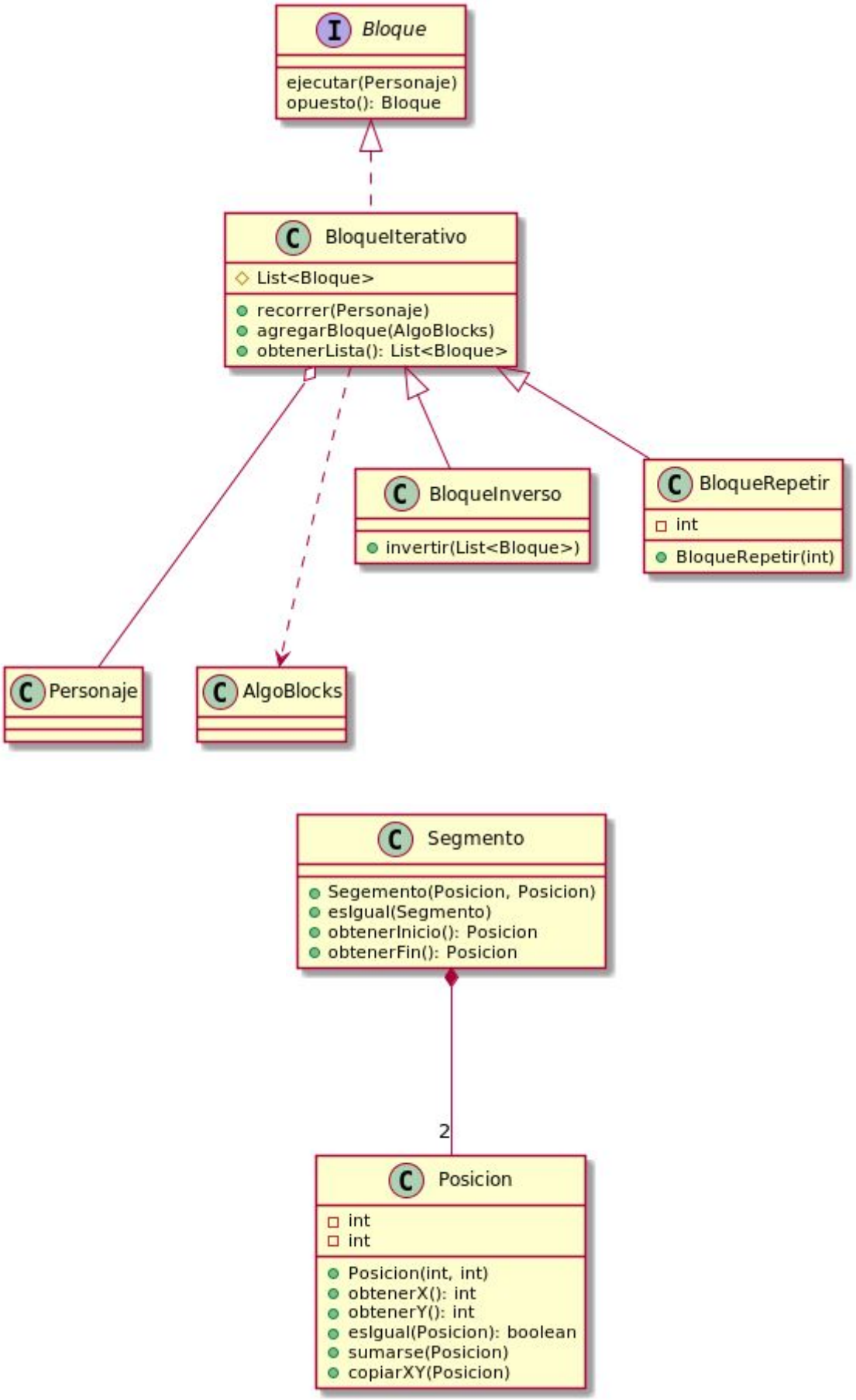
Es una excepción que se encuentra en el método guardarAlgoritmo(string) de la clase AlgoBlocks. Su motivo es no permitir que se cree un bloque personalizado (AlgoritmoPersonalizado) con un nombre ya existente.

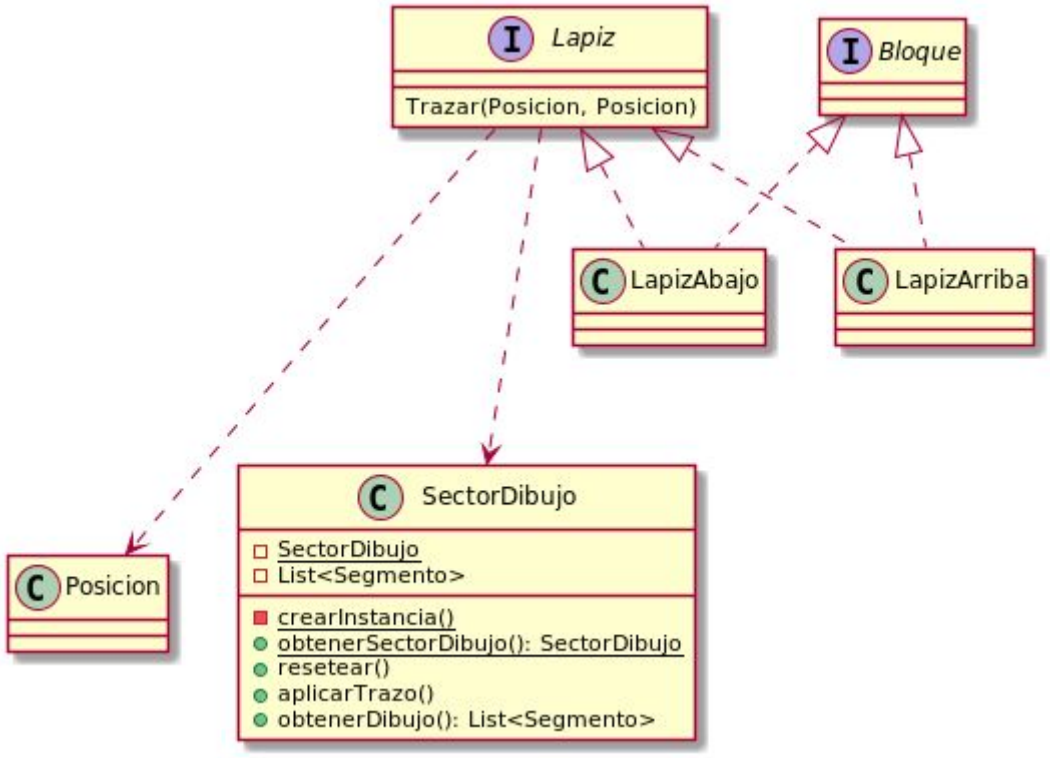
SegmentoInvalidoError

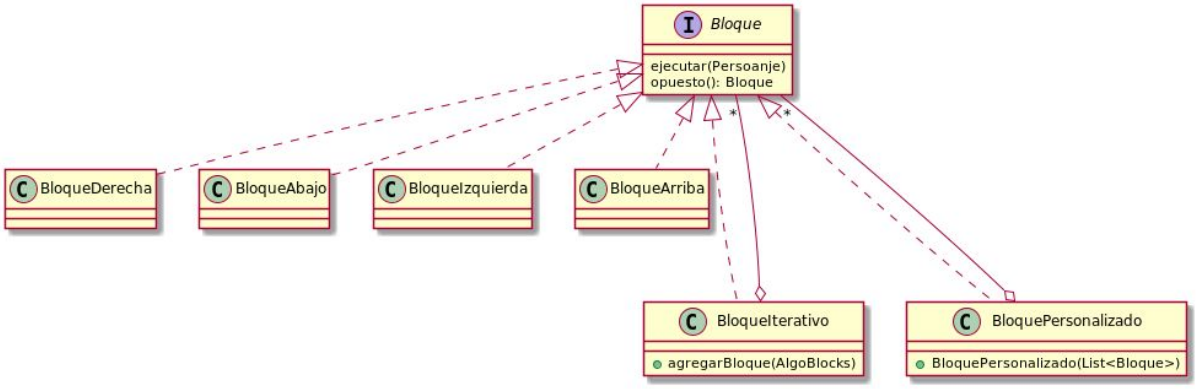
Excepción que se encuentra en el constructor de la clase Segmento(Posición inicial, Posición final) y controla que no se pueda crear dos segmentos iguales.

Diagrama de clases UML

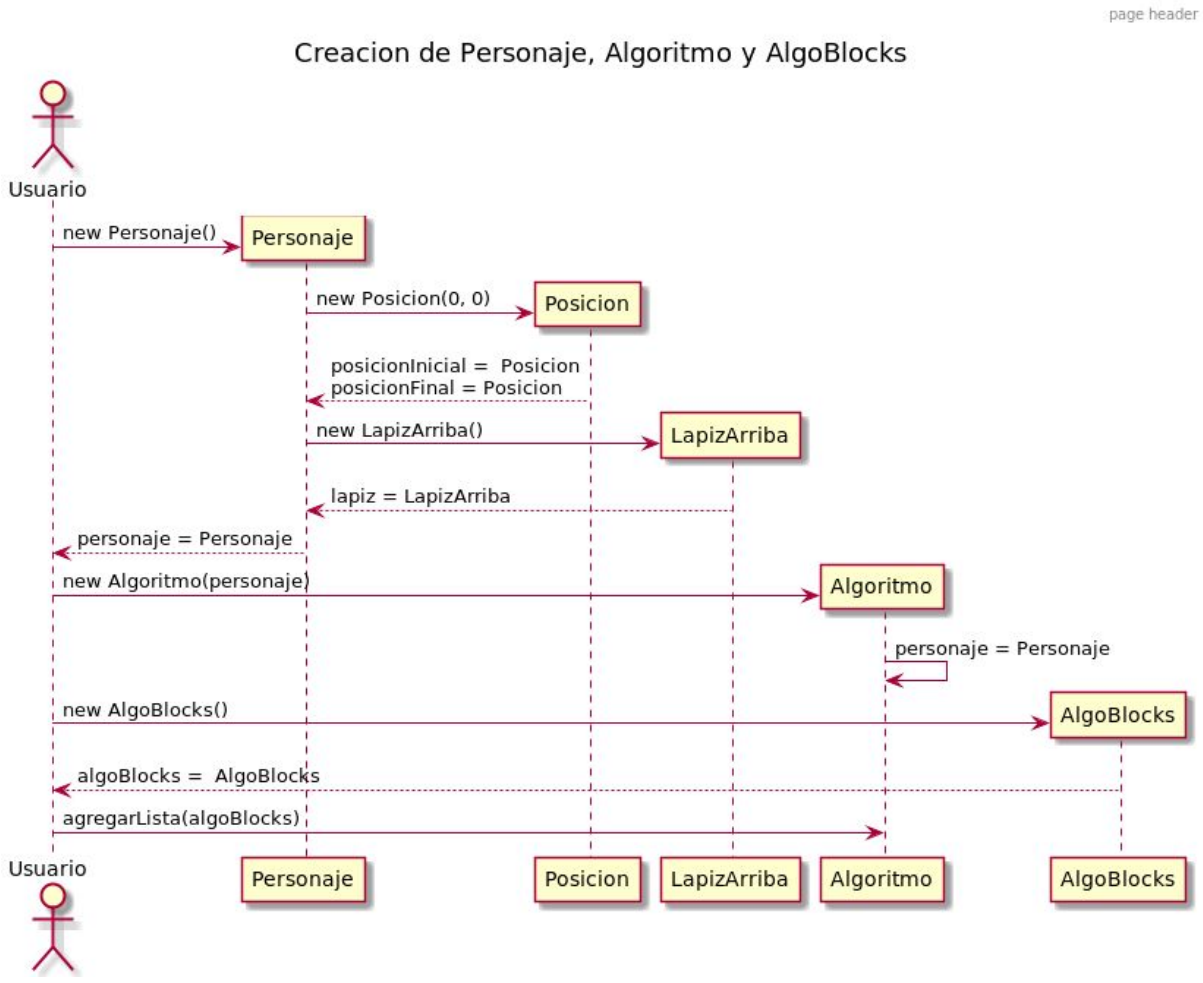


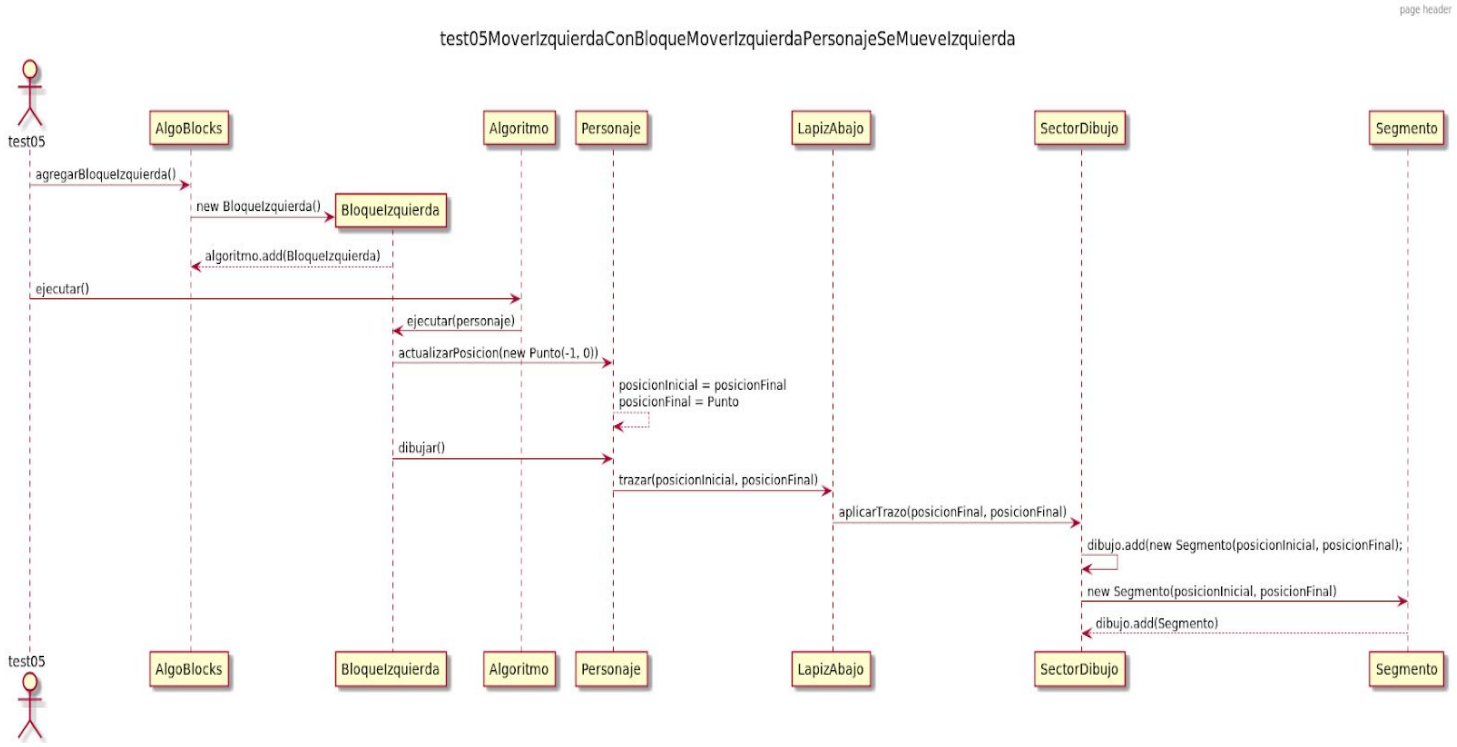

Bloque
ejecutar(Personaje)
opuesto(): Bloque
BloqueIterativo
List<Bloque>
recorrer(Personaje)
agregarBloque(AlgoBlocks)
obtenerLista(): List<Bloque>
BloqueInverso
invertir(List<Bloque>)
BloqueRepetir
int
BloqueRepetir(int)
Personaje
AlgoBlocks
Segmento
Segemento(Posicion, Posicion)
esIgual(Segmento)
obtenerInicio(): Posicion
obtenerFin(): Posicion
2
Posicion
int
int
Posicion(int, int)
obtenerX(): int
obtenerY(): int
esIgual(Posicion): boolean
sumarse(Posicion)
copiarXY(Posicion)


Lapiz
Trazar(Posicion, Posicion)
Bloque
LapizAbajo
LapizArriba
SectorDibujo
SectorDibujo
List<Segmento>
crearInstancia()
obtenerSectorDibujo(): SectorDibujo
resetear()
aplicarTrazo()
obtenerDibujo(): List<Segmento>
Posicion

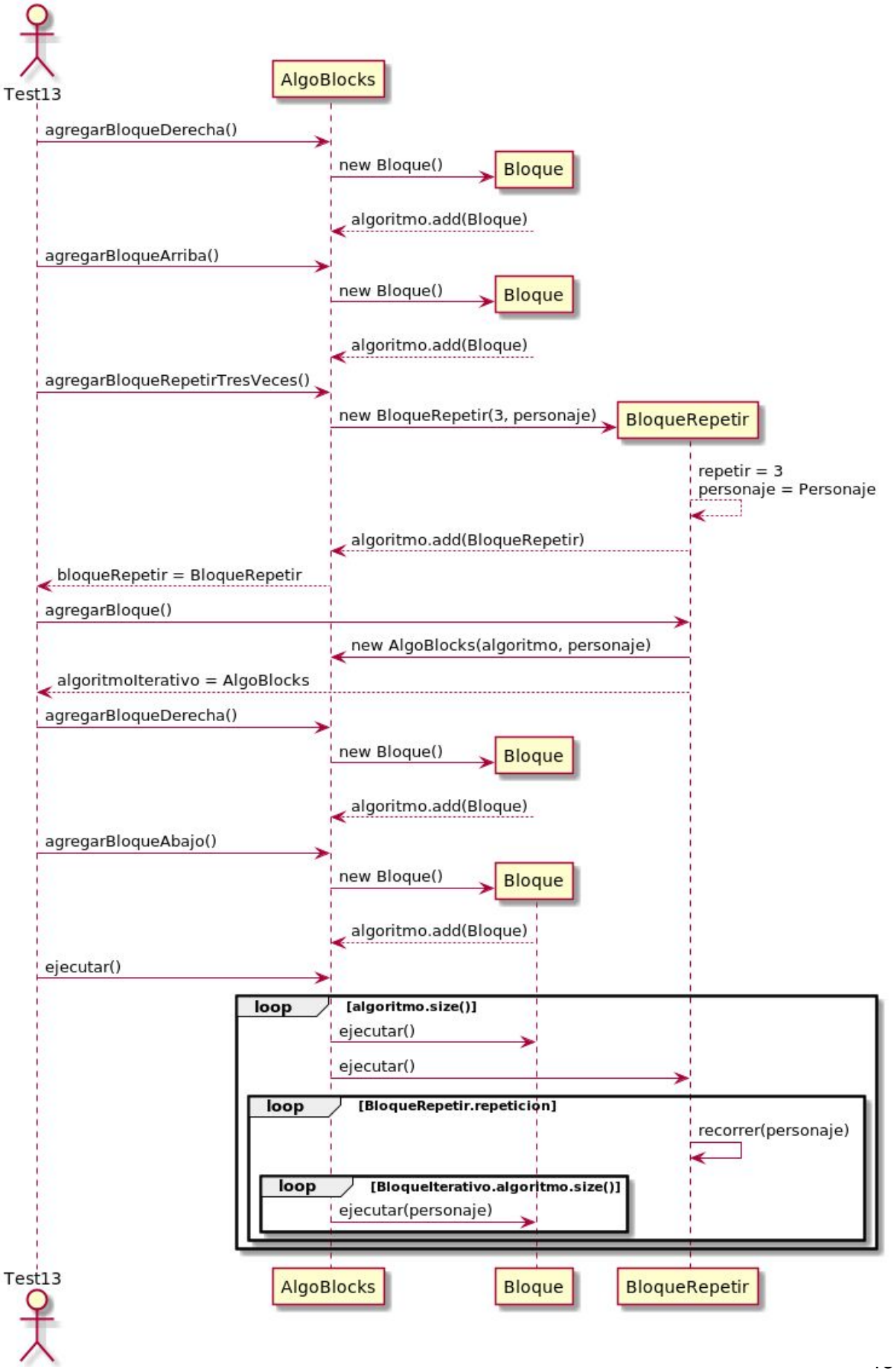

Bloque
ejecutar(Persoanje)
opuesto(): Bloque
BloqueDerecha
BloqueAbajo
BloqueIzquierda
BloqueArriba
BloqueIterativo
agregarBloque(AlgoBlocks)
BloquePersonalizado
BloquePersonalizado(List<Bloque>)

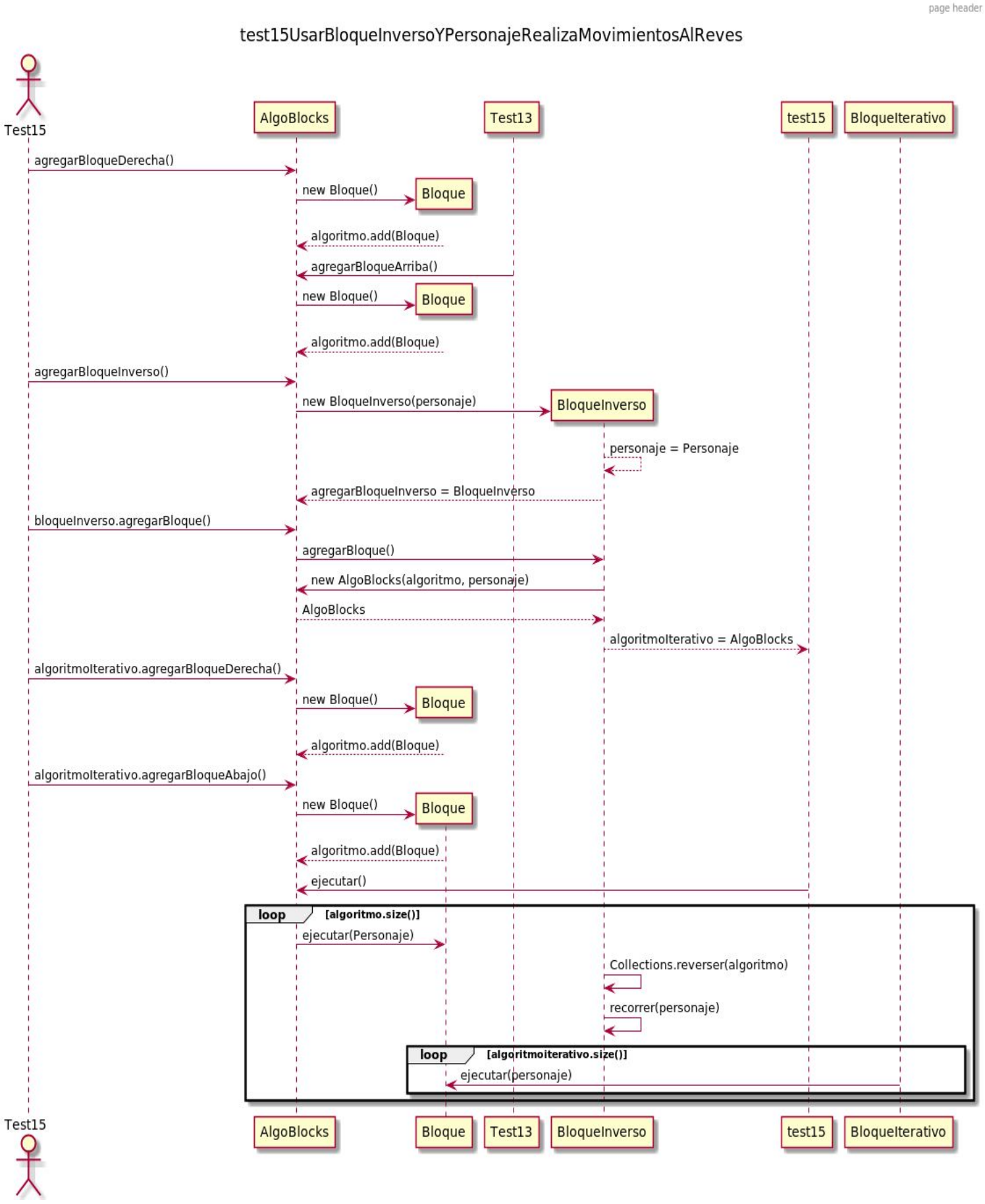
Diagramas de secuencia




test05MoverIzquierdaConBloqueMoverIzquierdaPersonajeSeMueveIzquierda
test05
AlgoBlocks
Algoritmo
Personaje
LapizAbajo
SectorDibujo
Segmento
agregarBloqueIzquierda()
new BloqueIzquierda()
BloqueIzquierda
algoritmo.add(BloqueIzquierda)
ejecutar()
ejecutar(personaje)
actualizarPosicion(new Punto(-1, 0))
posicionInicial = posicionFinal
posicionFinal = Punto
dibujar()
trazar(posicionInicial, posicionFinal)
aplicarTrazo(posicionFinal, posicionFinal)
dibujo.add(new Segmento(posicionInicial, posicionFinal);
new Segmento(posicionInicial, posicionFinal)
dibujo.add(Segmento)
test05
AlgoBlocks
BloqueIzquierda
Algoritmo
Personaje
LapizAbajo
SectorDibujo
Segmento

test13UsarBloqueRepetirTresVecesYPersonajeSeMueveCorrectamente




test15UsarBloqueInversoYPersonajeRealizaMovimientosAlReves
Test15
AlgoBlocks
Test13
test15
BloqueIterativo
agregarBloqueDerecha()
new Bloque()
Bloque
algoritmo.add(Bloque)
agregarBloqueArriba()
new Bloque()
Bloque
algoritmo.add(Bloque)
agregarBloqueInverso()
new BloqueInverso(personaje)
BloqueInverso
personaje = Personaje
agregarBloqueInverso = BloqueInverso
bloqueInverso.agregarBloque()
agregarBloque()
new AlgoBlocks(algoritmo, personaje)
AlgoBlocks
algoritmoIterativo = AlgoBlocks
algoritmoIterativo.agregarBloqueDerecha()
new Bloque()
Bloque
algoritmo.add(Bloque)
algoritmoIterativo.agregarBloqueAbajo()
new Bloque()
Bloque
algoritmo.add(Bloque)
ejecutar()
loop
[algoritmo.size()]
ejecutar(Personaje)
Collections.reverser(algoritmo)
recorrer(personaje)
loop
[algoritmoiterativo.size()]
ejecutar(personaje)
Test15
AlgoBlocks
Bloque
Test13
BloqueInverso
test15
BloqueIterativo

Diagramas de paquetes

Diagramas de estado

Diagrama de estado de AlgoBlocks:

Diagrama de estado de un Bloque:

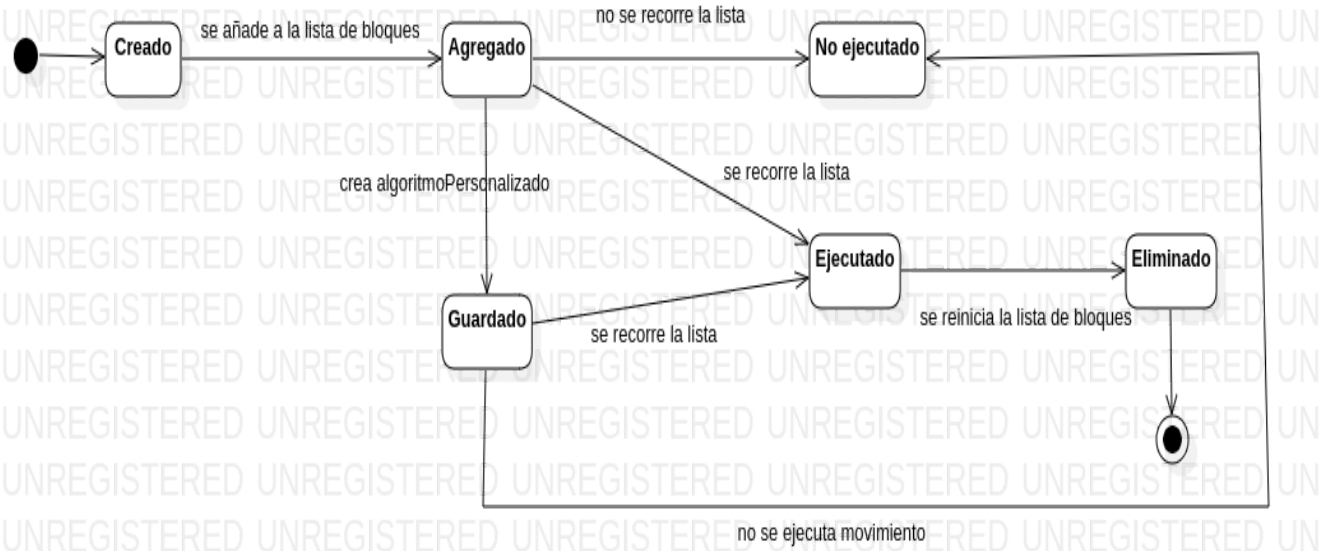


Diagrama de estado de la lista de recorrido que almacena los bloques:

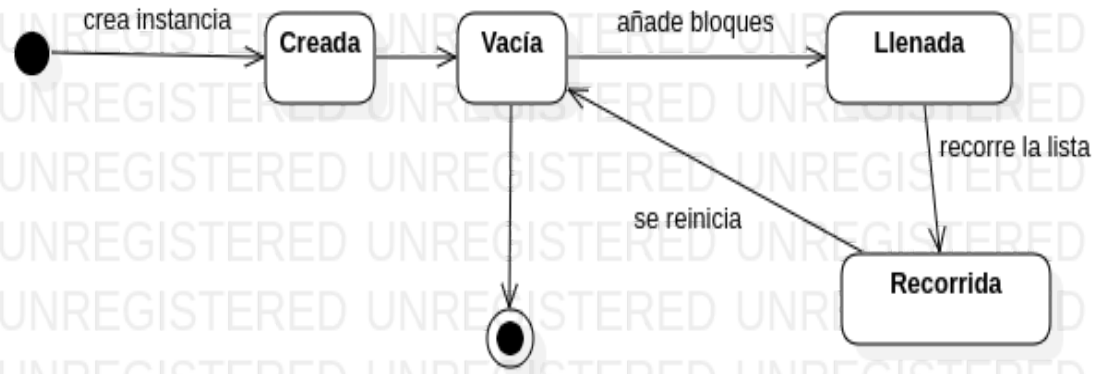


Diagrama de estado del objeto Posición:

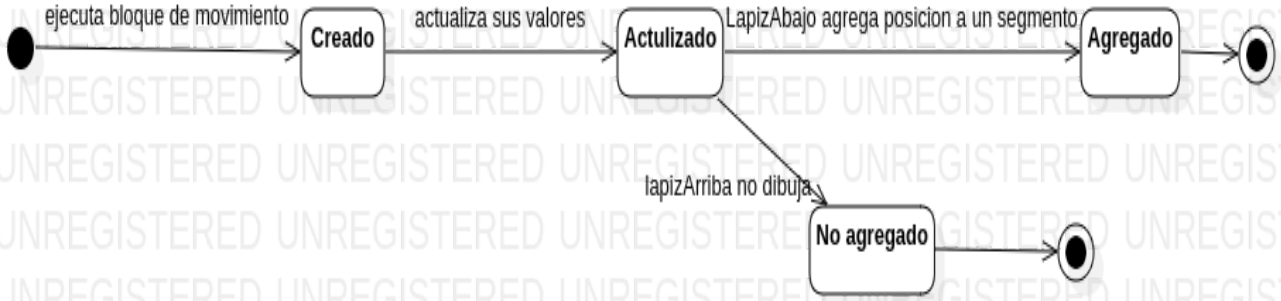


Diagrama de estado de una lista de un BloqueInverso:

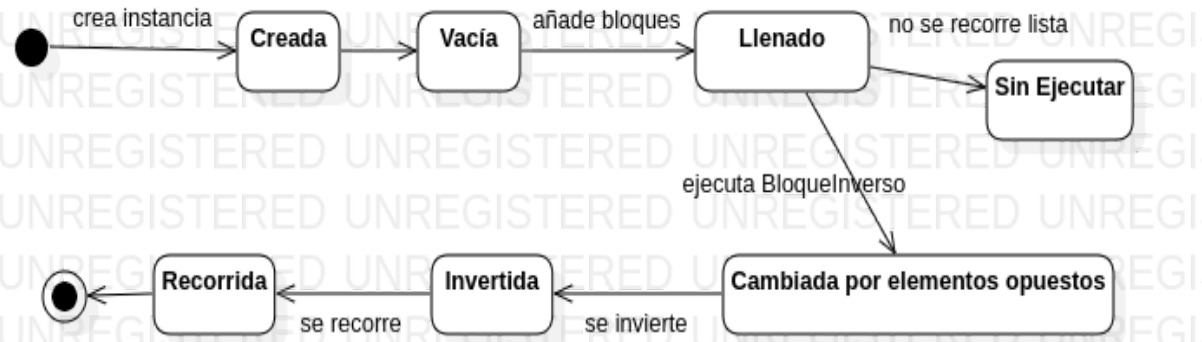


Diagrama de estado una lista de BloqueRepetirDosVeces:

Diagrama de estado de una lista de BloqueRepetirTresVeces: