FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Curso 2 – Grupo 10
Segundo Cuatrimestre 2020

Alumno 1:	Joaquín Lerer
Número de padrón:	105493
Email:	joacolerer@gmail.com

Alumno 2:	Nicolás Podesta
Número de padrón:	104077
Email:	npodesta@fi.uba.ar

Alumno 3:	Dante Reinaudo
Número de padrón:	102848
Email:	dreinaudo@fi.uba.ar

Alumno 4:	Sebastián Vera Benitez
Número de padrón:	104115
Email:	svera@fi.uba.ar

Índice

1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III. Este consiste en desarrollar una aplicación de un juego que permite aprender los conceptos básicos de programación, mediante el armado de algoritmos y la utilización de bloques visuales. Para lograr el objetivo se aplicaran de manera grupal 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.

2. Supuestos

La posición del personaje está restringida, por lo tanto, existen límites para el movimiento del personaje. En caso de superarse estos límites, el personaje cruzara las fronteras del sector dibujo y continuara realizando las acciones por el lado opuesto al límite que acaba de traspasar, tal como sucede en juegos como *Snake* o *Pac-Man*.

No existe una cantidad máxima de bloques que pueda almacenar un conjunto de bloques. Dado que a su vez los conjuntos de bloques, son al mismo tiempo bloques, es posible concatenar y anidar conjuntos dentro de conjuntos, sin ninguna limitación.

Solo es posible remover el último bloque dentro de un conjunto de bloques. No es posible remover el último bloque de un conjunto vacío, en caso de hacerlo se lanzara la excepción correspondiente.

Es posible ejecutar los conjuntos de bloques vacíos, pero estos no realizaran ninguna acción. Del mismo modo, también es posible ejecutarlos invertidos.

Una vez pintado un trazo en la pizarra, esta no puede borrarse, no existe una entidad que permita llevar a cabo este proceso.

Al invertir un bloque de repetición, se invertirán todos los bloques que este contenga y luego se repetirán la cantidad de veces establecidas. Por ejemplo: si se crea un bloque de repetición de tres iteraciones, y se le agregase un bloque de mover a la izquierda, al negar este bloque, el personaje se movería tres veces hacia la derecha.

Para almacenar un bloque personalizado, es necesario que este contenga por lo menos un bloque o una instrucción. Por lo tanto, no será posible almacenar bloques personalizados vacíos.

Una vez almacenado un algoritmo personalizado, este no podrá modificarse de ninguna manera. Por esta razón a un algoritmo personalizado, no se le es posible agregar o remover un bloque. Tampoco es posible cambiarle su nombre.

3. Diagrama de clase

A continuación, se muestran los diagramas de clases de nuestra solución propuesta para el presente trabajo práctico. Dado que nuestro modelo consta de varias clases, optamos en separarlo en tres diagramas para mejorar la visualización de nuestro esquema planteado. En ellos se muestran todos las clases del modelo, sus atributos, sus métodos y la relación que mantienen entre si dichas clases. Cabe aclarar que en los diagramas no aparecen absolutamente todos los métodos y atributos, sino los más representativos para esclarecer la solución planteada.

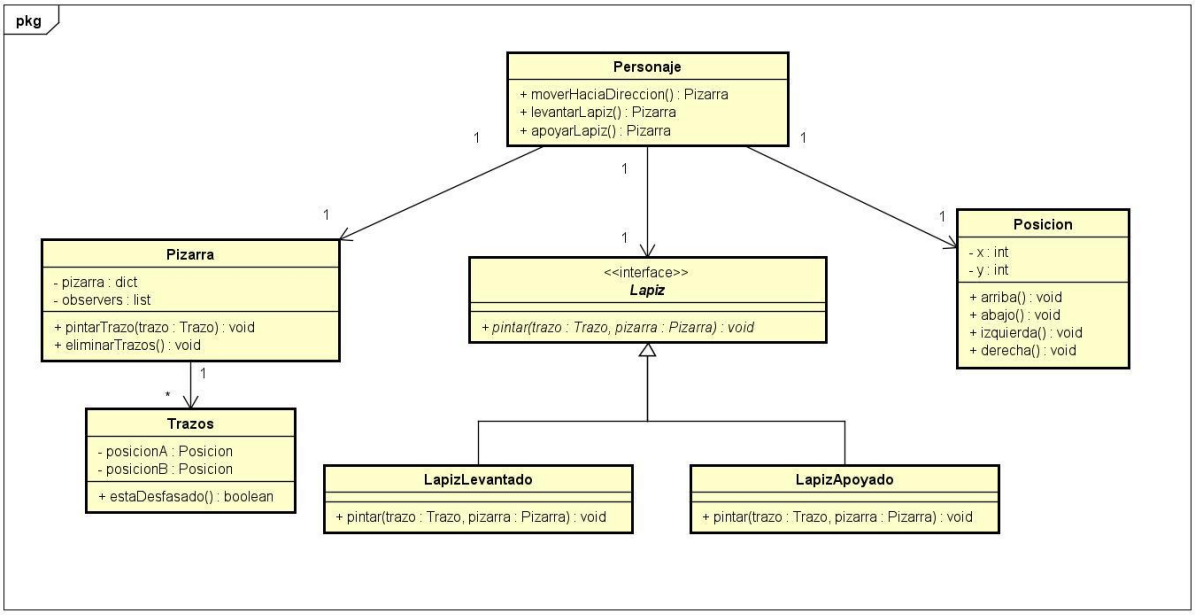

Figura 1: Diagrama de clases, estructura de la clase Personaje y sus asociaciones.

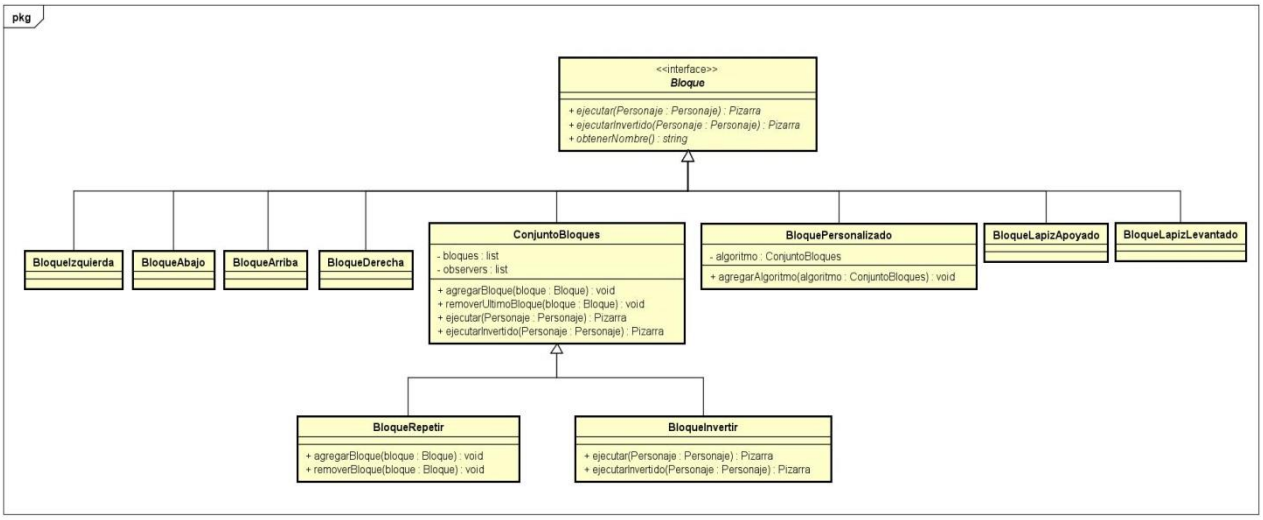

Figura 2: Diagrama de clases, estructura y organización de la interfaz *Bloque* y sus clases hijas.

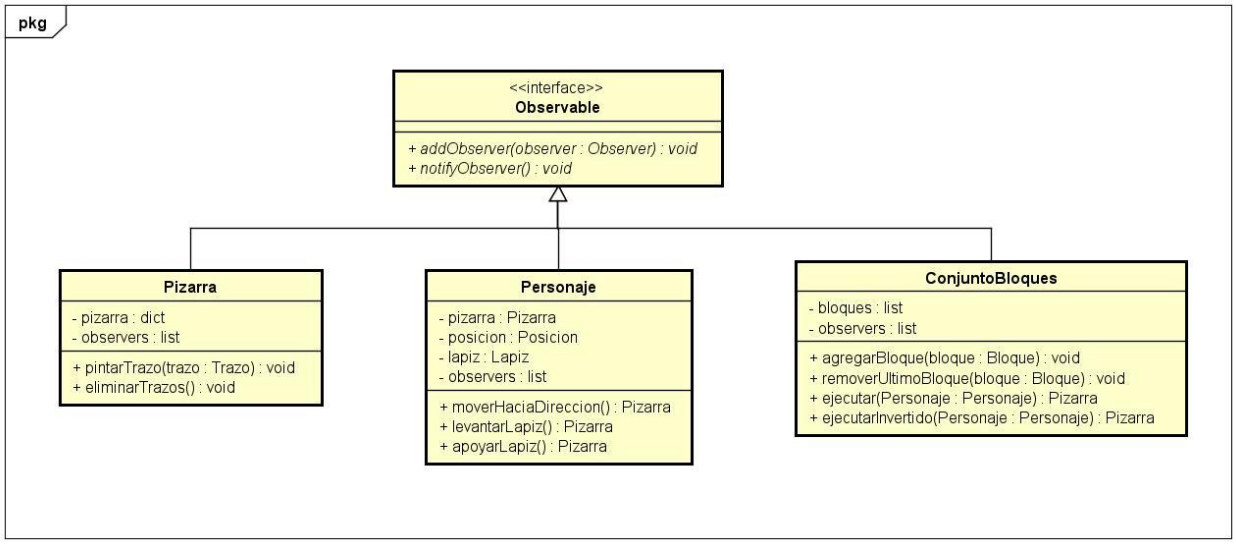

Figura 3: Diagrama de clases, implementaciones de la interfaz *Observable* utilizada para poder llevar a cabo el patrón Observer.

4. Diagramas de secuencia

Con el fin de comunicar y esclarecer un poco más la solución planteada para el sistema, en esta sección del informe, se presentaran algunos diagramas de secuencia donde se muestran distintas situaciones posibles que comprende el modelo. En ellos se podrá observar los mensajes enviados entre los distintos objetos del sistema a lo largo del tiempo. Es importante resaltar que en dichos diagramas no aparecen absolutamente todos los mensajes que llevan a cabo los objetos si no los más representativos, facilitando su lectura y comprensión dado que una de las funciones principales de estos diagramas es comunicar. Al mismo tiempo, serán de gran ayuda para complementar la información que aparece en el diagrama de clases.

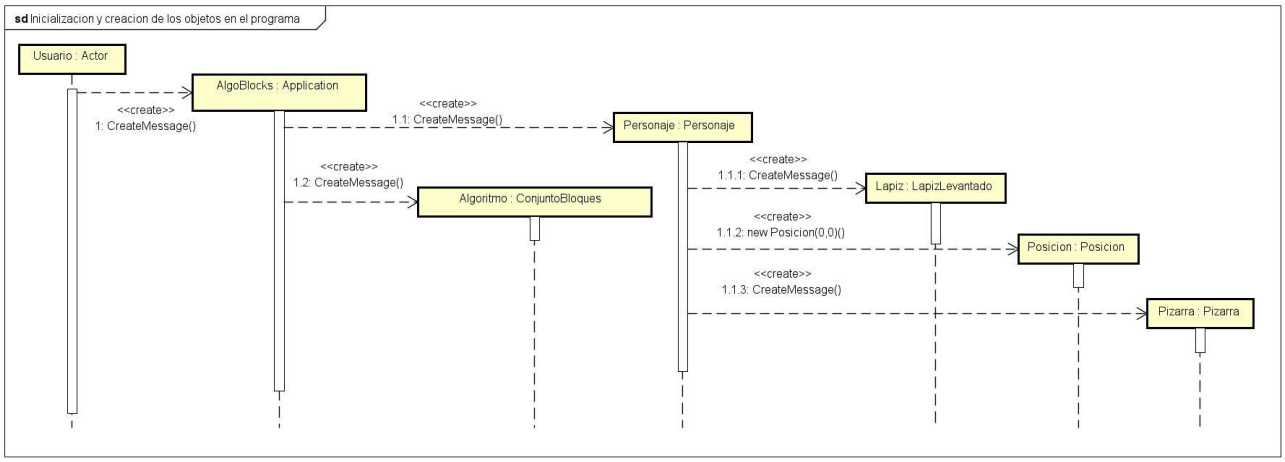

Figura 4: Diagrama de secuencia: Inicialización y creación de los objetos del programa

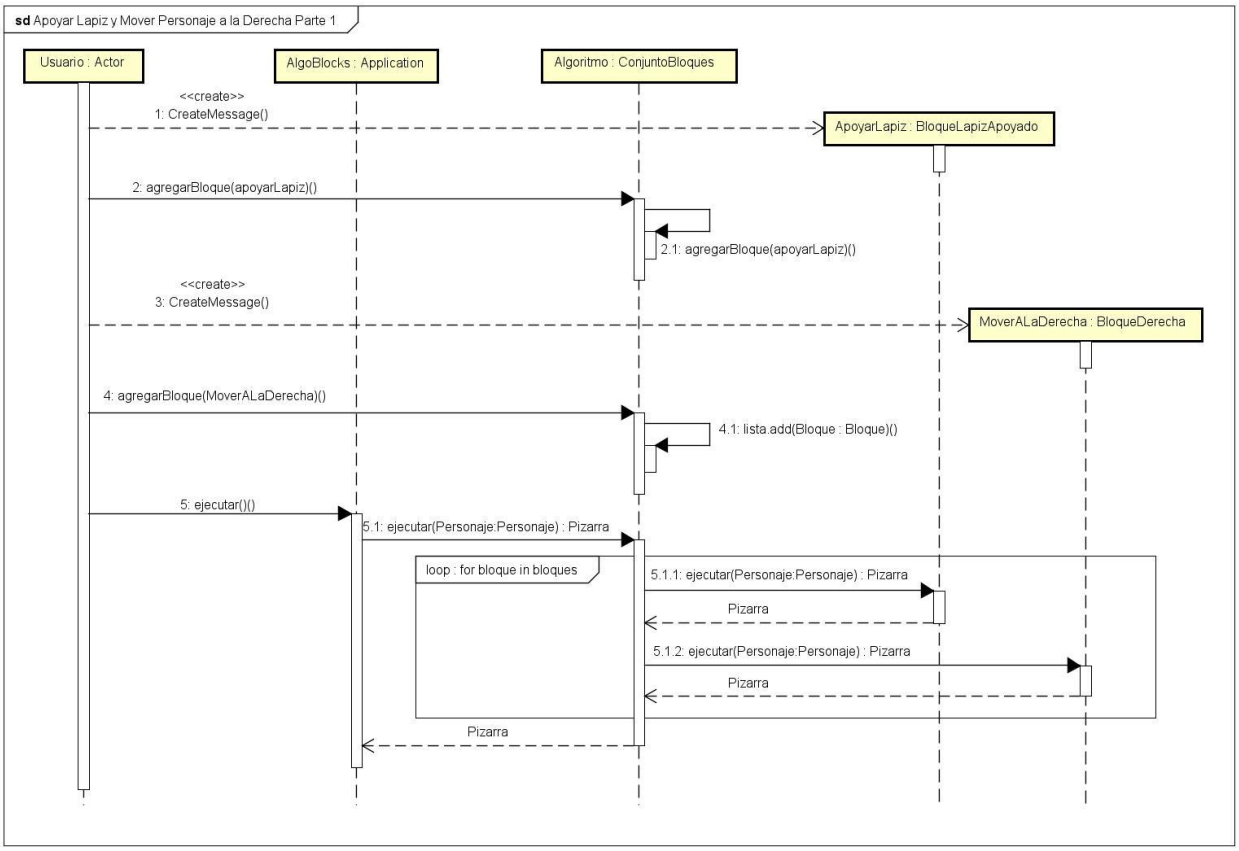

Figura 5: Diagrama de secuencia Apoyar Lapiz y mover Personaje a la derecha (Primera Parte).

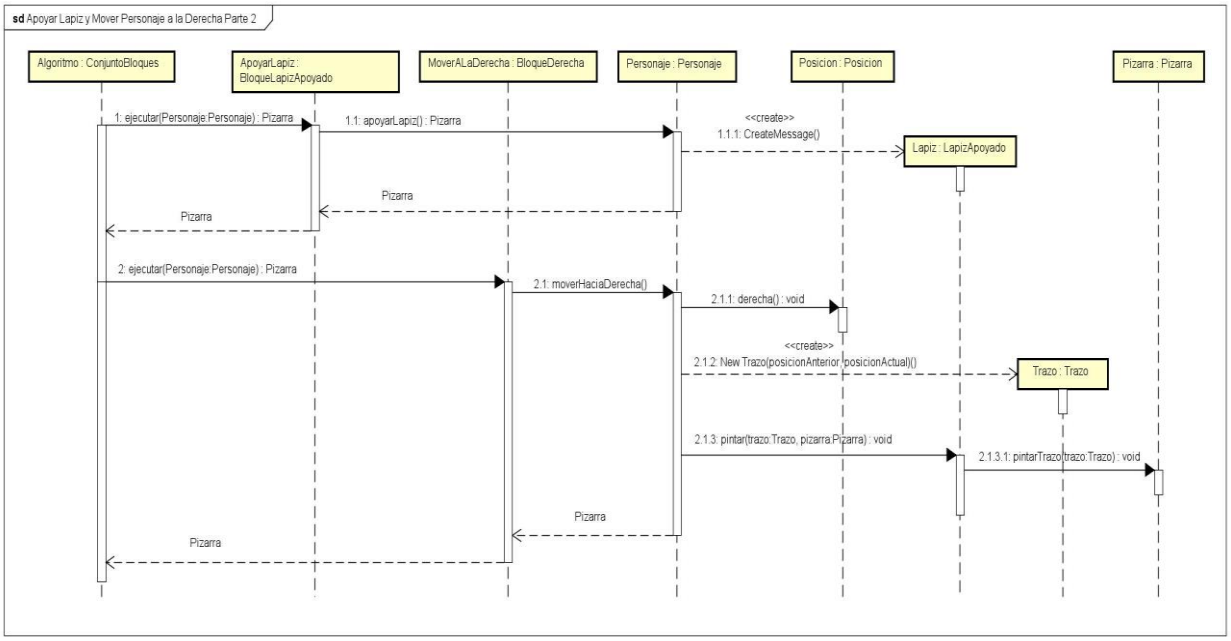

Figura 6: Diagrama de secuencia Apoyar Lapiz y mover Personaje a la derecha (Segunda Parte).

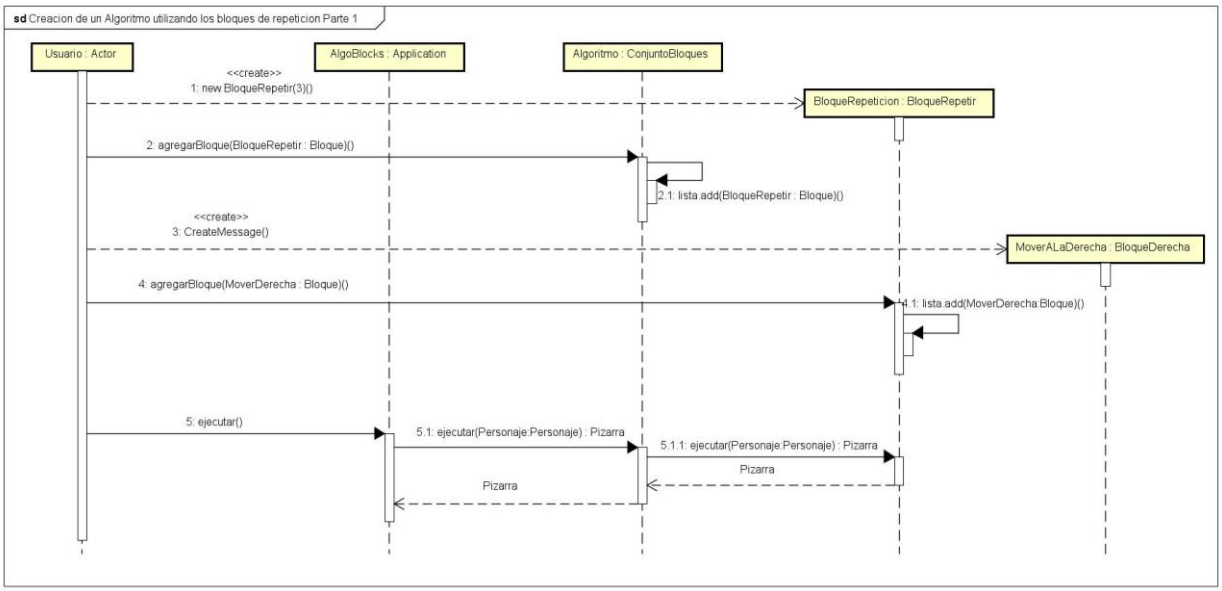

Figura 7: Diagrama de secuencia creación de un Algoritmo utilizando bloques de repetición (Primera Parte).

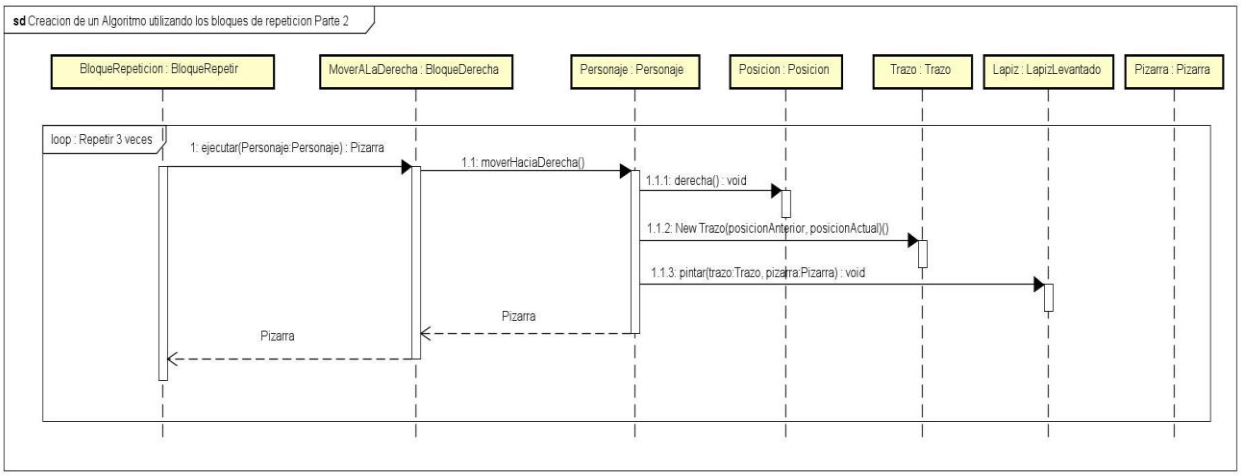

Figura 8: Diagrama de secuencia creación de un Algoritmo utilizando bloques de repetición (Segunda Parte).

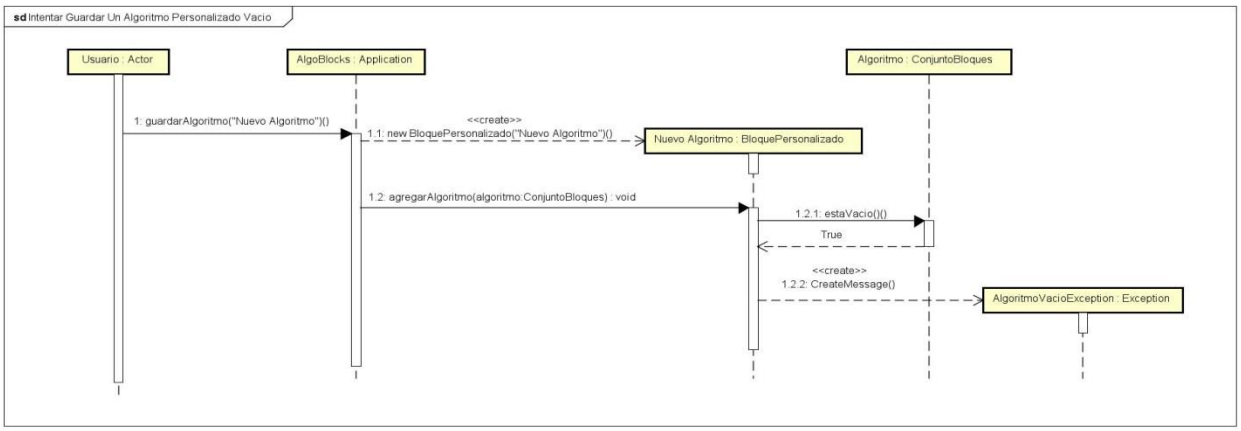

Figura 9: Diagrama de secuencia intentar almacenar un algoritmo sin bloques.

5. Diagramas de paquete

En esta sección del informe, se presentaran algunos diagramas de paquetes que muestran el acoplamiento de nuestro trabajo. En ellos se podrán ver las dependencias entre los paquetes que componen nuestro modelo.

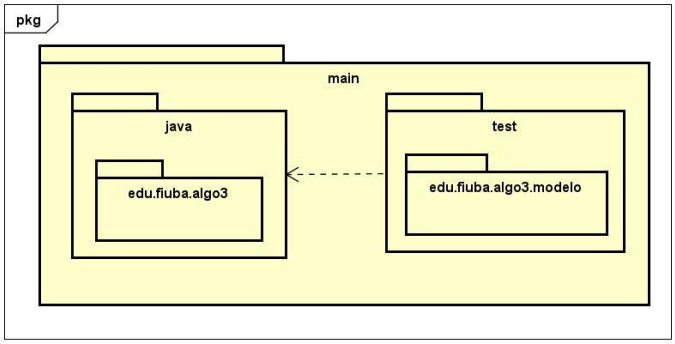

Figura 10: Diagrama de paquetes del fichero main.

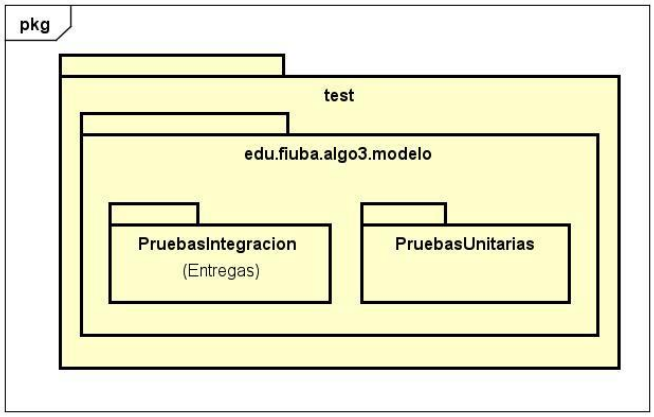

Figura 11: Diagrama del paquetes del fichero test.

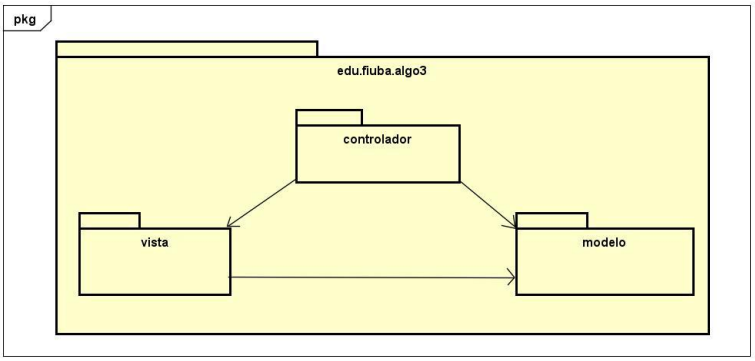

Figura 12: Diagrama del paquete edu.fiuba.algo3, se observa claramente el patrón modelo-vista-controlador.

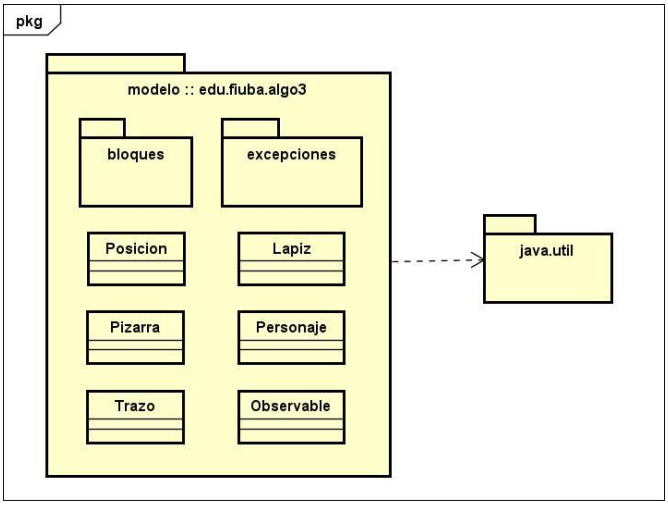

Figura 13: Diagrama de paquete del modelo.

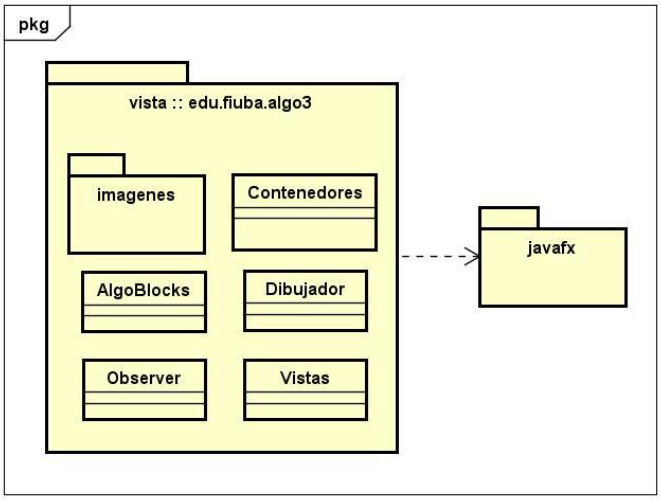

Figura 14: Diagrama de paquete de la vista.

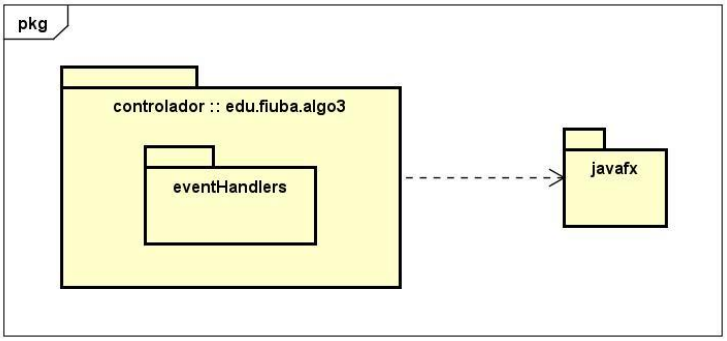

Figura 15: Diagrama de paquete del controlador.

6. Diagramas de estado

En este apartado del informe se mostraran diversos diagramas de estados, o también conocidos como diagramas de máquinas de estado. Estos diagramas se utilizan para modelar la vida y el comportamiento de un solo objeto, especificando la secuencia de eventos que un objeto atraviesa durante su tiempo de vida. Son muy útiles para mostrar las transiciones entre los diferentes estados posibles de un mismo objeto.

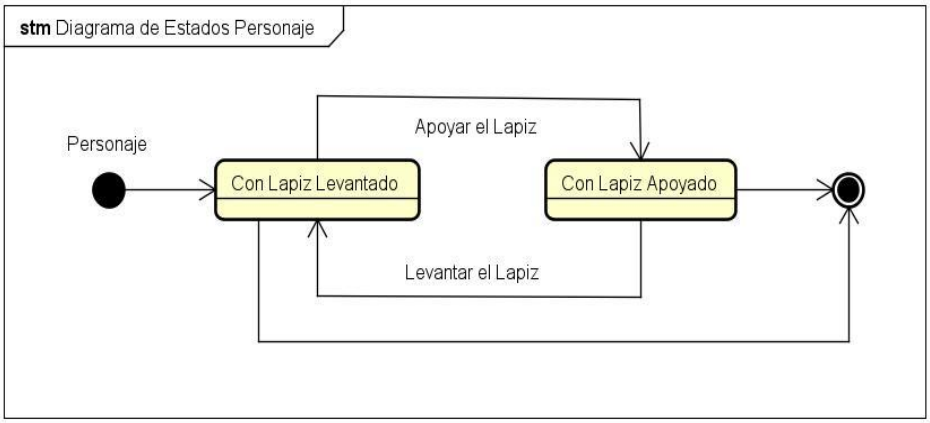

Figura 16: Diagrama de estado, modela la vida del objeto Personaje y sus estados posibles.

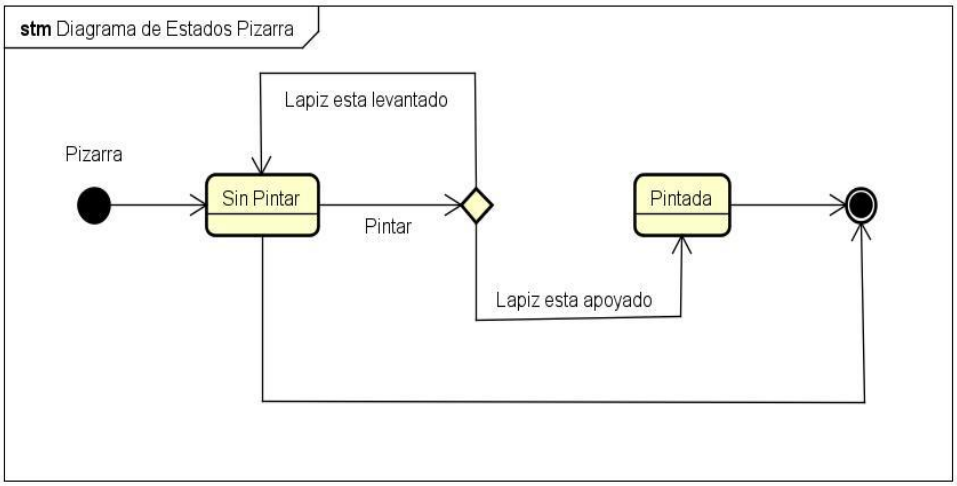

Figura 17: Diagrama de estado, modela la vida del objeto Pizarra y sus posibles estados.

7. Detalles de implementación

7.1. Técnicas de diseño y planteo de la solución

Para resolver este trabajo utilizamos la metodología TDD (Test-Driven Depelopment) vista en clase, esta técnica de diseño consiste en ir realizando pruebas unitarias primero, haciendo lo mínimo y suficiente para poder cumplir con dichas pruebas. Una vez logrado esto se refactoriza el código escrito para obtener una mayor prolijidad, y así, cumplir con las llamadas buenas prácticas de programación. Este proceso se repite una y otra vez hasta poder abarcar todos los campos y llegar a una solución muy optimizada. Esta metodología presenta muchas ventajas, entre ellas: se garantiza el correcto funcionamiento del código, gracias a que se cuenta con diversas pruebas del mismo; se minimiza la cantidad de código a escribir ya que al realizar pruebas, se evita la posibilidad de escribir código que no se va a utilizar; se mejora el diseño y la calidad de código.

Por otro lado, también hicimos uso de la práctica de Integración Continua, para así poder optimizar la detección de fallos y errores durante la compilación o ejecución de nuestro programa. Para lograr esto, utilizamos la herramienta de Integración Continua Travis-CI.

A su vez, para abordar diversas cuestiones de este trabajo práctico, aplicamos la metodología de Pair Programming, o también conocido como programación en pareja. Haciendo uso de la plataforma Disccord, pudimos debatir los puntos más conflictivos del trabajo, de esta manera, mientras que un compañero hacia *streaming* del código desde su computador y se encargaba de escribirlo, el resto de los integrantes del grupo supervisaba en tiempo real, debatiendo los problemas con los que pudiéramos toparnos y encontrando las soluciones.

Además, para realizar nuestro modelo, tomamos como guía los principios SOLID, estos son los cinco principios básicos de la programación orientada a objetos. Siguiendo estos principios, logramos producir un código de mayor calidad, que a su vez, es más reutilizable y mantenible a lo largo del tiempo. No obstante, existen diversos casos en los que se dificulta mucho poder cumplir con absolutamente todos ellos. Por lo tanto, es importante aclarar que estos principios se utilizan como guía para ayudar al programador, pero no son leyes que deben cumplirse obligatoriamente.

7.2. Explicación del Modelo: Clases utilizadas y sus métodos

Como puede verse claramente en el diagrama de clases mostrado en la tercera sección de este informe, el modelo propuesto consta de dieciséis clases concretas más tres interfaces, las cuales se detallaran a continuación:

- **Personaje**: es la entidad encargada de modelar al personaje que luego se moverá por la pizarra, obedeciendo las indicaciones que le den los bloques. Cuenta con tres atributos: un lápiz, el cual se encontrara en un estado Apoyado o Levantado; una pizarra, en donde se guardaran los movimientos del personaje y una posición. Al momento de su creación, sus atributos se inicializaran de la siguiente manera: el lápiz se crea en un estado Levantado; la pizarra se crea vacía, es decir, sin pintar; y la posición se inicializa en la ubicación relativa (0,0), elegida arbitrariamente. Por otro lado, para poder llevar a cabo las instrucciones recibidas por los

bloques, el personaje le delega sus responsabilidades a sus atributos. En cuanto a su movimiento, el personaje le delega esta responsabilidad a la posición, la cual se va actualizando por cada paso que dé el personaje. Luego de esto, el personaje le delega la responsabilidad al lápiz de comunicarle a la pizarra de sus acciones y de pintarla en caso de ser necesario. Por último, en caso de recibir la instrucción del bloque para subir o apoyar el lápiz, este le delega dicha responsabilidad al mismo lápiz, el cual cambia su estado aplicando un patrón de diseño que pasaremos a detallar más adelante.

- **Posicion**: esta entidad modela una coordenada cartesiana con un eje x y otro eje y, en el modelo se utiliza para tener noción de la ubicación relativa del personaje, dado que este puede moverse. Cuenta con dos atributos enteros, x e y, los cuales son declarados al inicializar la posición. A su vez, estos atributos pueden ser modificados por los métodos: abajo, arriba, izquierda o derecha. Los primeros dos métodos, afectan a la coordenada y, mientras que los últimos dos, afectan a la coordenada x.

- **Trazo**: modela las líneas que deja marcadas en la pizarra la trayectoria del personaje, cuando este realiza un movimiento. Cuenta con dos atributos de la clase Posicion, denominados posicionA y posicionB, donde ambos son declarados al inicializar este objeto.

- **Pizarra**: esta entidad se encarga de almacenar los trazos pintados por el personaje, para conseguir esto, cuenta con un atributo llamado trazosPintadas, el cual es un HashSet de trazos. Se optó por utilizar un hash, dado que esto evita las colisiones y nos permite que no pueda pintarse dos veces el mismo trazo. Al recibir el mensaje pintar, el cual obtiene por parámetro un trazo, simplemente se agrega este trazo al hash de posiciones.

- **Lapiz (Interfaz):** utilizamos esta interfaz como mecanismo para poder desarrollar polimorfismo sin herencia, esta clase abstracta define la firma del método pintar, el cual recibe por parámetro una pizarra y un trazo. Cualquier clase que implemente esta interfaz entenderá el mensaje pintar, y lo implementara de una forma específica. Gracias a esto, luego que el personaje realice un movimiento, simplemente, le envía el mensaje pintar a su lápiz, sin necesidad de saber si este esta levantado o apoyado. Con esto se evita el uso de variables booleanas del estilo estaApoyado o estaLevantado, las cuales rompen el encapsulamiento y dificultan la extensibilidad del código. De esta manera, se obtiene una solución más elegante, que no viola con el principio abierto/cerrado y que se adecua mejor al paradigma de la programación orientada a objetos.

- **LapizApoyado:** esta clase implementa la interfaz Lapiz, y su método pintar. En dicho método se le delega la responsabilidad a la pizarra de pintar el trazo que recibió por parámetro. De esta manera, si el personaje realiza un movimiento, y su lápiz está apoyado, su recorrido se verá reflejado en la pizarra. Cabe destacar, que al crear las clases LapizApoyado y LapizLevantado, estamos aplicando el patrón de diseño, denominado patrón *State*, este patrón es utilizado cuando el comportamiento de un objeto cambia dependiendo de su estado. Para solucionar esto, el patrón plantea implementar una clase por cada estado diferente del objeto y el desarrollo de un método especifico según cada estado determinado. En nuestro caso, cuando un personaje recibe la instrucción de un BloqueLapizApoyado o un BloqueLapizLevantado, se espera que su comportamiento cambie, dependiendo del estado en el que se encuentre el lápiz se modificara lo que el usuario vea por el sector dibujo. Aplicando este patrón, el personaje cuenta con un atributo lápiz, el cual cambia de estado según se lo especifique, intercambiando de la clase LapizApoyado a la clase LapizLevantado, o viceversa, dependiendo el contexto. Esta

técnica de diseño hace que el código sea mucho más abierto y extensible, por ejemplo, si el día de mañana quisiera agregar un estado del lápiz que fuera LapizPuntaRota, y que pintase un trazo sí y otro no, me bastaría simplemente con crear esta clase y definirle el método pintar de la manera especificada.

- **LapizLevantado:** esta clase implementa la interfaz Lapiz, y su método pintar. En esta ocasión, dicho método no realiza ninguna acción, esto es lógico ya que un lápiz levantado o alejado de una pizarra no dibuja sobre ella. De esta manera, si el personaje realiza un movimiento, y su lápiz está levantado, su recorrido no se verá reflejado en la pizarra.

- **Bloque (Interfaz):** al igual que Lapiz, esta interfaz también es utilizada como mecanismo para aplicar polimorfismo sin herencia. Esta clase abstracta define la firma de los métodos ejecutar y ejecutarInvertido, donde ambos reciben por parámetro un personaje y retornan una pizarra. Cualquier tipo de bloque que implemente esta clase, tiene la responsabilidad de comprender estos dos mensajes. Por lo tanto, en nuestro modelo, todo lo que sea un bloque debe saber ejecutarse y ejecutar su inverso, esto es así, dado que se solicita en el enunciado un bloque cuya función es negar los bloques que contenga. Implementando esta interfaz, se logra que el algoritmo al ejecutarse, simplemente le envíe el mismo mensaje ejecutar a cada bloque que contenga, sin la necesidad de saber qué tipo de bloque se está ejecutando.

- **BloqueAbajo:** este clase implementa la interfaz Bloque, al ejecutarse se encarga de decirle al personaje (el cual recibió por parámetro) que se dirija hacia abajo. Análogamente, al ejecutarse invertido, le envía un mensaje al personaje diciéndole que se mueva hacia arriba. Es importante destacar que los bloques de movimiento no se encargan de desplazar al personaje, estos le delegan la responsabilidad de moverse al mismo personaje. Este, a su vez, como vimos anteriormente, le delega esta misma responsabilidad a su posición.

- **BloqueArriba:** este clase implementa la interfaz Bloque, al ejecutarse se encarga de decirle al personaje recibido por parámetro que se dirija hacia arriba. Análogamente, al ejecutarse invertido, le envía un mensaje al mismo personaje diciéndole que se mueva hacia abajo.

- **BloqueDerecha:** este clase implementa la interfaz Bloque, al ejecutarse se encarga de decirle al personaje recibido por parámetro que se dirija hacia la derecha. Análogamente, al ejecutarse invertido, le envía un mensaje al mismo personaje diciéndole que se mueva hacia la izquierda.

- **BloqueIzquierda:** este clase implementa la interfaz Bloque, al ejecutarse se encarga de decirle al personaje recibido por parámetro que se dirija hacia la izquierda. Análogamente, al ejecutarse invertido, le envía un mensaje al mismo personaje diciéndole que se mueva hacia la derecha.

- **BloqueLapizApoyado:** esta clase es un bloque que implementa la interfaz Bloque, su método ejecutar lo implementa enviándole el mensaje apoyarLapiz al personaje recibido por parámetro. Este mensaje, cambia el atributo lápiz del personaje por una nueva instancia de la clase LapizApoyado. Análogamente, su método ejecutar invertido, le envía el mensaje levantarLapiz al mismo personaje, cambiándole su atributo lápiz por una nueva instancia de la clase LapizLevantado.

- **BloqueLapizLevantado:** esta clase también se encarga de implementar la interfaz Bloque y su funcionamiento es el opuesto al bloque descripto anteriormente. Su método le envía el mensaje levantarLapiz al personaje recibido por parámetro. Este mensaje, cambia el atributo lápiz del personaje por una nueva instancia de la clase LapizLevantado. Análogamente, su método ejecutar invertido, le envía el mensaje apoyarLapiz al mismo personaje, cambiándole el su atributo lápiz por una nueva instancia de la clase LapizApoyado.

- **BloquePersonalizado:** la particularidad de este bloque reside en que nos permite almacenar el "algoritmo" creado por el usuario bajo un cierto nombre. El nombre del bloque personalizado será provisto por el usuario al momento de la creación de este objeto. De esta manera, el usuario podrá crear tantos conjuntos de instrucciones como le parezca, y guardarlas bajo el nombre que más le guste, para así, poder usarlas más cómodamente en un futuro. La única restricción es que no se podrá crear algoritmos personalizados que no cuenten con ninguna instrucción, como se detalló en la sección supuestos del presente informe. Su funcionamiento es el siguiente, se crea bajo un cierto nombre, y luego se le agrega el conjunto de bloques que se desea almacenar mediante el método agregarAlgoritmo.

- **ConjuntoBloques:** a esta entidad se la puede pensar como un algoritmo, se encarga de modelar y almacenar una cierta cantidad de bloques. Cuenta con una lista de bloques como atributo, y con métodos que nos permiten agregar o removerle bloques. Al mismo tiempo, esta clase concreta implementa la interfaz Bloque. Por esta misma razón, comprende los mensajes ejecutar y ejecutarInvertido. Al recibir el mensaje ejecutar, simplemente recorre la lista de bloques y le envía el mensaje ejecutar a cada bloque contenido en esta, aquí queda en evidencia una de las grandes ventajas de haber implementado polimorfismo. Análogamente, al ejecutar invertido, le envía el mensaje ejecutar invertido a cada bloque contenido en su atributo. Optamos por modelar esta entidad, debido a que existen dos bloques, los cuales se describirán a continuación, en los que se requiere trabajar con varios bloques en simultáneo.

- **BloqueInvertir:** es una clase hija de la clase ConjuntoBloques, por ende, comprende tanto los mensajes agregarBloque, removerUltimoBloque, ejecutar y ejecutarInvertido, entre otros. Su función es tal que, al ejecutarse, todos los bloques que contenga ejecuten sus inversos correspondientes. Para lograr esto, al recibir el mensaje ejecutar, esta entidad le envía el mensaje ejecutarInvertido a todos los bloques presentes en la lista. A su vez, como esta entidad es un bloque, comprende el mensaje ejecutarInvertido, al recibirlo, le envía el mensaje ejecutar a cada uno de sus bloques.

- **BloqueRepetir:** esta clase también hereda de ConjuntoBloques, y por esa razón comprende los mismos mensajes que BloqueInvertir. Su función es repetir una cierta cantidad de veces las instrucciones de los bloques que contenga. La cantidad de iteraciones que realizara se definen al momento de su inicialización, recibiendo por parámetro un número entero en su constructor. Si bien en el trabajo practico solo se solicita los bloques para repetir dos y tres veces, este modelo nos permite tratar ambos casos de la misma manera y trabajarlos como si fueran instancias distintas de esta misma clase. Además de esta forma se permite que el código sea más abierto a futuras modificaciones, ya que si el día de mañana, quisiera crear un bloque que me permita repetir cinco veces, podría lograrlo con esta misma clase, sin necesidad de andar modificando el código. Al ejecutarse, se ejecutan secuencialmente todos los bloques contenidos en su lista, la cantidad de veces especificadas al momento de su creación. Análogamente, pasa con el ejecutar inverso, como se describió en la sección de supuestos.

7.3. Patrones de diseño utilizados para implementar la interfaz grafica

Para llevar a cabo la interfaz gráfica, hicimos uso del patrón de arquitectura denominado Modelo-Vista-Controlador (MVC), diseñado especialmente para la construcción de interfaces de usuario. La principal motivación de este patrón es la segregación de responsabilidades, para esto se propone separar el sistema en tres objetos: modelo, vista y controlador. El modelo es el dominio de nuestra aplicación, con su lógica y sus entidades de negocio. La vista son las formas en la que los objetos del modelo se muestran y se representan al usuario. Por último, el controlador define como la interfaz de usuario reacciona ante las acciones del usuario. De los muchos tipos de patrones diferentes de MVC que existen, nosotros aplicamos el patrón conocido como patrón Observer. Este patrón define una dependencia uno a muchos entre objeto de manera tal que cuando un objeto cambie su estado, todas sus dependencias sean notificadas y actualizadas automáticamente. Su motivación es mantener la consistencia entre objetos que dependen entre sí, reduciendo el acoplamiento y preservando la reusabilidad.

8. Excepciones

Excepción AlgoritmoVacioException. Dado que en la consigna del trabajo se especifica claramente que, al momento de crear el algoritmo personalizado, el algoritmo a guardar debe contar con al menos un bloque, optamos por crear esta excepción. Se lanza cuando se intenta guardar un algoritmo personalizado que este vacío, es decir, no cuenta con ningún bloque. También se utiliza en caso de que se desee remover el último bloque de un conjunto de bloques vacíos.