FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Curso 1
Segundo cuatrimestre de 2020

Alumno:	Stahl, Camila	103348	cstahl@fi.uba.ar
Alumno:	Watson, Francisco	105327	fwatson@fi.uba.ar
Alumno:	Loscocco, Ignacio	104002	iloscocco@fi.uba.ar
Alumno:	Vilardo, Ezequiel	104980	evilardo@fi.uba.ar

Índice

1. Introducción **2**

2. Supuestos **2**

3. Modelo de Clases **3**

4. Modelo de Secuencia **4**

5. Diagramas de Estado **7**

6. Diagramas de Paquete **7**

7. Detalles de Implementación **9**
7.1. Implementación del patrón Strategy en la clase Pincel 9
7.2. Implementación del patrón Observer en la clase Personaje 10

8. Excepciones **10**

1. Introducción

El presente informe reune la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar una aplicación de dibujo por bloques en Java utilizando los conceptos del paradigma de la orientación a objetos vistos hasta ahora en el curso.

Para este fin, hicimos uso en principio el patrón de diseño MVC, que divide la lógica del programa en tres elementos interconectados. De esta manera, podemos disociar la representación interna y la lógica del modelo de la forma en la que esa lógica es mostrada al usuario.

Por otro lado, para la realización del trabajo práctico hicimos uso de todos los conceptos teóricos vistos a lo largo de la materia. Para dar ejemplos:

- El modelado del programa en objetos y clases, siguiento los principios SOLID.
- El uso de polimorfismo entre los distintos objetos del modelo.
- El uso de Mocks para disociar las dependencias de las clases a la hora de hacer pruebas.
- Aplicar pruebas unitarias y de integración en nuestras clases.
- Utilizar patrones de diseño para resolver los problemas que fuimos encontrando en el modelo, que son recurrentes en la programación orientada a objetos, de manera flexible.

2. Supuestos

- Si el algoritmo personalizado se encuentra vació, al intentar guardar un algoritmo lanza la excepción RepetibleNoTieneAccionesTodaviaExcepcion.
- No se puede almacenar un algoritmo sin nombre.
- No se puede almacenar un algoritmo con el mismo nombre que otro.
- Cuando se presiona el botón ejecutar, la pila de bloques agregados se limpia automáticamente.
- El tablero se va agrandando a medida que el usuario hace movimientos, no tiene limite.
- Luego de que el personaje finaliza la secuencia, el personaje se mantendrá en esa posición para la próxima secuencia.
- El personaje inicia con el pincel arriba.
- Cuando se baja el pincel, se pinta la celda en la que se encuentra.

3. Modelo de Clases

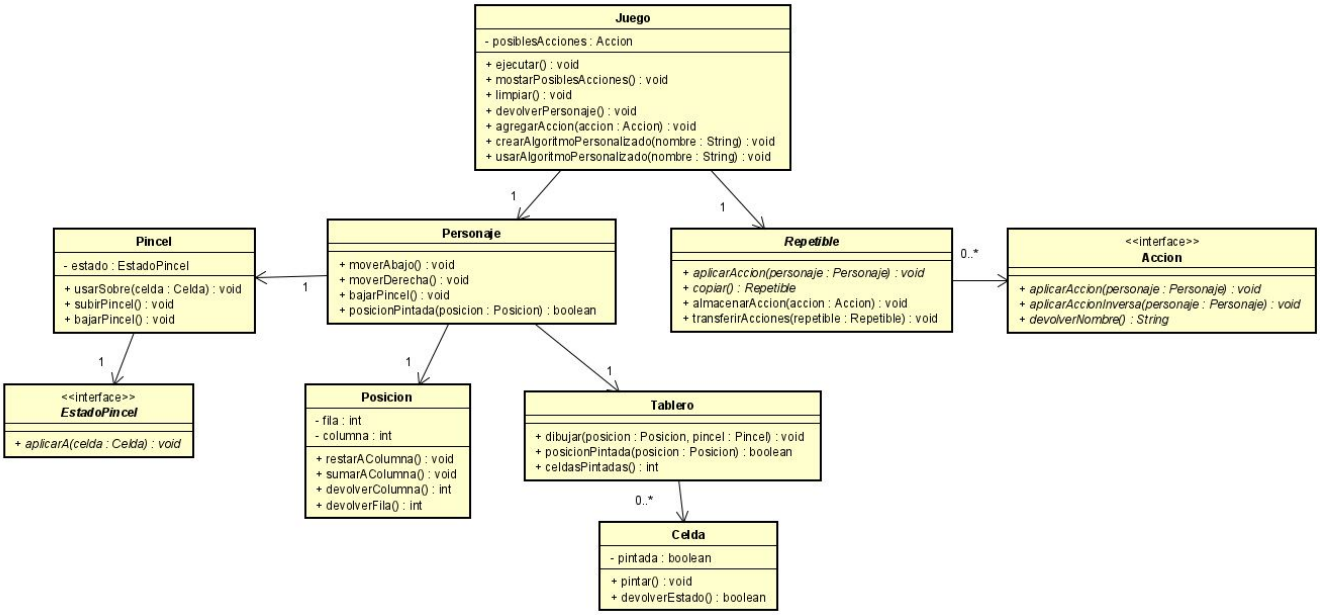

Figura 1: Diagrama de clases general.

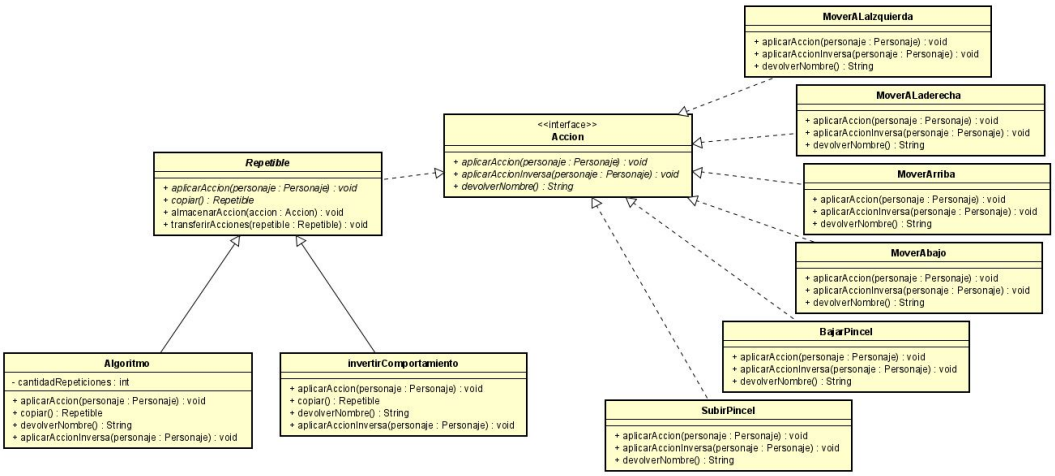

Figura 2: Diagrama de clases Reptible y Accion.

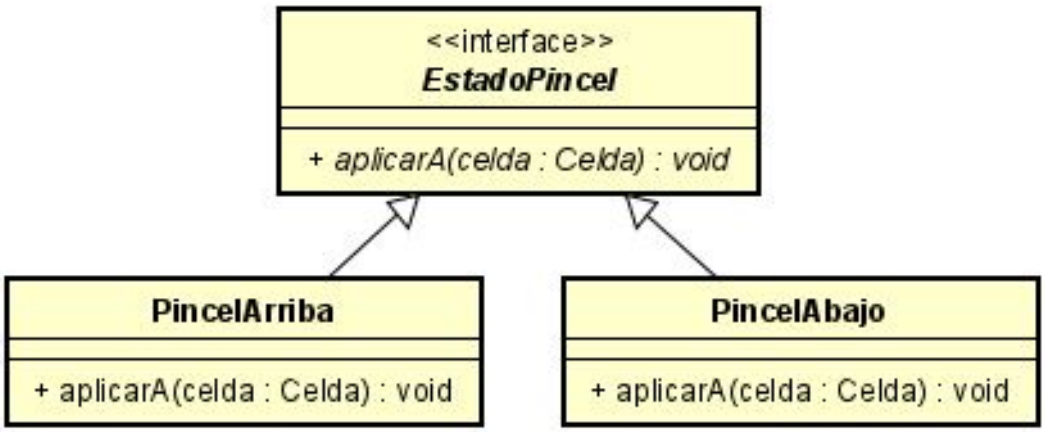

Figura 3: Diagrama de clases EstadoPincel.

4. Modelo de Secuencia

El diagrama a continuación representa una secuencia en la cual se crea el juego, se añade un bloque de bajar pincel para que pinte y un bloque de mover a la derecha y se ejecuta el algoritmo.

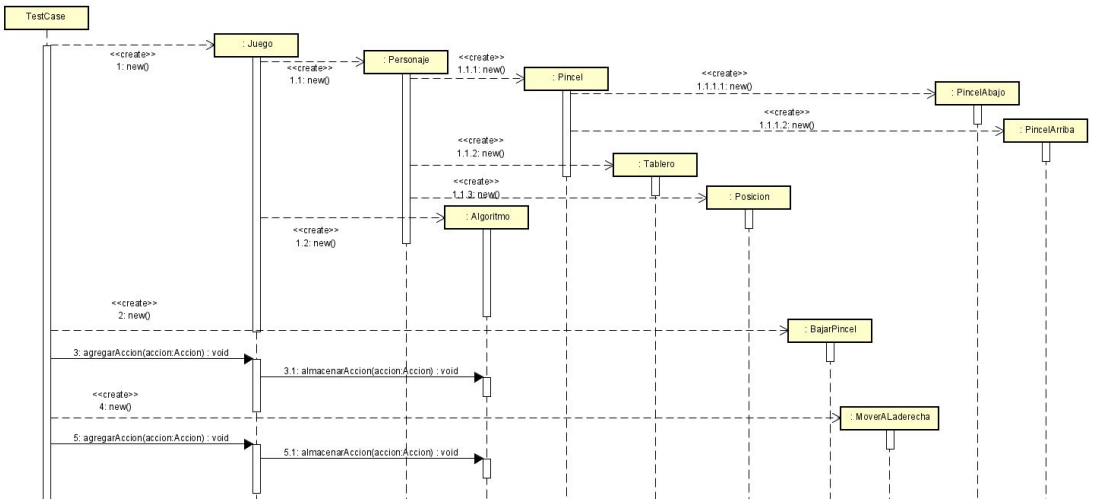

Figura 4: Diagrama de secuencia new + agregarBloque.

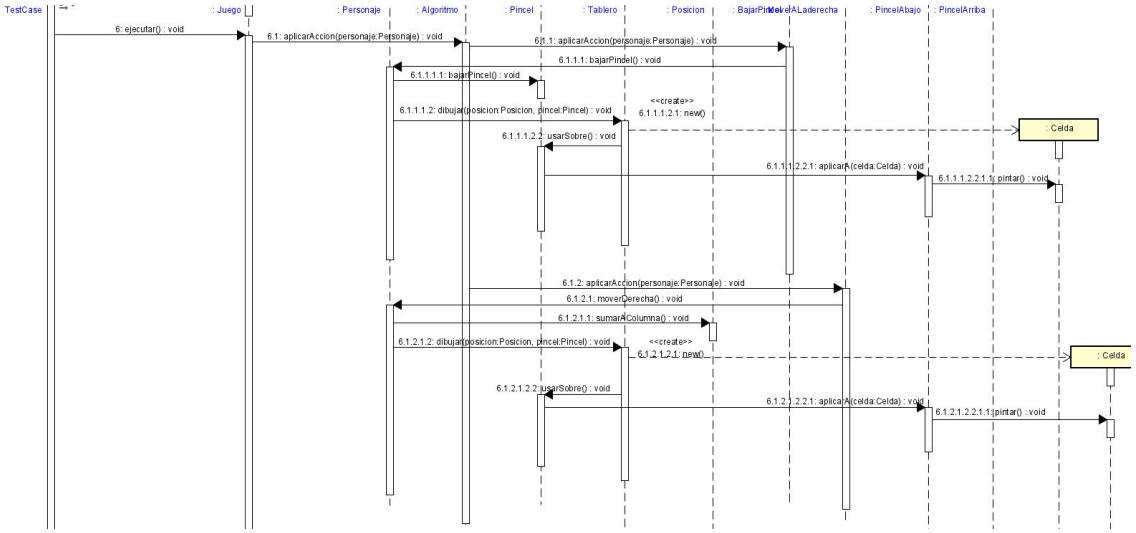

Figura 5: Diagrama de secuencia ejecutar.

El siguiente diagrama de secuencia incluye nuevamente el new de un Juego nuevo, y el intento de crear un algoritmo personalizado sin haber agregado bloques, lo que devuelve RepetibleNoTieneAccionesTodaviaExcepcion.

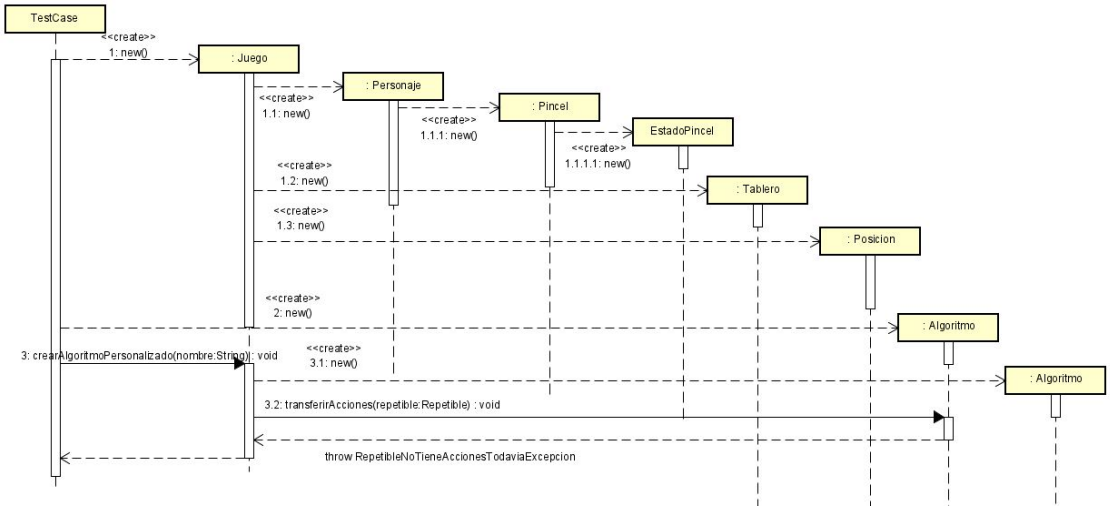

Figura 6: Diagrama de secuencia Crear Algoritmo personalizado error.

A continuacion vemos otro diagrama de secuencia que cuenta ya con la instancia de juego ya iniciada y un bloque de MoverALaderecha agregado, ahora si se crea el algoritmo personalizado, y muestra como se lo agregararia a la pila de ejecucion.

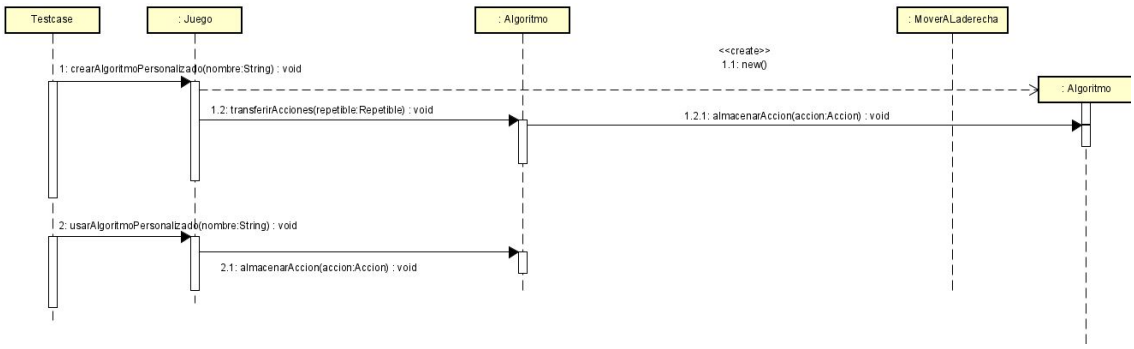

Figura 7: Diagrama de secuencia crear y ejecutar algoritmo personalizado.

Y en este ultimo diagrama de secuencia, volvemos a partir de un Juego ya creado, y le agregamos un bloque RepetirPorDos y mostramos como seria su ejecucion.

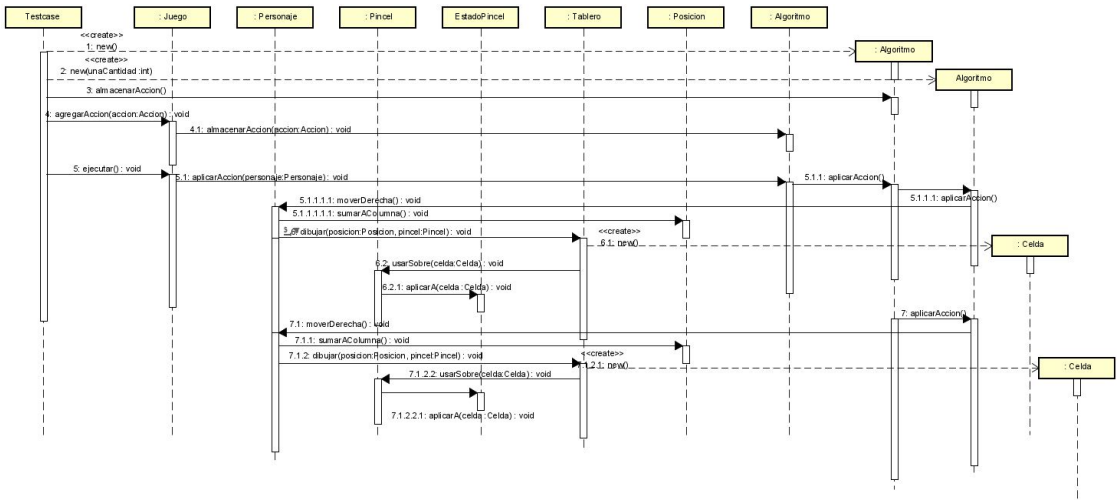

Figura 8: Diagrama de secuencia crear y usar bloque repetir por 2.

5. Diagramas de Estado

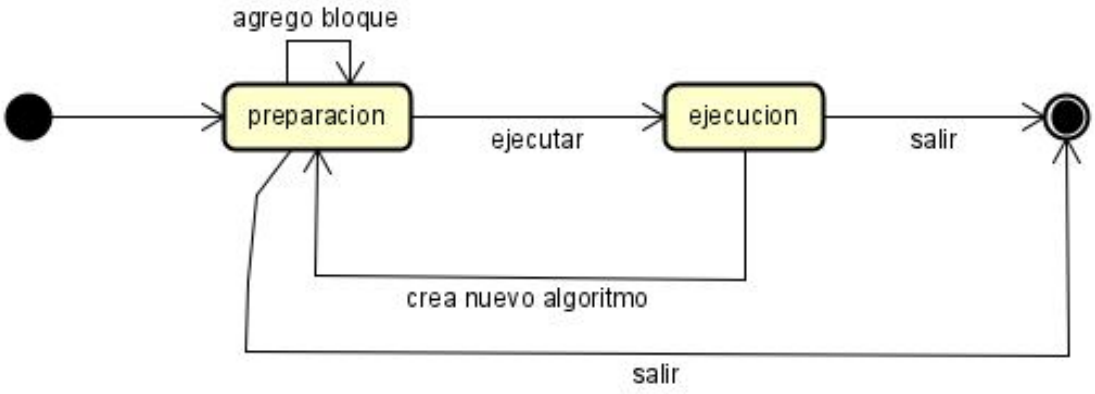

Figura 9: Diagrama de estado de la instancia de juego.

6. Diagramas de Paquete

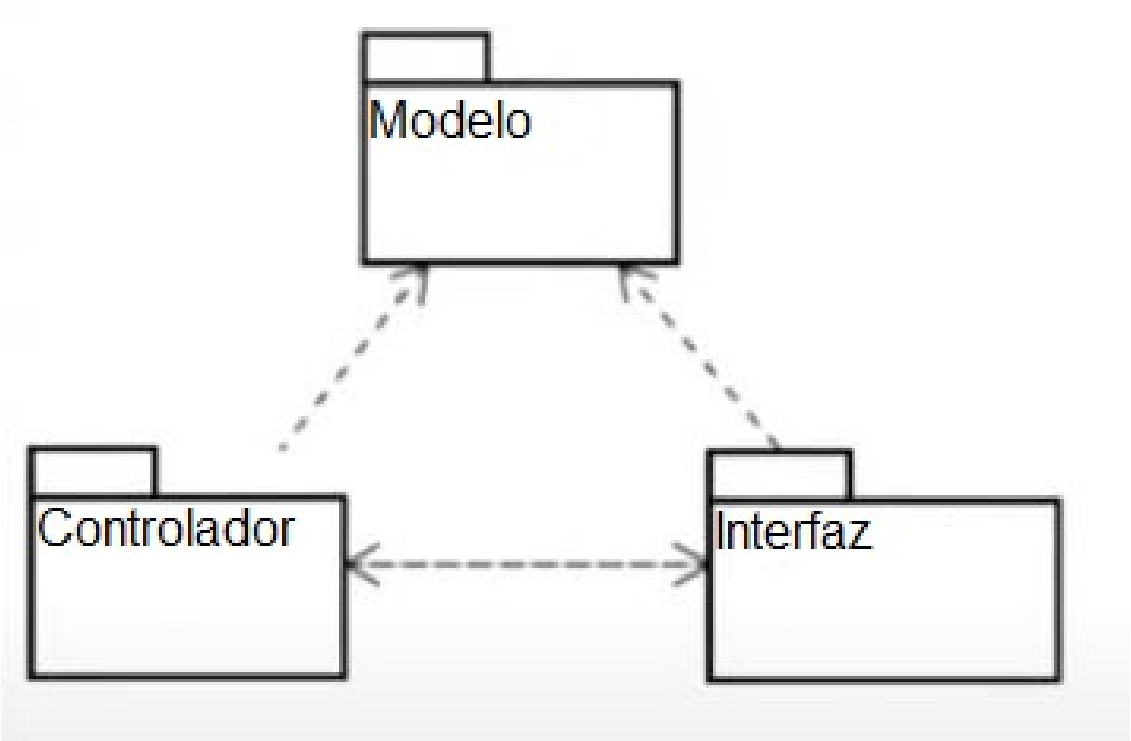

Figura 10: Diagrama de paquetes 1.

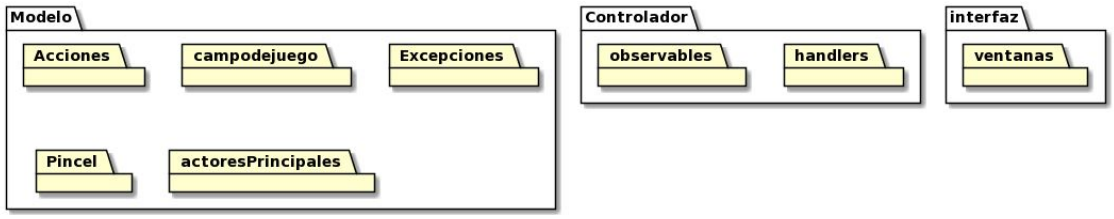

Figura 11: Diagrama de paquetes 2.

7. Detalles de Implementación

7.1. Implementación del patrón Strategy en la clase Pincel

En primer lugar consideramos que es importante explicar el detalle de la implementación de uno de los patrones de diseño vistos a lo largo del cuatrimestre en la clase Pincel. El pincel que posee el personaje en este trabajo práctico tendrá un comportamiento bastante particular puesto que de acuerdo a la consigna del trabajo, el personaje dejará pintadas todas las celdas sobre las que camine cuando su pincel esté bajo; caso contrario no pintará las celdas. Nosotros encontramos el patrón strategy ideal para encarar este tipo de situaciones, en las que una clase cambia su comportamiento de manera flexible y en tiempo de ejecución, sin uso de booleanos o condicionales que alarguen y llenen el código de 'bad smells'.

Cuando hablamos de 'bad smells' nos referimos a que si no implementamos este patrón de diseño estamos en principio violando el Tell-Don't-Ask principle al llenar de condicionales la clase para cambiar su comportamiento. En segundo lugar, estaríamos violando el Open/Closed principle que establece que las clases deberían estar abiertas a la extensión pero cerradas para su modificación.

El patrón strategy también nos permite desacoplar el comportamiento de la clase que implementa ese comportamiento. La clase pincel se "desentiende"de cómo pintar una celda, pero a la vez está abierta a la extensión de más algoritmos de pintado de celdas.

Aquí mostramos una parte de la clase pincel para que se entienda de manera más clara cómo funciona la implementación del patrón strategy.

```
public class Pincel {

    private EstadoPincel estadoArriba = new PincelArriba();
    private EstadoPincel estadoAbajo = new PincelAbajo();
    private EstadoPincel estado = estadoArriba;

    public void usarSobre(Celda celda){
        estado.aplicarA(celda);
    }

    public void subirPincel(){
        estado = estadoArriba;
    }

    public void bajarPincel(){
        estado = estadoAbajo;
    }

}
```

Como podemos ver nuestra implementación de la clase Pincel tiene dos estados y comienza estando arriba. Vemos que tanto cuando se baja como cuando se sube el pincel se cambia el estado del pincel, el cual maneja el comportamiento del pintado de celdas.

7.2. Implementación del patrón Observer en la clase Personaje

Para poder visualizar de manera gráfica el movimiento del personaje nosotros tuvimos que hacer uso del patrón de diseño Observer. Hicimos que la clase personaje extienda de la clase Observable, de manera que cada vez que el personaje se mueva le notifique a sus observadores de ese movimiento.

A su vez, hicimos que la clase Area de dibujado, encargada de mostrar de manera gráfica el recorrido del personaje, sea la clase que esté observando al personaje. Con esta implementación, cada vez que el personaje se mueve, se le notifica al área de dibujado de este cambio y así es como nosotros podemos mostrar en el gráfico las posiciones por las que pasa el personaje.

En cuanto la representación de las celdas pintadas, decidimos hacer uso del unico condicional if de nuestro modelo. Dentro de la clase Area de dibujado, cuando se le notifica que el personaje se ha movido, esta se encarga de 'pintar' la celda dependiendo de si el pincel del personaje esta en estado bajo.

Con la ayuda del patrón observer, pudimos resolver el problema de actualizar el grafico con las nuevas posiciones a medida que el personaje se mueve. Descartamos otras alternativas que no hacían uso de dicho patrón porque llenaban de resposabilidades las demás clases de nuestro modelo y no desacoplaban el modelo de la vista, como sí nos lo permite hacer el patrón observer. Es decir, las demás clases de nuestro modelo tendrían que haberse encargado de registrar los cambios en las posiciones del personaje y comunicárselas a la vista, lo que no solo habría violado el encapsulamiento de la clase personaje pero también como ya dijimos se habrían acoplado estas clases y el modelo no sería amigable a la extensión.

8. Excepciones

- **NombreNoValidoParaAlgoritmoExcepcion** Esta excepcion la lanza la instancia de juego cuando se intenta crear un algoritmo personalizado sin nombre o con un nombre ya utilizado.
- **RepetibleNoTieneAccionesTodaviaExcepcion** Esta excepcion la lanza la instancia de algoritmo cuando se intentaban transferir las acciones de este al nuevo algoritmo personalizado.