UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE SEDE SANTO DOMINGO
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN - DCCO-SS
CARRERA DE INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

PERIODO : Abril 2023 – Octubre 2023

ASIGNATURA : POO

TEMA : Juego de Pacman

NOMBRES : Ordoñez Eduardo
Kelvin Quezada

NIVEL-PARALELO : Segundo “A”

DOCENTE : Ing. Cevallos Farias Javier Moyota

FECHA DE ENTREGA : 29/05/2023

Grupo 4

SANTO DOMINGO - ECUADOR

2023

Índice

1. Introducción

Según (Vazquez) la herencia es la que permite que una clase herede los atributos y métodos de otra clase y también se basan en la reutilización de código.

Mediante el señor (Fernandez, s.f.) El polimorfismo es la que permite que los objetos de distintas clases puedan ser trasladados de manera uniforme ya que se puede utilizar clases de diferentes a través de una interfaz común.

Según (Canelo, 2020) nos dice que los patrones de diseños son soluciones generales, reutilizables y se aplica en diferentes tipos de problemas del diseño del Software ya que se trata de plantillas que identifica un problema en el sistema y proporciona soluciones ya que los desarrolladores se enfrentan durante un periodo de tiempo a las pruebas y errores.

En este informe, se presentará un programa desarrollado en el lenguaje de programación Java que integra todos los temas aprendidos en relación a la POO como clases, objetos, herencia, polimorfismo, encapsulación y abstracción sobreescritura de métodos arreglos y JFrama.

2. Objetivos

2.1.Objetivos Generales

- Desarrollar un juego en Java aplicando los temas aprendidos y utilizar lo que es JFramen para crear interfaz por medio de consola.

2.2.Objetivos Específicos

- Documentar de manera clara y concisa el código del programa desarrollado, incluyendo comentarios y explicaciones detalladas de su funcionamiento, para facilitar su comprensión.
- Realizar una guía del ejercicio para poder comprender más el procedimiento.
- Usar código limpio como buenas prácticas de programación

3. Marco Teórico

3.1. Programación Orientado a Objeto

Según (Martínez, 2020) la programación orientada a objetos (POO) es un paradigma de programación, es decir, un patrón o estilo de programación que nos dice cómo usarlo. Las ideas de clases y objetos sirven como base. El software se organiza utilizando este estilo de programación como fragmentos de código sencillos y reutilizables (clases) para producir distintas instancias de objetos.

3.1.2. Diagramas de UML

Mediante el señor (Mancuzo, 2021) nos dice que el lenguaje de modelado unificado (UML) se creó para proporcionar un lenguaje de modelado visual general, semántica y sintácticamente rico para la arquitectura, el diseño y la implementación de la estructura y el comportamiento de los sistemas de software complejos. UML tiene usos además del desarrollo de software, p. gramo. Por ejemplo, en un flujo de proceso de la industria manufacturera.

- **Clases:** una clase, como su nombre lo indica, representa una clase en un paradigma orientado a objetos y es el elemento principal de un diagrama.
- **Casos de Uso:** (Cabrera, 2022) nos dice que un diagrama de casos de uso le permite visualizar las posibles interacciones que un usuario o cliente podría tener con el sistema.

3.1.3. Encapsulamiento

(Lara, 2015) nos da a conocer que el proceso de almacenar en una misma parte los elementos de una abstracción que conforman su estructura y comportamiento; se utiliza para separar la interfaz contractual de una abstracción de su implementación. Para el encapsulado, existen tres niveles de acceso, a saber:.

- público(public)

- protegido(protected)
- privado(private)

3.1.4. Constructores

Un constructor es un elemento de clase cuyo identificador corresponde a la clase en cuestión y cuyo propósito es implementar y controlar cómo se inicializan las instancias de una clase dada, ya que Java no permite que las variables miembro de nuevas instancias permanezcan inicializadas.

3.1.5. Métodos

Un método Java es una pieza de código que realiza una tarea relacionada con un objeto, un método es básicamente una función que pertenece a un objeto o una clase.

Set: Un método Java es una pieza de código que realiza una tarea relacionada con un objeto, un método es básicamente una función que pertenece a un objeto o una clase.

Get: El método get es un método público al igual que una colección, pero el método get se encarga de mostrar la propiedad del objeto o el valor de la propiedad encapsulado en la clase correspondiente, es decir, declarado o protegido por la palabra reservada private.

3.1.6. Código Limpio

Según (Paredes, 2022) el código limpio no es un conjunto rígido de reglas, sino un conjunto de principios que ayudan a crear un código intuitivo y fácilmente modificable. Intuitivo en este caso significa que cualquier desarrollador profesional puede entenderlo de inmediato. El código fácilmente personalizable tiene las siguientes características:

- La secuencia de ejecución de todo el programa es lógica y la estructura es simple.
- Las conexiones entre los diversos componentes del código son evidentes.

- La tarea o función de cada clase, función, método y variable se puede entender de un vistazo.

3.2. Generalización/ Especialización

Según (Vazquez) la relación de especialización/generalización (o de herencia) entre dos clases. Se cree que los lenguajes POO suelen tener relaciones como esta. Una relación de herencia de la clase B (subclase, o clase hija) con respecto a (superclase o clase base) nos permite decir que la clase B obtiene todos los métodos y atributos de la clase A, y que luego puede añadir algunas características propias.

En el escenario anterior, se supone que usaremos la relación de herencia para afirmar que la clase B descendió de la clase A. A través de este mecanismo, la clase B puede compartir todos los rasgos (estados, comportamientos y atributos) de la clase. A. Esto no impide que se le pueda añadir a la clase B características adicionales (de nuevo, tanto atributos como métodos), o incluso, se modifique el comportamiento (la definición, no la declaración) de alguno de los métodos heredados.

La Herencia

Las instancias de una clase incluyen también las de su superclase o superclases. Como resultado, además de los atributos y métodos específicos de la clase, también utilizan los definidos en la superclase.

Esta capacidad se llama herencia, lo que significa que una clase hereda las propiedades y funciones de sus superclases para que sus instancias puedan usarlas. Al colocar el símbolo delante de los atributos y métodos heredados en las subclases, UML brinda la opción de representarlos.

3.2.1. Implementación

Según Sols (2015), la programación y la implementación de clases funcionan para adaptarse a la forma en que las personas piensan y abordan los problemas. Las clases son abstracciones que representan un grupo de objetos con un comportamiento y una interfaz de usuario similares. La declaración y el cuerpo de la clase son las dos partes que componen la implementación de una clase.

Ejemplo de sintaxis

Definidos los tipos y funciones pasamos a la declaración e implementación de clases con ello podemos identificar la estructura de la programación.

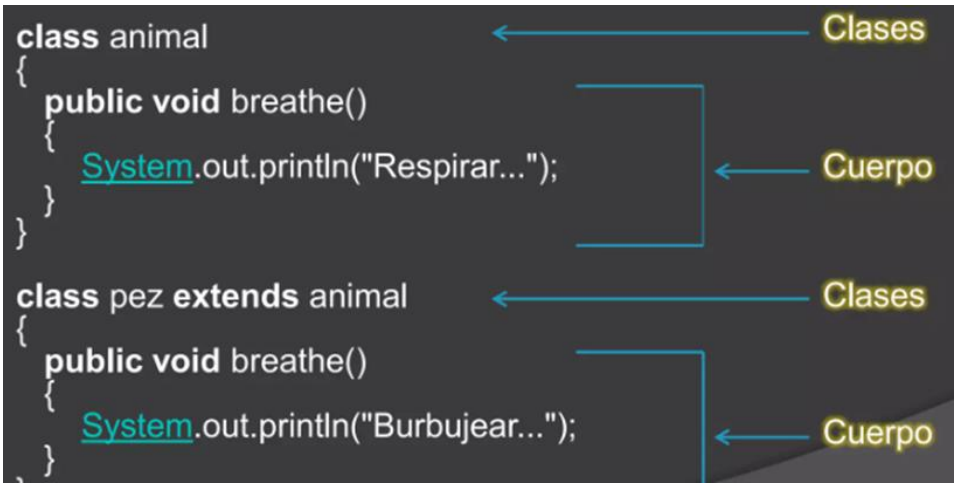

Imagen 1 Ejemplo de sintaxis de Sols

3.3. Gestión de Defecto testing

3.3.1. Verificacion y Validacion

Los procesos de verificación y análisis que aseguran que el software que se está desarrollando está en línea con su especificación y satisface las necesidades de los clientes se denominan verificación y validación. Un proceso de ciclo de vida completo es VandV. Las revisiones de los requisitos vienen primero, luego las revisiones de los diseños y el código, y finalmente las pruebas del producto. En cada etapa del proceso de desarrollo de software, se llevan a cabo actividades de V&V. A pesar de la facilidad con la que pueden confundirse, Boehm (1979) describió sucintamente la diferencia entre verificación y validación de la siguiente manera.

Verificación: las responsabilidades de Verificación incluyen asegurarse de que el software cumpla con sus especificaciones. Se confirma el cumplimiento del sistema con los requisitos funcionales y no funcionales establecidos.

La validación es un proceso que se usa más ampliamente: El software necesita ser revisado para ver si cumple con las expectativas del cliente. El software se prueba para ver si funciona como el usuario espera en lugar de lo especificado, lo que va más allá de determinar si el sistema cumple con sus especificaciones.

La fase de análisis de requisitos del sistema es propensa a errores y omisiones, y cuando esto sucede, el software final a menudo no cumple con las expectativas del cliente. Sin embargo, en realidad, no todos los problemas que presenta una aplicación no se pueden encontrar a través de la validación de requisitos. Cuando el sistema se haya implementado por completo, es posible que se encuentren algunos errores en los requisitos. (Drake, 2009)

3.3.2. Polimorfismo

Según (Fernandez, s.f.) al crear objetos con comportamientos compartidos, el polimorfismo nos permite procesar objetos de diversas formas. Implica tener la capacidad de mostrar la misma interfaz de usuario para varios formularios o tipos de datos subyacentes. Los objetos pueden reemplazar comportamientos primarios comunes con comportamientos secundarios particulares mediante el uso de la herencia. La sobrecarga de métodos y la anulación de métodos son dos formas en que el polimorfismo permite que el mismo método lleve a cabo varios comportamientos.

Las clases con tipos compatibles se pueden usar en cualquier parte de nuestro código gracias al polimorfismo. La compatibilidad de tipos en Java se refiere a cómo una clase se extiende a otra o cómo una clase implementa una interfaz. Informalmente: Podemos enlazar referencias de sus hijas a una referencia de tipo padre. Cualquier instancia de una clase que implemente la interfaz se puede conectar a una referencia de tipo de interfaz.

3.4. Modelo Vista controlador

3.4.1. El Modelo

El señor (Pantoja, 2004) el modelo es un conjunto de clases que representa la información del mundo real que debe procesar el sistema. Sin tener en cuenta los métodos por los cuales se generarán los datos o cómo se mostrarán, es decir, sin tener una conexión con otra entidad dentro de la aplicación. Las vistas y un controlador están ocultos para el modelo, que no los conoce. SmallTalk sugiere que el modelo en realidad consta de dos submódulos: el modelo de dominio y el modelo de red. Si bien ese enfoque suena intrigante, no es práctico porque debe haber interfaces que permitan que los módulos se comuniquen entre sí. solicitud.

3.4.2. La Vista

Las vistas se encargan de mostrar al usuario los datos del modelo. Las vistas y el modelo tienen una relación de muchos a uno, lo que significa que cada vista tiene un modelo asociado, aunque puede haber varias vistas vinculadas a un solo modelo. De esta forma, por ejemplo, podría tener una vista que muestre la hora del sistema como un reloj analógico y otra vista que muestre la misma información como un reloj digital. La vista solo requiere los datos necesarios del modelo para ejecutar una visualización. La vista se altera a través de notificaciones que produce el modelo de aplicación cada vez que se realiza una acción que implica un cambio en el modelo de dominio. En pocas palabras, cuando se produce una modificación, la representación visual del modelo vuelve a dibujar los componentes necesarios.

3.4.3. El Controlador

Mediante el señor (Hernandez, 2021) la responsabilidad del controlador es extraer, modificar y proporcionar datos al usuario. Esencialmente, el controlador es el enlace entre y el modelo a través de las funciones getter y setter, el controlador extrae datos del modelo e inicializa las vistas si hay alguna actualización desde las vistas, modifica los datos con una función setter.

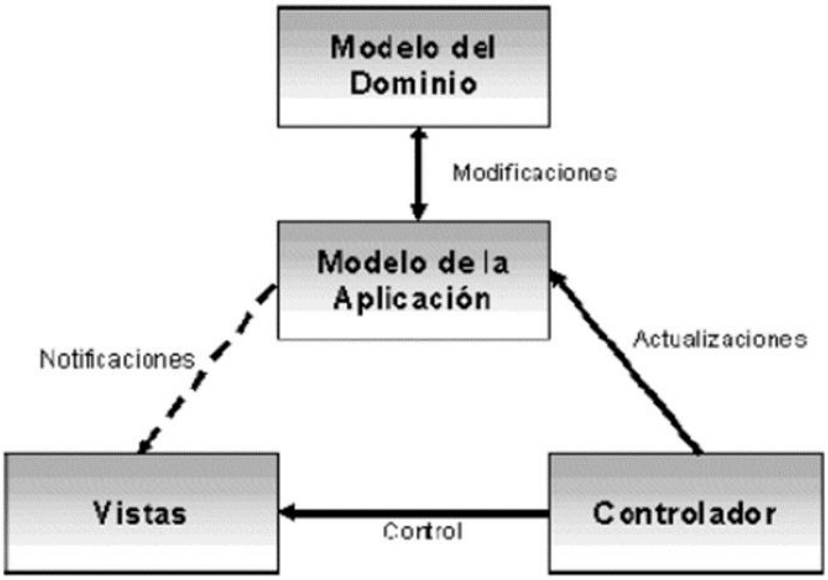

Imagen 2 Modelo Vista controlador de Pantoja

3.5. Programación modular

Según (Lara, 2015) el modularidad consiste en dividir un programa por medio de módulos ya que se puede compilarse por separados y tendrá conexiones mediante otros módulos. La modularidad también cuenta con tres principios que son: Capacidad de descomponer un sistema complejo: Un sistema se puede dividir en un subprograma por módulos de problemas porque se puede dividir en problemas más pequeños. Capacidad de descomponer un sistema complejo: Dado que se utilizan algunos módulos y el programa está destinado a ser reutilizable, sugiere que puede entenderse cuando se crea mediante software.

```
// Nos dara a conocer las categoría de los modulos
case "A": maxper=8;break;
case "B": maxper=6;break;
case "C": maxper=4;break;
case "D": maxper=2;break;
```

Imagen 3 Programación modular por Kelvin Quezada

Se conocer la descomposición de un sistema complejo de los módulos por Kelvin Quezada.

Comprensión de sistema en parte: Se encarga de separar las partes que nos ayuda a la comprensión del código del sistema mediante la modificación.

```java
/*Esta funcion calcula y devuelve el numero maximo de personas permitidas en una categoria especificada.
Recibe como entrada el codigo de categoria 'cate' y devuelve el numero maximo de personas 'maxper'*/
    static int MaximoPersonas(String cate){
        // Inicializa la variable que almacenara el numero maximo de personas permitidas.
        int maxper=0;
        // Utiliza un switch para asignar el valor correcto de 'maxper' segun la categoria especificada.
        switch(cate){
            // Nos dara a conocer las categoria
            case "A": maxper=8;break;
            case "B": maxper=6;break;
            case "C": maxper=4;break;
            case "D": maxper=2;break;

        }
        // Devuelve el numero maximo de personas permitidas segun la categoria especificada.
        return maxper;
    }
/*Esta funcion calcula y devuelve el numero maximo de PagoMensual permitidas en una categoria especificada.
Recibe como entrada el codigo de categoria 'cate' y devuelve el numero maximo de pagoMenusal 'maxper'*/
    static double PagoMensual(String cate){
        double pm=0;
        switch(cate){
            case "A": pm=40.00;break;
            case "B": pm=30.00;break;
            case "C": pm=20.00;break;
            case "D": pm=10.00;break;
        }
         // Devuelve el numero maximo de personas permitidas segun la categoria especificada.
        return pm;
    }
```

Imagen 4 Programación modular por Kelvin Quezada

Nos ayuda a separar nuestros módulos en partes diferentes para conocer las categorías de cada uno de los módulos por Kelvin Quezada.

3.5.1. ¿Cuáles son las características de la programación modular?

Según (Jose, 2023) el encapsulamiento ya que cada módulo es una unidad independiente. La modularidad ya que permite los bloques de códigos que son pequeños y muy fáciles de comprender. La abstracción mediante módulos se utiliza para proporcionar un aislamiento de los procesos subyacentes que ya que el código sea más fácil de comprender. La reutilización ya que cada pieza de código se puede utilizar en otros proyectos lo que ayuda a reducir el tiempo y los costos del desarrollo del programa. Desacoplamiento por medio de módulos no dependen entre si y se pueden cambiar y actualizar de forma aislada si afectar a otros módulos.

3.6. Qué son los patrones de diseños

Según (Canelo, 2020) los patrones de diseño, lo podemos conocer como design patterns, son soluciones generales que son reutilizables aplicables a los distintos problemas de diseño de software. Estas plantillas identifican problemas comunes en el sistema y ofrecen soluciones probadas a través de la experiencia acumulada a lo largo del tiempo. Mediante (Sánchez, 2017) cada patrón de diseño tiene un propósito específico y aborda problemas comunes de diseño. Es posible que estemos ante un patrón de diseño de software si el modelo de solución encontrado es adaptable a varios campos. Si la eficacia de estos modelos de solución se ha demostrado al resolver problemas análogos en el pasado, se consideran patrones de diseño. Su reutilización también debe estar probada. Daremos a conocer las ventajas y desventajas de los patrones de diseño:

Ventajas

- Reutilización de soluciones probadas.
- Mejora de la calidad del código.
- Facilita la comunicación y colaboración en el equipo.

Desventajas

- Puede introducir complejidad innecesaria.
- Existe el riesgo de caer en el sobre diseño.

3.6.1. Tipos de patrones de diseños

Hay un total de 23 patrones de diseño diferentes, y las tres categorías más populares contienen los patrones de diseño más utilizados. Estos son los cuatro grandes grupos:

- Patrones en la creación.
- Patrones arquitectónicos.
- Patrones de comportamiento

- Comportamiento

Imagen 5 Patrones de diseños por Canelo

3.6.2. Patrones de creación

Los patrones de compilación ofrecen una variedad de mecanismos de creación de objetos que aumentan la flexibilidad y permiten la reutilización adecuada de la situación del código preexistente. Como resultado, el programa tiene más libertad para elegir qué objetos crear para un caso de uso particular. Estos son los patrones de la creación.

1. **El patrón Abstract Factory:** Se utiliza mediante interfaz para asi crear conjuntos de objetos que son relacionados en las diferentes clases sin especificar el nombre.
2. **Buider Patterns:** Abstrae la construcción metódica de objetos complejos, permitiendo varias representaciones del mismo proceso de construcción.
3. **Factory Method:** especifica una interfaz para crear objetos, pero permite que las subclases elijan qué clase instanciar.
4. **Prototype:** permite la duplicación de objetos sin necesidad de que su código dependa de las clases de los objetos originales.
5. **Singleton:** asegura que una clase tenga solo una instancia y ofrece un único punto de acceso global.

3.6.3. Patrones estructurales

Según (Soto, 2021) el objetivo de los patrones estructurales es facilitar el ensamblaje de elementos de clases estructurales más grandes manteniendo la flexibilidad y la eficiencia.

1. **Adapter:** Adaptador se utiliza para que objetos mediante interfaces incompatibles entre sí.
2. **Bridge:** resuelve un problema habitual en la herencia ya que se divide en clases relacionando entre dos jerarquías diferentes: la implementación y abstracción.
3. **Composite:** Solo se recomienda utilizar Compuesto cuando el modelo de código.
4. **Decorator:** esta herramienta se usa para agregar funcionalidad a un objeto encapsulando objetos que ya tienen esa funcionalidad.
5. **Facade:** brinda a una biblioteca, marco u otro conjunto complicado de clases una interfaz de usuario optimizada.
6. **Flyweight:** una técnica para reducir el tamaño de los objetos que implica almacenar solo el estado intrínseco (también conocido como información constante) del objeto y distribuir el resto de la información (también conocido como estado extrínseco) entre varios objetos similares.
7. **Proxy:** La interfaz se utiliza para acceder a las funcionalidades de otras clases u objetos, y se utiliza para crear objetos que puedan representar esas funcionalidades.

3.2.4. Patrones de Comportamiento

Los patrones de comportamiento tienen como objetivo mejorar la comunicación entre varias áreas.

1. **Chain of responsibility:** podemos permitir que más de un objeto responda a una solicitud evitando que se adjunte a un solo receptor.
2. **Command:** Se utiliza cuando es necesario encapsular dentro de un objeto todos los parámetros que una acción requiere para ejecutarse.

3. **Interpreter:** Utilizando Interpreter podremos evaluar un lenguaje a través de una interfaz que indique el contexto en el cual se interpreta.
4. **Iterator:** Este patrón de comportamiento se utiliza cuando necesitamos iterar en colecciones o conjuntos de objetos sin la necesidad de intercambiar información relevante.
5. **Mediator:** Se utiliza cuando necesitamos controlar las comunicaciones directas entre objetos y disminuir sus dependencias caóticas.
6. **Memento:** Este patrón es capaz de almacenar y restaurar la información de un objeto. Observer: A través de este patrón de comportamiento varios objetos interesados (suscriptores) en un objeto en particular (notificador) pueden recibir notificaciones de su comportamiento mientras estén suscriptos a sus notificaciones.
7. **State:** Se utiliza para modificar el comportamiento de una clase de objetos dependiendo del estado actual (comportamiento interno) de dichos objetos.
8. **Strategy:** Permite separar todos los algoritmos de una clase específica en nuevas clases separadas donde los objetos pueden intercambiarse.

3.7. Desarrollo

Se desarrollará un juego en el software Apache NetBeans que se utilizará JFrame y se utilizará solo código para así poder comprender cómo utilizar nuestra interfaz para la creación del juego y utilizamos temas de programación orientada a objeto utilizamos herencia polimorfismo, sobreescritura de métodos, abstracción, usos de interfaces y arreglos hasta la manipulación de gráficos.

3.7.1. Código del Programa

Clase tablero

```
public class tablero extends JPanel implements ActionListener {
// Declare instance variables
    private Dimension d; // Almacena el tamaño del tablero
    private final Font smallfont = new Font("Helvetica", Font.BOLD, 14); // Fuente para el texto en el juego
    private Image ii; // Almacena una imagen
    private final Color dotcolor = new Color(192, 192, 0); // Color de los puntos en el juego
    private Color mazecolor; // Color del laberinto
    private boolean ingame = false; // Indica si el juego está en curso o no
    private boolean dying = false; // Indica si el jugador está perdiendo una vida
    private final int blocksize = 24; //Tamaño de cada bloque en el laberinto.
    private final int nrofblocks = 15; //Número de bloques en cada fila/columna
    private final int scrsize = nrofblocks * blocksize; // Tamaño total de la pantalla basado en el tamaño del bloque y el número de bloques
    private final int pacanimdelay = 2;// Retraso de animación para Pacman
    private final int pacmananimcount = 4;// Número de fotogramas en la animación de Pacman.
    private final int maxghosts = 12;// Número máximo de fantasmas en el juego.
    private final int pacmanspeed = 6;// Velocidad de movimiento de Pacman.
    // Variables de animación
    private int pacanimcount = pacanimdelay;
    private int pacanimdir = 1;
    private int pacmananimpos = 0;
    private int nrofghosts = 6;// Número actual de fantasmas.
    private int pacsleft, score;// Vidas restantes y puntuación.
    private int[] dx, dy;// Matrices para el movimiento de fantasmas.
    // Matrices para atributos fantasma
    private int[] ghostx, ghosty, ghostdx, ghostdy, ghostspeed;
// Imágenes para elementos del juego.
    private Image ghost;
    private Image pacman1, pacman2up, pacman2left, pacman2right, pacman2down;
    private Image pacman3up, pacman3down, pacman3left, pacman3right;
    private Image pacman4up, pacman4down, pacman4left, pacman4right;
// Posición y movimiento de Pacman
    private int pacmanx, pacmany, pacmandx, pacmandy;
    private int reqdx, reqdy, viewdx, viewdy;
// Datos de diseño del laberinto
    private final short leveldata[] = {
        19, 26, 26, 26, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 22,
        21, 0, 0, 0, 17, 16, 16, 16, 16, 16, 16, 16, 16, 16, 20,
        21, 0, 0, 0, 17, 16, 16, 16, 16, 16, 16, 16, 16, 16, 20,
        21, 0, 0, 0, 17, 16, 16, 24, 16, 16, 16, 16, 16, 16, 20,
        17, 18, 18, 18, 16, 16, 20, 0, 17, 16, 16, 16, 16, 16, 20,
        17, 16, 16, 16, 16, 16, 20, 0, 17, 16, 16, 16, 16, 24, 20,
        25, 16, 16, 16, 24, 24, 28, 0, 25, 24, 24, 16, 20, 0, 21,
```

```
        1, 17, 16, 20, 0, 0, 0, 0, 0, 0, 0, 17, 20, 0, 21,
        1, 17, 16, 16, 18, 18, 22, 0, 19, 18, 18, 16, 20, 0, 21,
        1, 17, 16, 16, 16, 16, 20, 0, 17, 16, 16, 16, 20, 0, 21,
        1, 17, 16, 16, 16, 16, 20, 0, 17, 16, 16, 16, 20, 0, 21,
        1, 17, 16, 16, 16, 16, 16, 18, 16, 16, 16, 16, 20, 0, 21,
        1, 17, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 20, 0, 21,
        1, 25, 24, 24, 24, 24, 24, 24, 24, 24, 16, 16, 16, 18, 20,
        9, 8, 8, 8, 8, 8, 8, 8, 8, 8, 25, 24, 24, 24, 28
    };
// Velocidades de movimiento válidas para Pacman
    private final int validspeeds[] = {1, 2, 3, 4, 6, 8};
    private final int maxspeed = 6;
// Velocidad de movimiento actual de Pacman.
    private int currentspeed = 3;
    // Matriz para almacenar datos de diseño de laberinto.
    private short[] screendata;
    // Temporizador para bucle de juego.
    private Timer timer;

// Constructor de la clase "tablero".
public tablero() {
    // Inicializar el juego.
    // Cargar las imágenes necesarias para el juego.
    loadImages();
    // Inicializar las variables necesarias para el juego.
    initVariables();
    // Configurar el oyente clave para la entrada del usuario.
    addKeyListener(new TAdapter());
    // Hacer que el panel sea foco para que pueda recibir eventos de teclado.
    setFocusable(true);
    // Establecer el color de fondo del panel a negro.
    setBackground(Color.BLACK);
    // Habilitar el doble almacenamiento en búfer para lograr una representación fluida.
    setDoubleBuffered(true);
}

// Método para inicializar las variables del juego
/*
 Los arreglos ghostx, ghosty, ghostdx, ghostdy, ghostspeed, dx, y dy almacenan instancias de
diferentes tipos
(como int y Image). Esto también es una forma de polimorfismo, ya que se puede tratar a
objetos de diferentes
tipos a través de una interfaz común.
*/
    private void initVariables() {
 // Inicializar el arreglo que almacena la información de la pantalla (laberinto)
```

```
        screendata = new short[nrofblocks * nrofblocks];
        // Establecer el color del laberinto
        mazecolor = new Color(5, 100, 5);
        // Crear una instancia de Dimension para el tamaño de la pantalla
        d = new Dimension(400, 400);
        // Inicializar arreglos para los fantasmas
        ghostx = new int[maxghosts];
        ghostdx = new int[maxghosts];
        ghosty = new int[maxghosts];
        ghostdy = new int[maxghosts];
        ghostspeed = new int[maxghosts];
        // Inicializar arreglos para los movimientos de los fantasmas y Pacman
        dx = new int[4];
        dy = new int[4];
// Inicializar y configurar el temporizador del bucle del juego
        timer = new Timer(40, this);// Cada 40 milisegundos se invocará el actionPerformed()
        // Iniciar el temporizador
        timer.start();
    }
/// método addNotify() para inicializar el juego cuando se agrega a un contenedor
    @Override
    public void addNotify() {
        super.addNotify();
        initGame();
    }
    // Método para realizar la animación de la boca de Pacman
    private void doAnim() {
// Decrementar el contador de la animación
        pacanimcount--;
// Verificar si es momento de cambiar la animación
        if (pacanimcount <= 0) {
            // Reiniciar el contador de la animación
            pacanimcount = pacanimdelay;
            // Cambiar la posición de la animación de Pacman
            pacmananimpos = pacmananimpos + pacanimdir;
// Comprobar si la posición de la animación está en el último cuadro o en el primero
            if (pacmananimpos == (pacmananimcount - 1) || pacmananimpos == 0) {
                // Cambiar la dirección de la animación invirtiendo el signo de pacanimdir
                pacanimdir = -pacanimdir;
            }
        }
    }
// Método para ejecutar el ciclo principal del juego
    private void playGame(Graphics2D g2d) {
// Verificar si Pacman está en proceso de morir
        if (dying) {
```

```
// Realizar acciones relacionadas con la muerte de Pacman
            death();

        } else {
            // Mover a Pacman
            movePacman();
            // Dibujar a Pacman en la pantalla
            drawPacman(g2d);
            // Mover a los fantasmas
            moveGhosts(g2d);
            // Verificar el estado del laberinto
            checkMaze();
        }
    }
// Método para mostrar la pantalla de introducción
    private void showIntroScreen(Graphics2D g2d) {
// Establecer el color de fondo de la pantalla de introducción
        g2d.setColor(new Color(0, 32, 48));
        // Dibujar un rectángulo en el centro de la pantalla para el mensaje
        g2d.fillRect(50, scrsize / 2 - 30, scrsize - 100, 50);
        // Establecer el color de los bordes del rectángulo
        g2d.setColor(Color.white);
        // Dibujar el borde del rectángulo
        g2d.drawRect(50, scrsize / 2 - 30, scrsize - 100, 50);
        // Mensaje a mostrar en la pantalla de introducción
        String s = "Presiona s para empezar.";
        // Crear una fuente para el mensaje
        Font small = new Font("Helvetica", Font.BOLD, 15);
        // Obtener información de la fuente para medir el ancho del mensaje
        FontMetrics metr = this.getFontMetrics(small);
        // Establecer el color del texto
        g2d.setColor(Color.white);
        // Establecer la fuente para el texto
        g2d.setFont(small);
        // Dibujar el mensaje centrado en la pantalla de introducción
        g2d.drawString(s, (scrsize - metr.stringWidth(s)) / 2, scrsize / 2);
    }

    private void drawScore(Graphics2D g) {

        int i;
        String s;
        // Establecer la fuente y el color para la puntuación
        g.setFont(smallfont);
        g.setColor(new Color(96, 128, 255));
        // Crear un mensaje de puntuación en forma de cadena
```

```
    s = "Score: " + score;
    // Dibujar la puntuación en la pantalla
    // scrsize / 2 + 96 es la posición horizontal donde se dibujará la puntuación
    // scrsize + 16 es la posición vertical donde se dibujará la puntuación
    g.drawString(s, scrsize / 2 + 96, scrsize + 16);
    // Dibujar las vidas restantes de Pacman en la parte inferior
    // Itera a través de las vidas restantes y dibuja las imágenes de Pacman
    for (i = 0; i < pacsleft; i++) {
        // Dibuja la imagen de Pacman (pacman3left) en la posición (i * 28 + 8, scrsize + 1)
        g.drawImage(pacman3left, i * 28 + 8, scrsize + 1, this);
    }
}

private void checkMaze() {

    short i = 0;
    boolean finished = true;
    // Recorre todos los bloques en el laberinto
    while (i < nrofblocks * nrofblocks && finished) {
        // Verifica si el bloque contiene puntos (bits 4 y 5 están establecidos)
        if ((screendata[i] & 48) != 0) {
            // Aún hay puntos sin comer, por lo que no hemos terminado
            finished = false;
        }

        i++;
    }
    // Si todos los puntos han sido comidos (finished es verdadero)
    if (finished) {
        // Incrementa la puntuación en 50 puntos
        score += 50;
        // Si el número actual de fantasmas es menor que el máximo permitido
        if (nrofghosts < maxghosts) {
            nrofghosts++;// Incrementa el número de fantasmas
        }
        // Si la velocidad actual del juego es menor que la velocidad máxima permitida
        if (currentspeed < maxspeed) {
            currentspeed++; // Incrementa la velocidad del juego
        }
        // Inicia un nuevo nivel reiniciando el laberinto y los elementos del juego
        initLevel();
    }
}

private void death() {//murio
    // Reduce el número de vidas restantes
```

```
        pacsleft--;
        // Si el número de vidas restantes es igual a cero
        if (pacsleft == 0) {
            // Establece el estado del juego como "no en juego" (game over)
            ingame = false;
        }
        // Continúa con el mismo nivel reiniciando los elementos del juego
        continueLevel();
    }

/*
El uso de clases e interfaces abstrae el comportamiento de los objetos del juego, como Pacman
y los fantasmas.
se define un método moveGhosts que acepta una instancia de Graphics2D como parámetro y
se encarga de mover
y dibujar los fantasmas en pantalla.
*/
    private void moveGhosts(Graphics2D g2d) {//mover fantasmas
//creamo variables
        short i;
        int pos;
        int count;
        // Recorre todos los fantasmas
        for (i = 0; i < nrofghosts; i++) {
            // Verifica si el fantasma se encuentra en una posición de bloque completa
            if (ghostx[i] % blocksize == 0 && ghosty[i] % blocksize == 0) {
                // Calcula la posición actual del fantasma en la matriz screendata
                pos = ghostx[i] / blocksize + nrofblocks * (int) (ghosty[i] / blocksize);
                count = 0;
                // Verifica las direcciones disponibles para el movimiento del fantasma
                if ((screendata[pos] & 1) == 0 && ghostdx[i] != 1) {
                    dx[count] = -1;
                    dy[count] = 0;
                    count++;
                }

                if ((screendata[pos] & 2) == 0 && ghostdy[i] != 1) {
                    dx[count] = 0;
                    dy[count] = -1;
                    count++;
                }

                if ((screendata[pos] & 4) == 0 && ghostdx[i] != -1) {
                    dx[count] = 1;
                    dy[count] = 0;
                    count++;
```

```
            }

            if ((screendata[pos] & 8) == 0 && ghostdy[i] != -1) {
               dx[count] = 0;
               dy[count] = 1;
               count++;
            }
            // Verifica si el fantasma está atrapado en una posición sin movimientos válidos
            if (count == 0) {

               if ((screendata[pos] & 15) == 15) {
                  ghostdx[i] = 0;
                  ghostdy[i] = 0;
               } else {
                  ghostdx[i] = -ghostdx[i];
                  ghostdy[i] = -ghostdy[i];
               }

            } else {
               // Escoge aleatoriamente una de las direcciones disponibles
               count = (int) (Math.random() * count);

               if (count > 3) {
                  count = 3;
               }

               ghostdx[i] = dx[count];
               ghostdy[i] = dy[count];
            }

         }
         // Actualiza la posición del fantasma de acuerdo a su dirección y velocidad
         ghostx[i] = ghostx[i] + (ghostdx[i] * ghostspeed[i]);
         ghosty[i] = ghosty[i] + (ghostdy[i] * ghostspeed[i]);
         // Dibuja el fantasma en su nueva posición
         drawGhost(g2d, ghostx[i] + 1, ghosty[i] + 1);
         // Verifica si Pacman colisiona con un fantasma
         if (pacmanx > (ghostx[i] - 12) && pacmanx < (ghostx[i] + 12)
               && pacmany > (ghosty[i] - 12) && pacmany < (ghosty[i] + 12)
               && ingame) {
            // Establece el estado de "muriendo" para Pacman
            dying = true;
         }
      }
   }
//realiza sobreescritura de metodos para dibujar drawGhost
```

```
    private void drawGhost(Graphics2D g2d, int x, int y) {
// Dibuja la imagen del fantasma en la posición (x, y) utilizando el objeto Graphics2D
        g2d.drawImage(ghost, x, y, this);
    }

    private void movePacman() {

        int pos;
        short ch;
// Si se solicita un cambio de dirección opuesto, actualiza la dirección sin cambiar la vista
        if (reqdx == -pacmandx && reqdy == -pacmandy) {
            pacmandx = reqdx;
            pacmandy = reqdy;
            viewdx = pacmandx;
            viewdy = pacmandy;
        }
        // Verifica si Pacman está en una posición de cuadrícula
        if (pacmanx % blocksize == 0 && pacmany % blocksize == 0) {
            pos = pacmanx / blocksize + nrofblocks * (int) (pacmany / blocksize);
            ch = screendata[pos];
            // Verifica si hay una píldora en la celda actual
            if ((ch & 16) != 0) {
                screendata[pos] = (short) (ch & 15);// Elimina la píldora de la celda
                score++;// Incrementa la puntuación del jugador
            }
            // Verifica si se solicita un cambio de dirección y si es posible moverse en esa dirección
            if (reqdx != 0 || reqdy != 0) {
                if (!((reqdx == -1 && reqdy == 0 && (ch & 1) != 0)
                        || (reqdx == 1 && reqdy == 0 && (ch & 4) != 0)
                        || (reqdx == 0 && reqdy == -1 && (ch & 2) != 0)
                        || (reqdx == 0 && reqdy == 1 && (ch & 8) != 0))) {
                    pacmandx = reqdx;
                    pacmandy = reqdy;
                    viewdx = pacmandx;
                    viewdy = pacmandy;
                }
            }

            // Verifica si Pacman está en una posición donde no puede moverse
            if ((pacmandx == -1 && pacmandy == 0 && (ch & 1) != 0)
                    || (pacmandx == 1 && pacmandy == 0 && (ch & 4) != 0)
                    || (pacmandx == 0 && pacmandy == -1 && (ch & 2) != 0)
                    || (pacmandx == 0 && pacmandy == 1 && (ch & 8) != 0)) {
                pacmandx = 0;
                pacmandy = 0;
            }
```

```
        }
        // Actualiza la posición de Pacman basado en la dirección y la velocidad
        pacmanx = pacmanx + pacmanspeed * pacmandx;
        pacmany = pacmany + pacmanspeed * pacmandy;
    }
//aplico la sobreescritura del metodo drawPacman
    private void drawPacman(Graphics2D g2d) {

        if (viewdx == -1) {
            // Dibuja Pacman mirando a la izquierda
            drawPacnanLeft(g2d);
        } else if (viewdx == 1) {
            // Dibuja Pacman mirando a la derecha
            drawPacmanRight(g2d);
        } else if (viewdy == -1) {
            // Dibuja Pacman mirando hacia arriba
            drawPacmanUp(g2d);
        } else {
            // Dibuja Pacman mirando hacia abajo
            drawPacmanDown(g2d);
        }
    }
//aplico la sobreescritura del metodo drawPacman
    private void drawPacmanUp(Graphics2D g2d) {
// Seleccionar la imagen de Pacman a dibujar basada en el valor de pacmananimpos
        switch (pacmananimpos) {
            case 1:
                // Dibujar la segunda imagen de Pacman mirando hacia arriba
                g2d.drawImage(pacman2up, pacmanx + 1, pacmany + 1, this);

                break;
            case 2:
                // Dibujar la tercera imagen de Pacman mirando hacia arriba
                g2d.drawImage(pacman3up, pacmanx + 1, pacmany + 1, this);
                break;
            case 3:
                // Dibujar la cuarta imagen de Pacman mirando hacia arriba
                g2d.drawImage(pacman4up, pacmanx + 1, pacmany + 1, this);
                break;
            default:
                //Si no coincide con ninguna de las opciones anteriores, dibujar la imagen
predeterminada de Pacman
                g2d.drawImage(pacman1, pacmanx + 1, pacmany + 1, this);
                break;
        }
    }
```

```
//aplico la sobreescritura del metodo drawPacman
   private void drawPacmanDown(Graphics2D g2d) {

      switch (pacmananimpos) {
         case 1:
            g2d.drawImage(pacman2down, pacmanx + 1, pacmany + 1, this);
            break;
         case 2:
            g2d.drawImage(pacman3down, pacmanx + 1, pacmany + 1, this);
            break;
         case 3:
            g2d.drawImage(pacman4down, pacmanx + 1, pacmany + 1, this);
            break;
         default:
            g2d.drawImage(pacman1, pacmanx + 1, pacmany + 1, this);
            break;
      }
   }

   private void drawPacnanLeft(Graphics2D g2d) {
// Seleccionar la imagen de Pacman a dibujar basada en el valor de pacmananimpos
      switch (pacmananimpos) {
         case 1:
            // Dibujar la segunda imagen de Pacman mirando hacia abajo
            g2d.drawImage(pacman2left, pacmanx + 1, pacmany + 1, this);
            break;
         case 2:
            // Dibujar la tercera imagen de Pacman mirando hacia abajo
            g2d.drawImage(pacman3left, pacmanx + 1, pacmany + 1, this);
            break;
         case 3:
            // Dibujar la cuarta imagen de Pacman mirando hacia abajo
            g2d.drawImage(pacman4left, pacmanx + 1, pacmany + 1, this);
            break;
         default:
            // Si no coincide con ninguna de las opciones anteriores, dibujar la imagen
predeterminada de Pacman
            g2d.drawImage(pacman1, pacmanx + 1, pacmany + 1, this);
            break;
      }
   }

   private void drawPacmanRight(Graphics2D g2d) {
// Seleccionar la imagen de Pacman a dibujar basada en el valor de pacmananimpos
      switch (pacmananimpos) {
         case 1:
```

```
            // Dibujar la segunda imagen de Pacman mirando hacia la derecha
            g2d.drawImage(pacman2right, pacmanx + 1, pacmany + 1, this);
            break;
         case 2:
             // Dibujar la tercera imagen de Pacman mirando hacia la derecha
            g2d.drawImage(pacman3right, pacmanx + 1, pacmany + 1, this);
            break;
         case 3:
            // Dibujar la cuarta imagen de Pacman mirando hacia la derecha
            g2d.drawImage(pacman4right, pacmanx + 1, pacmany + 1, this);
            break;
         default:
            // Si no coincide con ninguna de las opciones anteriores, dibujar la imagen
predeterminada de Pacman
            g2d.drawImage(pacman1, pacmanx + 1, pacmany + 1, this);
            break;
      }
   }
//se realizo una sobreescritura de Metodos para dibujar drawMaze
   private void drawMaze(Graphics2D g2d) {

      short i = 0;
      int x, y;
      // Recorrer el arreglo screendata para dibujar el laberinto
      for (y = 0; y < scrsize; y += blocksize) {
         for (x = 0; x < scrsize; x += blocksize) {

            g2d.setColor(mazecolor);
            g2d.setStroke(new BasicStroke(2));
            // Verificar si hay una pared en la dirección izquierda (bit 1)
            if ((screendata[i] & 1) != 0) {
               // Dibujar una línea vertical izquierda
               g2d.drawLine(x, y, x, y + blocksize - 1);
            }
            // Verificar si hay una pared en la dirección arriba (bit 2)
            if ((screendata[i] & 2) != 0) {
               // Dibujar una línea horizontal superior
               g2d.drawLine(x, y, x + blocksize - 1, y);
            }
            // Verificar si hay una pared en la dirección derecha (bit 3)
            if ((screendata[i] & 4) != 0) {
               g2d.drawLine(x + blocksize - 1, y, x + blocksize - 1,
                     y + blocksize - 1); // Dibujar una línea vertical derecha
            }
            // Verificar si hay una pared en la dirección abajo (bit 4)
            if ((screendata[i] & 8) != 0) {
```

```
                g2d.drawLine(x, y + blocksize - 1, x + blocksize - 1,
                        y + blocksize - 1);// Dibujar una línea horizontal inferior
            }
            // Verificar si hay un punto en la celda (bit 5)
            if ((screendata[i] & 16) != 0) {
                g2d.setColor(dotcolor);
                // Dibujar un punto en la celda
                g2d.fillRect(x + 11, y + 11, 2, 2);
            }

            i++;
        }
    }
}

private void initGame() {

pacsleft = 3;         // Inicializar la cantidad de vidas del jugador
score = 0;            // Inicializar la puntuación del jugador
initLevel();          // Inicializar el nivel de juego
nrofghosts = 6;       // Establecer el número de fantasmas
currentspeed = 3;     // Establecer la velocidad inicial del juego
}

private void initLevel() {
    // Copiar los datos del nivel actual desde leveldata a screendata
    int i;
    for (i = 0; i < nrofblocks * nrofblocks; i++) {
        screendata[i] = leveldata[i];
    }
    // Continuar con la inicialización del nivel
    continueLevel();
}

private void continueLevel() {
short i;
int dx = 1; // Dirección inicial de los fantasmas
int random; // Variable para generar un número aleatorio

// Configuración inicial de los fantasmas
for (i = 0; i < nrofghosts; i++) {
    ghosty[i] = 4 * blocksize; // Posición vertical de los fantasmas
    ghostx[i] = 4 * blocksize; // Posición horizontal de los fantasmas
    ghostdy[i] = 0; // Dirección vertical de los fantasmas
    ghostdx[i] = dx; // Dirección horizontal de los fantasmas
    dx = -dx; // Invertir la dirección horizontal para el siguiente fantasma
```

```
        random = (int) (Math.random() * (currentspeed + 1)); // Generar un número aleatorio para
la velocidad

        if (random > currentspeed) {
            random = currentspeed; // Limitar la velocidad aleatoria a la velocidad actual
        }

        ghostspeed[i] = validspeeds[random]; // Asignar la velocidad aleatoria a los fantasmas
    }

    pacmanx = 7 * blocksize; // Posición horizontal inicial de Pacman
    pacmany = 11 * blocksize; // Posición vertical inicial de Pacman
    pacmandx = 0; // Dirección horizontal de Pacman
    pacmandy = 0; // Dirección vertical de Pacman
    reqdx = 0; // Dirección horizontal deseada de Pacman
    reqdy = 0; // Dirección vertical deseada de Pacman
    viewdx = -1; // Dirección horizontal de la vista del jugador
    viewdy = 0; // Dirección vertical de la vista del jugador
    dying = false; // Bandera que indica si Pacman está muriendo
}


    private void loadImages() {
        // Cargar la imagen del fantasma
        ghost = new ImageIcon(getClass().getResource("../imagenes/ghost.gif")).getImage();
        // Cargar las imágenes de las animaciones de Pacman en diferentes direcciones y estados
        pacman1 = new
ImageIcon(getClass().getResource("../imagenes/pacman1.gif")).getImage();
        pacman2up = new
ImageIcon(getClass().getResource("../imagenes/pacman2up.gif")).getImage();
        pacman3up = new
ImageIcon(getClass().getResource("../imagenes/pacman3up.gif")).getImage();
        pacman4up = new
ImageIcon(getClass().getResource("../imagenes/pacman4up.gif")).getImage();
        pacman2down = new
ImageIcon(getClass().getResource("../imagenes/pacman2down.gif")).getImage();
        pacman3down = new
ImageIcon(getClass().getResource("../imagenes/pacman3down.gif")).getImage();
        pacman4down = new
ImageIcon(getClass().getResource("../imagenes/pacman4down.gif")).getImage();
        pacman2left = new
ImageIcon(getClass().getResource("../imagenes/pacman2left.gif")).getImage();
        pacman3left = new
ImageIcon(getClass().getResource("../imagenes/pacman3left.gif")).getImage();
        pacman4left = new
ImageIcon(getClass().getResource("../imagenes/pacman4left.gif")).getImage();
```

```
        pacman2right = new
ImageIcon(getClass().getResource("../imagenes/pacman2right.gif")).getImage();
        pacman3right = new
ImageIcon(getClass().getResource("../imagenes/pacman3right.gif")).getImage();
        pacman4right = new
ImageIcon(getClass().getResource("../imagenes/pacman4right.gif")).getImage();

    }
//sobre escritura
   @Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);

    // Llamar al método para realizar el dibujo de los componentes gráficos
    doDrawing(g);
}

// Método principal para realizar el dibujo de los componentes gráficos
//Estos métodos se sobrescriben en la clase tablero para proporcionar la lógica específica de
dibujo.
private void doDrawing(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;

    // Establecer el color de fondo y llenar el panel
    g2d.setColor(Color.black);
    g2d.fillRect(0, 0, d.width, d.height);

    // Dibujar el laberinto
    drawMaze(g2d);

    // Dibujar la puntuación del juego
    drawScore(g2d);

    // Realizar animaciones
    doAnim();

    // Si el juego está en marcha, mostrar el juego actual
    if (ingame) {
        playGame(g2d);
    } else {
        // Mostrar la pantalla de introducción
        showIntroScreen(g2d);
    }

    // Dibujar la imagen en el panel
    g2d.drawImage(ii, 5, 5, this);
```

```
    // Sincronizar la imagen con el hardware gráfico
    Toolkit.getDefaultToolkit().sync();

    // Liberar los recursos de gráficos
    g2d.dispose();
}

class TAdapter extends KeyAdapter {

    @Override
    public void keyPressed(KeyEvent e) {
        int key = e.getKeyCode();

        if (ingame) {
            // Manejar las teclas presionadas durante el juego
            if (key == KeyEvent.VK_LEFT) {
                reqdx = -1;
                reqdy = 0;
            } else if (key == KeyEvent.VK_RIGHT) {
                reqdx = 1;
                reqdy = 0;
            } else if (key == KeyEvent.VK_UP) {
                reqdx = 0;
                reqdy = -1;
            } else if (key == KeyEvent.VK_DOWN) {
                reqdx = 0;
                reqdy = 1;
            } else if (key == KeyEvent.VK_ESCAPE && timer.isRunning()) {
                ingame = false;
            } else if (key == KeyEvent.VK_PAUSE) {
                if (timer.isRunning()) {
                    timer.stop();
                } else {
                    timer.start();
                }
            }
        } else {
            // Iniciar el juego desde la pantalla de introducción al presionar 's' o 'S'
            if (key == 's' || key == 'S') {
                ingame = true;
                initGame();
            }
        }
    }
```

```
    @Override
    public void keyReleased(KeyEvent e) {
        int key = e.getKeyCode();

        // Restablecer la dirección cuando se suelta una tecla de dirección
        if (key == KeyEvent.VK_LEFT || key == KeyEvent.VK_RIGHT
                || key == KeyEvent.VK_UP || key == KeyEvent.VK_DOWN) {
            reqdx = 0;
            reqdy = 0;
        }
    }
}


@Override
public void actionPerformed(ActionEvent e) {
    // Este método se llama automáticamente en intervalos regulares
    // Repintar el componente para actualizar la visualización del juego
    repaint();
}
}
```

Clase Pacman

```
public class Pacman extends JFrame {
    public Pacman() {
        // Inicializa la interfaz gráfica del juego
        initUI();
    }
    private void initUI() {
        // Agrega un tablero (instancia de la clase 'tablero') a la ventana
        add(new tablero());
        // Configura las propiedades de la ventana
        setTitle("Juego de Pac-Man"); // Título de la ventana
        setDefaultCloseOperation(EXIT_ON_CLOSE);     // Termina la aplicación al cerrar la
ventana
        setSize(380, 420);                    // Tamaño de la ventana
        setLocationRelativeTo(null);              // Centra la ventana en la pantalla
        setVisible(true);                     // Hace la ventana visible
    }
    public static void main(String[] args) {
        // Ejecuta la creación y visualización de la ventana en el hilo de eventos
        EventQueue.invokeLater(() -> {
            Pacman ex = new Pacman(); // Crea una instancia de la clase 'Pacman'
```

```
            ex.setVisible(true);      // Hace la ventana visible
        });
    }
}
```

3.5.2. Ejecución de la Cuenta

Imagen 6 Ejecución juego de Pac-man por Eduardo Ordoñez

4. Conclusiones

- El código muestra un enfoque de Programación orientado a objetos al modelar el juego Pacman. Utiliza clases y objetos para representar diferentes componentes del juego, como el tablero, Pacman y los fantasmas. Esto ayuda a organizar y abstraer la lógica del juego de manera más clara y modular.
- El código maneja gráficos y animaciones de manera efectiva utilizando la clase Graphics2D para dibujar elementos en el lienzo del juego. Implementa una lógica de

animación para cambiar las imágenes de Pacman y crea una sensación visual agradable en el juego.

5. Recomendaciones

- El programa debe estar bien comentado para comprender y entender cada funcionamiento del código ya que nos ayuda mucho al entendimiento del juego.
- Familiarízate con patrones específicos de dominio para abordar problemas particulares.

6. Bibliografía

Cabrera, I. (2022, January 12). *Todo lo que necesitas saber sobre el diagrama de caso de uso*. Venngage. Retrieved May 30, 2023, from https://es.venngage.com/blog/diagrama-de-caso-de-uso/

Lara, D. (2015, July 7). *Encapsulamiento en la programación orientada a objetos*. Styde.net. Retrieved May 30, 2023, from https://styde.net/encapsulamiento-en-la-programacion-orientada-a-objetos/

Mancuzo, G. (2021, June 24). *Qué son los Diagramas de UML? + Tipos + Importancia*. Blog – ComparaSoftware. Retrieved May 30, 2023, from https://blog.comparasoftware.com/diagramas-de-uml-que-significa-esta-metodologia/

Martínez, M. (2020, November 2). *¿Qué es la Programación Orientada a Objetos?* Profile. Retrieved May 30, 2023, from https://profile.es/blog/que-es-la-programacion-orientada-a-objetos/

Paredes, B. (2022, January 26). *¿Qué es código limpio?* LinkedIn. Retrieved May 30, 2023, from https://es.linkedin.com/pulse/qu%C3%A9-es-c%C3%B3digo-limpio-b-parde

Burbeck, S. (2023, January 6). *Programación de Aplicaciones*. Model-View-Controller. Retrieved June 24, 2023, from http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

Diácono, J. (2023, January 6). *Arquitectura Modelo-Vista-Controlador (MVC)*. Retrieved June 24, 2023, from http://www.jdl.co.uk/briefings/mvc.pdf

Hernandez, R. D. (2021, June 28). *El patrón modelo-vista-controlador: Arquitectura y frameworks explicados*. freeCodeCamp. Retrieved June 24, 2023, from https://www.freecodecamp.org/espanol/news/el-modelo-de-arquitectura-view-controller-pattern/

Pantoja, B. (2004). *El patrón de diseño Modelo-Vista-Controlador (MVC) y su implementación en Java Swing*. SciELO Bolivia. Retrieved June 24, 2023, from http://www.scielo.org.bo/scielo.php?pid=S1683-07892004000100005&script=sci_arttext

jose. (2023, Febrero 5). *¿Qué es la programación modular? - Conoce todo sobre la descomposición de un programa en trozos más pequeños | Qué es*. Qué es - Web de referencia en Significados, Definiciones, Conceptos, Ejemplos y ¿Qué es? Retrieved August 18, 2023, from https://quees.com/programacion-modular/#%C2%BFQu%C3%A9_es_la_programaci%C3%B3n_modular_en_JavaScript?

Lara, D. (2015, July 7). *Modularidad en la programación orientada a objetos*. Styde.net. Retrieved August 18, 2023, from https://styde.net/modularidad-en-la-programacion-orientada-a-objetos/

Martínez, M. (2020, June 24). *Qué son los Patrones de Diseño de software / Design Patterns*. Profile. Retrieved August 18, 2023, from https://profile.es/blog/patrones-de-diseno-de-software/#%C2%BFQue_son_los_patrones_de_diseno_design_patterns

Soto, N. (2021, July 2). *¿Qué son los patrones de diseño en programación? 2023*. Craft - Code. Retrieved August 18, 2023, from https://craft-code.com/que-son-los-patrones-de-diseno/