Proyecto Final. Primera entrega. Dominó

Integrantes
Acosta Porcayo Alan Omar 320206102
Gutiérrez Grimaldo Alejandro 320282098
Medina Villa Samuel 320249538

Índice

1. Antecedentes históricos

El origen de este juego no es tan claro, ya que algunos historiadores y antropólogos fueron los griegos, también otros afirman que fueron los hebreos. Sin embargo, el juego actual del dominó parece que se inició en China, donde se jugaba hace 1.500 años de manera semejante a como se hace hoy. Se podría considerar como el primer dominó de la historia.

La forma actual conocida en Europa y el mundo no aparece hasta el siglo XVIII, cuando lo introdujeron los italianos en este continente. Gracias a la enorme expansión de la cultura europea a través del mundo, llegó a diversos países y culturas.

Inicialmente, las fichas se fabricaban mediante la unión de dos láminas de ébano en ambos lados de la ficha de hueso. Este método tenía la ventaja de prevenir posibles trampas al ocultar los puntos en la parte posterior de la ficha bajo ciertas condiciones de iluminación, además de crear un atractivo contraste entre los puntos blancos y el fondo negro, lo que permitía que el hueso fuera visible a través de los agujeros en el ébano.

2. Descripción

El objetivo principal del juego de dominó es ser el primer jugador en quedarse sin fichas en su mano. Para lograrlo, los jugadores deben emparejar las fichas que poseen con las que ya están en la mesa, asegurándose de que los números de los puntos coincidan. Esto implica tomar decisiones estratégicas sobre cuándo y cómo jugar sus fichas para maximizar sus posibilidades de quedarse sin fichas antes que sus oponentes.

3. Definición de clases

3.1. Clase *Ficha*

Esta clase representa una ficha de dominó simple.

3.1.1. Atributos

1. *caraIzq.* Variable privada de tipo *int* con el valor de la parte izquierda de la ficha.
2. *caraDer.* Variable privada de tipo *int* con el valor de la parte derecha de la ficha.
3. *suma.* Variable privada de tipo *int* con la suma de ambas partes de la ficha, se calcula al momento de crear la ficha.
4. *mula.* Variable privada de tipo *boolean* que indica si la ficha es una mula o no.

3.1.2. Métodos

1. *Ficha(int caraIzq, int caraDer).* Método constructor que recibe dos enteros que representan los valores de las caras de la ficha. Calcula la suma y determina si la ficha es una mula.
2. *getCaraIzq().* Método público que retorna el valor de la cara izquierda de la ficha.

3. *getCaraDer().* Método público que retorna el valor de la cara derecha de la ficha.
4. *getSuma().* Método público que retorna el valor de la suma de la ficha.
5. *esMula().* Método público que retorna un booleano que indica si la ficha es una mula o no.
6. *girar().* Método público que intercambia los valores de las caras de la ficha.
7. *toString().* Método público (sobreescritura) que retorna una cadena con formato del valor de las caras de la ficha.

3.2. Clase Abstracta *Jugador*

Esta clase representa a un jugador de dominó, ya sea persona o bot.

3.2.1. Atributos

1. *nombre.* Variable protegida de tipo *String* con el nombre del jugador.
2. *fichas.* Variable protegida de tipo *ArrayList<Ficha>* con las fichas del jugador.
3. *puedeJugar.* Variable protegida de tipo *boolean* que indica si el jugador tiene una ficha que pueda jugar.

3.2.2. Métodos

1. *getNombre().* Método público que retorna el nombre del jugador.
2. *getFichas().* Método público que retorna las fichas del jugador.
3. *getFicha(int i).* Método público que retorna la ficha en la posición i del jugador.
4. *agregarFicha(Ficha ficha).* Método público que agrega una ficha a las fichas del jugador.
5. *puedeJugar().* Método público que retorna un booleano que indica si el jugador puede jugar.
6. *puedeJugar(ArrayList<Ficha> mesa).* Método público (sobrecarga) que comprueba si dentro de las fichas del jugador hay alguna que pueda jugarse en la mesa actual y actualiza el valor de *puedeJugar*.
7. *robar(ArrayList<Ficha> pozo).* Método público que sustrae una ficha del pozo y la agrega a las fichas del jugador.
8. *primerTurno(ArrayList<Ficha> mesa).* Método público que realiza el primer turno del jugador.
9. *turno(ArrayList<Ficha> mesa).* Método final y público que realiza el turno del jugador. Se divide en pasos que son métodos abstractos y concretos que son implementados en las subclases *Persona* y *Bot*.
10. *buscarFicha(ArrayList¡Ficha¿mesa).* Método abstracto que devuelve la instrucción de la ficha y su orientación en la mesa.
11. *validarFicha(ArrayList¡Ficha¿mesa, int indice, char orientacion).* Método privado que es un paso del método *turno* que valida si la ficha que se quiere jugar es válida.

12. *jugarFicha(ArrayList¡Ficha¿mesa, int indice, char orientacion).* Método privado que es un paso del método *turno* que juega la ficha en la mesa.

3.3. Clase *Persona*

Esta clase representa a un jugador controlado por el usuario.

3.3.1. Métodos

1. *Humano(String nombre).* Método constructor que recibe el nombre del jugador.
2. *buscarFicha(ArrayList¡Ficha¿mesaActual).* Método protegido (sobreescritura) que le pide al usuario que ingrese el índice de la ficha que quiere jugar y la orientación en la que la quiere jugar.

3.4. Clase *Bot*

Esta clase representa a un jugador controlado por la computadora.

3.4.1. Métodos

1. *Bot(String nombre).* Método constructor que recibe el nombre del jugador.
2. *turno(ArrayList<Ficha> mesaActual).* Método protected (sobreescritura) que busca una ficha para jugar en la mesa actual.
3. *evaluarFicha(Ficha ficha, ArrayList¡Ficha¿mesa).* Método privado que calcula las posibilidades de una de las fichas del bot.
4. *contarPosibilidades(Ficha ficha, ArrayList¡Ficha¿mesa).* Método privado que cuenta las posibilidades de una de las fichas del bot.
5. *elegirFichaGenerica(ArrayList¡Ficha¿mesa).* Método privado que solo se utiliza en caso de que los métodos *evaluarFicha* y *contarPosibilidades* no encuentren una ficha para jugar.

3.5. Clase *Modelo*

Esta clase representa la mesa de juego.

3.5.1. Atributos

1. *mesa.* Variable privada de tipo *ArrayList<Ficha>* con las fichas de la mesa.
2. *pozo.* Variable privada de tipo *ArrayList<Ficha>* con las fichas sobrantes del juego.
3. *jugadores.* Variable privada de tipo *List<Jugador>* con los jugadores del juego. Inicialmente contiene un objeto de tipo *Humano* y dos de tipo *Bot*.
4. *turno.* Variable privada de tipo *int* que lleva la cuenta de los turnos.

3.5.2. Métodos

1. *Modelo(String nombre).* Primer método constructor (sobrecarga) que recibe el nombre del jugador y crea un objeto de tipo *Persona* con ese nombre y un objeto de tipo *Bot*, reparte las fichas y decide quien empieza el juego.
2. *Modelo().* Segundo método constructor (sobrecarga) que crea dos objetos de tipo *Bot*, reparte las fichas y decide quien empieza el juego.
3. *Modelo(Configuracion conf).* Tercer método constructor (sobrecarga) que recibe un objeto de tipo *Configuracion* y copia los jugadores y el pozo de la configuración.
4. *getMesa().* Método público que retorna las fichas de la mesa.
5. *getPozo().* Método público que retorna las fichas del pozo.
6. *getTamanioPozo().* Método publico que retorna el tamaño del pozo.
7. *getJugadores().* Método público que retorna los jugadores del juego.
8. *getJugador(int i)* Método público que retorna un objeto de tipo *Jugador* en la posición i de la lista de jugadores.
9. *getTurno().* Método público que retorna el turno actual.
10. *repartir().* Método privado que crea los objetos de tipo *Ficha*, los agrega al pozo, los mezcla y los reparte entre los jugadores.
11. *decidirPrimerTurno().* Método privado que determina quien empieza el juego buscando quien tiene la mula mas alta y si no hay mula, quien tiene la ficha con la suma mas alta. En caso de que el primer turno sea del jugador en la posición 1 de la lista de jugadores se cambia el orden de los jugadores.
12. *cambioDeTurno().* Método público que cambia el orden de los jugadores en la lista.

3.6. Clase *Historial*

Esta clase permite guardar y cargar un historial de partidas en un archivo de texto.

3.6.1. Métodos

1. *getHistorial().* Método público y estático que retorna un objeto de tipo *ArrayList<Partida>* con las partidas guardadas en el archivo de texto.
2. *guardarResultado(String nombre, int turno).* Método público y estático que recibe el nombre del ganador y el número de turnos que duró la partida y lo guarda en el archivo de texto junto con la fecha actual.

3.7. Clase *Partida*

Esta clase representa una partida guardada en el historial.

3.7.1. Atributos

1. *nombre.* Variable privada de tipo *String* con el nombre del ganador.
2. *turno.* Variable privada de tipo *int* con el número de turnos que duró la partida.
3. *fecha.* Variable privada de tipo *String* con la fecha en la que se jugó la partida.

3.7.2. Métodos

1. *getNombre().* Método público que retorna el nombre del ganador.
2. *getTurno().* Método público que retorna el número de turnos que duró la partida.
3. *getFecha().* Método público que retorna la fecha en la que se jugó la partida.

3.8. Clase *Configuracion*

Esta clase permite guardar y cargar una configuración de una partida en un archivo de texto debido a la serialización y deserialización.

3.8.1. Atributos

1. *pozo.* Variable privada de tipo *ArrayList<Ficha>* con las fichas del pozo.
2. *jugadores.* Variable privada de tipo *ArrayList<Jugador>* con los jugadores de la partida.
3. *listaArchivos.* Variable privada, estática y trasndental de tipo *ArrayList<String>* con los nombres de los archivos de texto en la carpeta *configuraciones*.

3.8.2. Métodos

1. *Configuracion(String nombreArchivo).* Primer método constructor (sobrecarga) que recibe el nombre del archivo de texto y carga la configuración de la partida guardada en el archivo.
2. *Configuracion(ArrayList¡Ficha¿pozo, ArrayList¡Jugador¿jugadores).* Segundo método constructor (sobrecarga) privado que recibe el pozo y los jugadores de una partida.
3. *getPozo().* Método público que retorna las fichas del pozo.
4. *getJugadores().* Método público que retorna los jugadores de la partida.
5. *getListaArchivos().* Método público y estático que retorna un objeto de tipo *ArrayList<String>* con los nombres de los archivos de texto en la carpeta *configuraciones*.
6. *guardarConfiguracion(String nombreArchivo, ArrayList¡Ficha¿pozo, ArrayList¡Jugador¿jugadores).* Método público y estático que recibe el nombre del archivo de texto, el pozo y los jugadores de una partida y guarda la configuración en el archivo utilizando el método constructor privado.

3.9. Clase *Musica*

Esta clase permite reproducir clips de audio.

3.9.1. Atributos

1. *musica.* Variable privada y estática de tipo *Clip* que representa el clip de audio.

3.9.2. Métodos

1. *reiniciarMusica()* Método público que reinicia el clip de audio.
2. *musicaFinal().* Método público que reproduce el clip de audio de victoria.

3.10. Clase *Controlador*

Esta clase permite controlar el flujo del programa.

3.10.1. Atributos

1. *sc.* Variable privada y estática de tipo *Scanner* que permite leer las entradas del usuario.
2. *modelo.* Variable privada y estática de tipo *Modelo* que representa la mesa de juego.
3. *musica.* Variable privada y estática de tipo *Musica* que permite reproducir clips de audio.
4. *ESPERA.* Variable privada, estática y final de tipo *int* que representa el tiempo de espera entre turnos.
5. *opcion.* Variable privada y estática de tipo *int* que representa la opción seleccionada por el usuario en el menú.

3.10.2. Métodos

1. *main(String[] args).* Método público y estático que inicia el programa y controla el flujo del mismo.
2. *esperar(int tiempo).* Método privado y estático que recibe un entero que representa el tiempo de espera en milisegundos.
3. *guardarConfiguracion().* Método privado y estático que guarda le pregunta al usuario si desea guardar la configuración de la partida actual y la guarda en caso de que la respuesta sea afirmativa.
4. *continuarJuego().* Método privado y estático que evalua las condiciones para continuar el juego de acuerdo a las fichas de los jugadores y el pozo.
5. *decidirGanador().* Método privado y estático que determina quien es el ganador de la partida.

3.11. Clase interna *HiloTurno*

Esta clase permite realizar el turno de un jugador en un hilo diferente al principal.

3.11.1. Atributos

1. *jugador.* Variable privada de tipo *Jugador* que representa al jugador que va a realizar su turno.
2. *mesa.* Variable privada de tipo *ArrayList<Ficha>* que representa las fichas de la mesa.

3. *turno.* Variable privada de tipo *int* que representa el turno actual.
4. *tamanioPozo.* Variable privada de tipo *int* que representa el tamaño del pozo.

3.11.2. Métodos

1. *HiloTurno(Jugador jugador, ArrayList¡Ficha¿mesa, int turno, int tamanioPozo).* Método constructor que recibe el jugador que va a realizar su turno, las fichas de la mesa, el turno actual y el tamaño del pozo.
2. *run().* Método público (sobreescritura) que realiza el turno del jugador.

3.12. Clase *Vista*

Esta clase permite imprimir en la terminal.

3.12.1. Métodos

1. *limpiarPantalla().* Método público y estático que limpia la terminal.
2. *continuar().* Método público y estático que le pide al usuario que presione enter para continuar.
3. *pedirNombre().* Método público y estático que le pide al usuario que ingrese su nombre.
4. *mostrarMenu().* Método público y estático que imprime el menú de opciones.
5. *mostrarPozo(int pozo).* Método público y estático que imprime el número de fichas en el pozo.
6. *mostrarTurno(String nombre).* Método público y estático que imprime el nombre del jugador que tiene el turno.
7. *mostrarMesa(ArrayList¡Ficha¿mesa).* Método público y estático que imprime las fichas de la mesa.
8. *mostrarFichas(ArrayList¡Ficha¿fichas).* Método público y estático que imprime las fichas de un jugador.
9. *mostrarGanador(String nombre).* Método público y estático que imprime el nombre del ganador.
10. *mostrarEmpate().* Método público y estático que imprime un mensaje de empate.
11. *mostrarHistorial(ArrayList¡Partida¿historial).* Método público y estático que imprime el historial de partidas.
12. *mostrarArchivos(ArrayList¡String¿archivos).* Método público y estático que imprime los nombres de los archivos de texto en la carpeta *configuraciones*.

4. Definición de objetos

- *ArrayList<Ficha>.* Colección de objetos de tipo *Ficha*. Se crea para representar las fichas de la mesa, las fichas del pozo y las fichas de los jugadores.
- *ArrayList<Jugador>.* Colección de objetos de tipo *Jugador*. Se crea para representar los jugadores de la partida.

- *ArrayList<Partida>*. Colección de objetos de tipo *Partida*. Se crea para representar las partidas guardadas en el historial.
- *ArrayList<String>*. Colección de objetos de tipo *String*. Se crea para representar los nombres de los archivos de texto en la carpeta *configuraciones*.
- *Scanner*. Objeto que permite leer las entradas del usuario.
- *Clip*. Objeto que permite reproducir clips de audio.
- *ProcessBuilder*. Objeto que permite ejecutar comandos en la terminal.
- *Thread*. Objeto que permite crear hilos.
- *ObjectInputStream*. Objeto que permite deserializar objetos.
- *ObjectOutputStream*. Objeto que permite serializar objetos.
- *File*. Objeto que permite crear archivos.
- *FileReader*. Objeto que permite leer archivos.
- *BufferedReader*. Objeto que permite leer archivos.
- *FileWriter*. Objeto que permite escribir archivos.
- *BufferedWriter*. Objeto que permite escribir archivos.
- *Date*. Objeto que permite obtener la fecha actual.

5. Algoritmo

Se comienza creando un objeto *Scanner* llamado *sc* para leer las entradas del usuario y un objeto *Clip* llamado *musica* para reproducir los clips de audio. Para reproducir un clip se hace dentro de un bloque *try-catch* para evitar que el programa se detenga en caso de que no se encuentre el archivo de audio.

```
Scanner sc = new Scanner(System.in);
Clip musica = null;

try {
  musica = AudioSystem.getClip();
  musica.open(AudioSystem.getAudioInputStream(Domino.class.getResource("/recursos/MoonlightSonata.wav")));
  musica.loop(Clip.LOOP_CONTINUOUSLY);
} catch (Exception e) {
  e.printStackTrace();
}
```

Luego, se imprime el menú con las opciones de juego para que el usuario pueda seleccionar jugador vs bot o bot vs bot. Se lee la entrada del usuario y se crea un objeto de tipo *Mesa* llamado *mesa* con el tipo de juego seleccionado por el usuario.

```
System.out.println("%%%%%%%%%%%%%%% Bienvenido a Domino %%%%%%%%%%%%%%%");
System.out.println("Escoga el modo de juego:");
System.out.println("1. Jugador vs Bot");
System.out.println("2. Bot vs Bot");
System.out.print("$ ");
int tipoJuego = sc.nextInt();
Mesa mesa = new Mesa(tipoJuego);
```

Dentro de un ciclo *while* con la condición de repetición de que los jugadores tengan fichas y el pozo no este vacío, se limpia la terminal usando una un bloque *try-catch* y un objeto *ProcessBuilder*.

```
try {
  new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();
} catch (Exception e) {
  e.printStackTrace();
}
```

Posteriormente, se imprime las fichas de la mesa, el número de fichas en el pozo y se anuncia el turno del jugador en la posición 0 de la lista de jugadores e imprime sus fichas.

```
System.out.println("\n\nMesa actual:");
mesa.imprimir();
System.out.println("\nNumero de fichas en el Pozo: " + mesa.getPozo().size());

System.out.println("\n%%%%% Turno de " + mesa.getJugadores().get(0).getNombre() + " %%%%%");
mesa.imprimir(mesa.getJugadores().get(0).getFichas());
```

El siguiente bloque de código corresponde al turno de los jugadores. Si la mesa esta vacia quiere decir que es el primer turno, en ese caso se llama al método *primerTurno* y se continua con el flujo del programa. Para los turnos siguientes primero se comprueba si el jugador puede jugar una ficha realiza su turno con el método *turno*. En caso de que no pueda jugar y no haya fichas en el pozo se imprime un mensaje diciendo que el jugador con el turno actual pasa de turno, y si hay fichas en el pozo roba una, se vuelve a imprimir sus fichas y se pasa de turno.

```
if (mesa.getMesa().isEmpty())
  mesa.getJugadores().get(0).primerTurno(mesa.getMesa());
else {
  if (mesa.getJugadores().get(0).puedeJugar(mesa.getMesa()))
    mesa.getJugadores().get(0).turno(mesa.getMesa());
  else {
    if (mesa.getPozo().isEmpty()) {
      System.out.println("No puedes jugar y no hay fichas en el pozo, pasas tu turno");

      try {
        Thread.sleep(3000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    } else {
      if (mesa.getJugadores().get(0) instanceof Humano)
        System.out.println("Necesitas robar una ficha");

      try {
        Thread.sleep(3000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }

      System.out.println(mesa.getJugadores().get(0).getNombre() + " roba 1 ficha y pasa su turno");
      mesa.getJugadores().get(0).robar(mesa.getPozo());
      mesa.imprimir(mesa.getJugadores().get(0).getFichas());

      try {
        Thread.sleep(3000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Al salir del ciclo *while*, dentro de un bloque *try-catch* se para la reproducción del clip de audio y se inicia otro clip de audio con la música de victoria.

```
try {
  if (musica != null)
    musica.stop();

  musica = AudioSystem.getClip();
  musica.open(AudioSystem.getAudioInputStream(Domino.class.getResource("/recursos/Victoria.wav")));
  musica.start();
} catch (Exception e) {
  e.printStackTrace();
}
```

Para terminar se decide el ganador. Si un jugador se queda sin fichas se imprime un mensaje diciendo que gano pero si el pozo se vacía se busca quien tiene la suma más alta y se imprime un mensaje diciendo que gano o si hay empate.

```
if (!mesa.getJugadores().get(0).puedeJugar() &&
      !mesa.getJugadores().get(1).puedeJugar() &&
      mesa.getPozo().isEmpty() ) {
  int suma1 = 0, suma2 = 0;
  System.out.println("Se acabaron las fichas");

  for (int i = 0; i < mesa.getJugadores().get(0).getCantidadFichas(); i++)
    suma1 += mesa.getJugadores().get(0).getFicha(i).getSuma();
  for (int i = 0; i < mesa.getJugadores().get(1).getCantidadFichas(); i++)
    suma2 += mesa.getJugadores().get(1).getFicha(i).getSuma();

  if (suma1 < suma2)
    System.out.println("El ganador es " + mesa.getJugadores().get(0).getNombre());
  else if (suma1 > suma2)
    System.out.println("El ganador es " + mesa.getJugadores().get(1).getNombre());
  else
    System.out.println("Empate");
} else
  System.out.println("\n%%%%%%%%%% El ganador es " + mesa.getJugadores().get(0).getNombre() + " %%%%%%%%%%");
```

6. Herencia de clases

La herencia de clases se utilizó principalmente en las clases *Jugador*, *Persona* y *Bot*. La clase *Jugador* es una clase abstracta que tiene los atributos y métodos que comparten los jugadores de la partida. La clase *Persona* hereda de la clase *Jugador* y tiene los métodos que permiten al usuario jugar su turno. La clase *Bot* hereda de la clase *Jugador* y tiene los métodos que permiten al bot jugar su turno.

Otra herencia de clases se encuentra en la clase interna *HiloTurno* que extiende de la clase *Thread* y se sobreescribe el método *run* para que el jugador realice su turno.

7. Implementación de interfaces

La implementación de interfaces se utilizó en la clases *Configuracion*, *Ficha* y *Jugador* para la serialización y deserialización de objetos. La clase *Configuracion* implementa la interfaz *Serializable* para poder serializar los *ArrayList* de jugadores y fichas del pozo de una partida en un archivo txt y poder cargarla posteriormente si el usuario lo desea.

8. Manejo de excepciones

Para el manejo de excepciones se utilizó principalmente los bloques *try-catch* para evitar que el programa se detenga en caso de que ocurra un error como en el ejemplo siguiente que involucra el uso de hilos:

```
 try {
  Thread.sleep(tiempo);
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

En otras ocasiones se utilizó el bloque *try-catch-with-resources* para cerrar los recursos utilizados en el programa como en el ejemplo siguiente:

```
try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("configuraciones/" + nombreArchivo))) {
  Configuracion conf = (Configuracion) ois.readObject();
  this.pozo = conf.getPozo();
  this.jugadores = conf.getJugadores();
} catch (Exception e) {
  System.out.println("Error al cargar la configuracion");
}
```

Además, implementamos excepciones propias para manejar los errores de entrada del usuario como en el siguiente ejemplo:

```
try {
  System.out.println("Donde la quieres jugar? (i o d)");
  System.out.print("$ ");
  orientacion = sc.next().charAt(0);
  if (orientacion != 'i' && orientacion != 'd')
    throw new OrientacionInvalidaException("Orientacion no valida. Debe ser 'i' o 'd'.");
} catch (OrientacionInvalidaException e) {
  System.out.println(e.getMessage() + "\n");
}
```

Las excepciones propias que creamos son:

```
package domino;

/**
 * Excepcion lanzada cuando se recibe una respuesta invalida.
 */
class RespuestaInvalidaException extends Exception {
  /**
    * Constructor de la excepcion.
    * @param mensaje Mensaje que describe la naturaleza de la respuesta invalida.
    */
  public RespuestaInvalidaException(String mensaje) {
    super(mensaje);
  }
}

/**
 * Excepcion lanzada cuando se recibe un indice de ficha invalido.
 */
class IndiceFichaInvalidoException extends Exception {
  /**
    * Constructor de la excepcion.
    * @param mensaje Mensaje que describe la naturaleza del indice de ficha invalido.
    */
  public IndiceFichaInvalidoException(String mensaje) {
    super(mensaje);
  }
}
```

```
/**
 * Excepcion lanzada cuando se recibe una orientacion invalida.
 */
class OrientacionInvalidaException extends Exception {
  /**
    * Constructor de la excepcion.
    * @param mensaje Mensaje que describe la naturaleza de la orientacion invalida.
    */
  public OrientacionInvalidaException(String mensaje) {
    super(mensaje);
  }
}
```

9. Manejo de archivos y serialización

En cuanto a manejo de archivos se utilizó la lectura/escritura de un historial de partidas en un archivo de texto con la clase Historial. Las entradas del archivo se guardan con ayuda de una instancia de la clase *Partida* que tiene los atributos del nombre del ganador, el numero de turnos y la fecha de la partida.

```
Bot 1|14|Tue Dec 12 16:25:41 CST 2023
Bot 2|30|Tue Dec 12 16:52:18 CST 2023
Empate|43|Tue Dec 12 16:57:21 CST 202
Bot 2|18|Tue Dec 12 16:59:21 CST 2023
Bot 2|36|Tue Dec 12 17:00:18 CST 2023
Bot 1|30|Tue Dec 12 17:00:35 CST 2023
Bot 1|14|Tue Dec 12 17:00:56 CST 2023
Bot 2|33|Tue Dec 12 17:01:13 CST 2023
Bot 1|13|Tue Dec 12 17:02:52 CST 2023
Bot 1|27|Tue Dec 12 17:10:19 CST 2023
Bot 2|50|Tue Dec 12 17:11:18 CST 2023
Bot 1|28|Tue Dec 12 17:11:36 CST 2023
Bot 2|21|Tue Dec 12 17:12:47 CST 2023
Bot 2|13|Tue Dec 12 17:17:55 CST 2023
Bot 2|13|Tue Dec 12 17:19:18 CST 2023
Bot 1|14|Tue Dec 12 17:21:08 CST 2023
Bot 2|14|Tue Dec 12 17:21:40 CST 2023
```

Figura 1: Ejemplo de un historial de partidas

Para la serialización se utilizó la clase Configuración para guardar el *ArrayList* de jugadores y fichas del pozo de una partida en un archivo txt y poder cargarla posteriormente si el usuario lo desea. Esto se logró implementando la interfaz *Serializable* y los métodos *guardarConfiguracion* y *cargarConfiguracion*.

10. Programación concurrente

La programación concurrente se utilizó para el turno de un jugador en la clase *Controlador*, se creó una clase interna llamada *HiloTurno* que extiende de la clase *Thread* y se sobreescribió el método *run* para que el jugador realice su turno. Evitando de esta manera que un jugador pueda jugar mientras el otro esta jugando.

```
/**
```

```
* Representa un hilo para manejar el turno de un jugador en un juego de domino.
*/
public static class HiloTurno extends Thread {
  private Jugador jugador;
  private ArrayList<Ficha> mesa;
  private int turno;
  private int tamanioPozo;

  /**
    * Construye un nuevo objeto HiloTurno.
    *
    * @param jugador   el jugador asociado con el hilo
    * @param mesa      la lista de fichas de domino en la mesa
    * @param turno     el numero de turno actual
    * @param tamanioPozo el tamanio del monton de robo
    */
  public HiloTurno(Jugador jugador, ArrayList<Ficha> mesa, int turno, int tamanioPozo) {
    this.jugador = jugador;
    this.mesa = mesa;
    this.turno = turno;
    this.tamanioPozo = tamanioPozo;
  }

  /**
    * Ejecuta la logica del turno del jugador.
    */
  @Override
  public void run() {
    if (turno == 1) {
      jugador.primerTurno(mesa);
      esperar(ESPERA);
    } else {
      if (jugador.puedeJugar(mesa)) {
        if (jugador instanceof Bot)
          esperar(ESPERA);
        jugador.turno(mesa);
      } else {
        if (tamanioPozo == 0)
          System.out.println("No puedes jugar, pasas el turno");
        else {
          esperar(ESPERA);
          System.out.println("No puedes jugar, robas una ficha");
          esperar(ESPERA);
          jugador.robar(modelo.getPozo());
          Vista.mostrarFichas(jugador.getFichas());
        }
      }
    }
  }
}
```

11. Patrones de diseño

Los patrones de diseño que aplicamos fueron dos: el patrón Modelo-Vista-Controlador *MVC* y el patrón de comportamiento *Template Method*. *MVC* se utilizó en la estructura general del código, dividiéndolo en 3 secciones importantes como su nombre lo indica y utilizando otras clases ya sea para hacer conjunción de objetos o utilizar sus metodos dentro de las clases principales.

Template Method se utilizó en la clase *Jugador* para crear un método *final* con el algoritmo del turno de un jugador, y separando el algoritmo en pasos en forma de métodos abstractos y concretos. De manera que las subclases *Persona* y *Bot* puedan implementar los métodos abstractos de manera diferente y así cambiar el algoritmo del turno de un jugador dependiendo del tipo de jugador.

12. Conclusiones

En este proyecto de implementación de un juego de dominó en Java, se han abordado de manera integral aspectos clave de la programación orientada a objetos y la interacción con el usuario. La estructura del diseño se fundamenta en la encapsulación de fichas, jugadores y la mesa en clases independientes, fomentando así una organización eficiente y facilitando el mantenimiento del código.

La simulación del juego sigue meticulosamente las reglas auténticas del dominó, destacándose por su atención al detalle en la determinación de la jugabilidad de las fichas conforme a las reglas del juego. La gestión de situaciones como el robo de fichas y el cambio de turno se ejecuta de manera efectiva, evidenciando una comprensión profunda de las dinámicas del dominó. Esta capacidad se ha logrado mediante la aplicación de herencia y la implementación de métodos abstractos en las clases de jugadores.

Además, la implementación aborda de manera proactiva la gestión de excepciones, asegurando que el juego sea robusto y confiable en diversas circunstancias. Este enfoque en la robustez del código no solo garantiza una experiencia de juego fluida, sino que también demuestra un compromiso con las mejores prácticas de programación.

La realización de este proyecto no solo ha permitido aplicar las habilidades adquiridas durante el curso, sino que también ha fomentado la investigación y el trabajo colectivo. Este tipo de esfuerzo colaborativo promueve el desarrollo integral de habilidades, consolidando el aprendizaje teórico en un contexto práctico y realista.

13. Referencias

De Tenis De Huelva, R. C. R. (s. f.). *Dominó* — https://rcrtenishuelva1889.com/page/9/domino#:~:text=El%20domin%C3%B3%20surgi%C3%B3%20hace%20mil,los%20italianos%20por%20todas%20partes

Familia Vintage. (2018, June 5). *Como Jugar Al Domino, Reglas del Dominó* [Video]. YouTube. https://www.youtube.com/watch?v=-hbARCT1qow

Historia, C., & Historia, C. (2023, 22 abril). *La historia del dominó, es una de las más interesantes que existe Read more.* CurioSfera Historia. https://curiosfera-historia.com/historia-del-domino-origen-inventor/

Java Platform SE 8. (2023). Oracle.com. https://docs.oracle.com/javase/8/docs/api/

JUEGOS CON HISTORIA: *El Dominó* — Lekotek. (s. f.). https://www.lekotek.org.ar/juegos-con-historia-el-domino/

Solano, J. (2017, 20 enero). *Manual de prácticas de Programación Orientada a Objetos.* Laboratorio de Computación Salas A y B. http://lcp02.fi-b.unam.mx/

14. Apéndice

14.1. Diagramas UML de clases

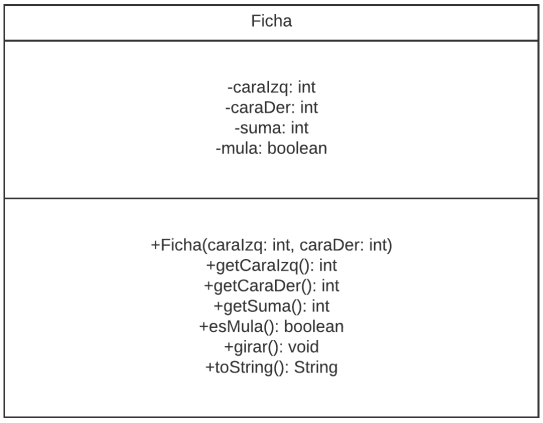

Figura 2: UML de clase *Ficha*

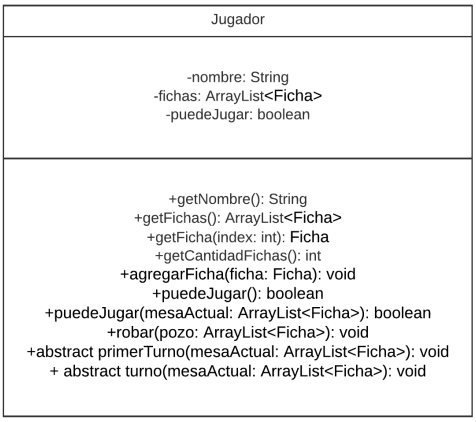

Figura 3: UML de clase *Jugador*

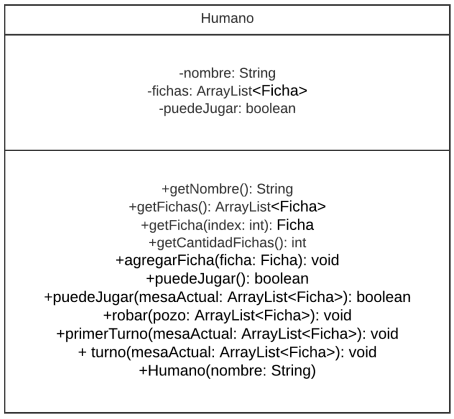

Figura 4: UML de clase *Humano*

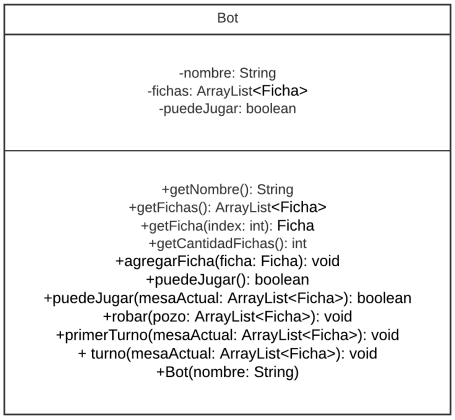

Figura 5: UML de clase *Bot*

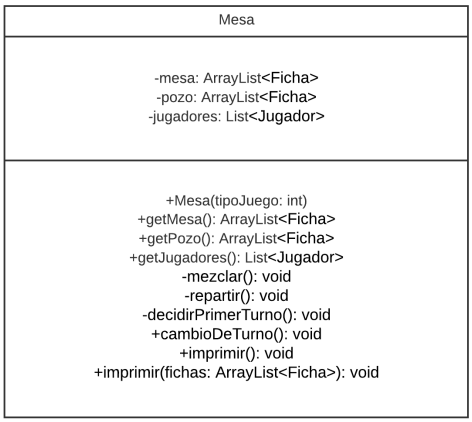

Figura 6: UML de clase *Mesa*

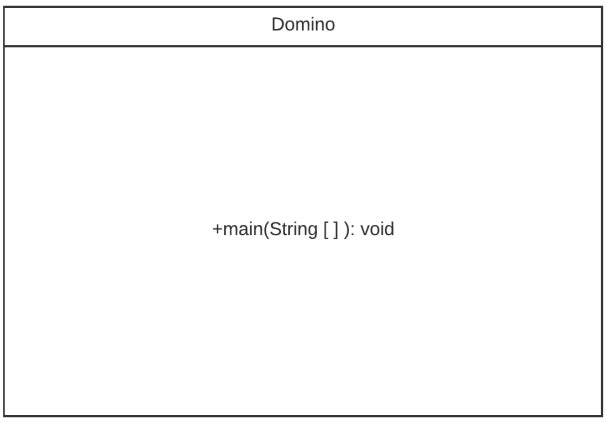

Figura 7: UML de clase *Domino*

14.2. Diagramas UML de objetos

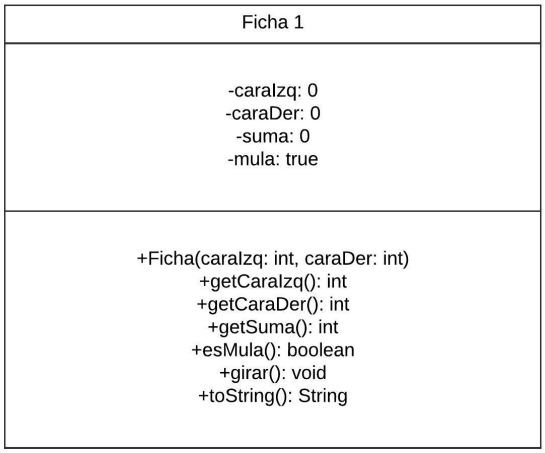

Figura 8: UML de objeto *Ficha1*

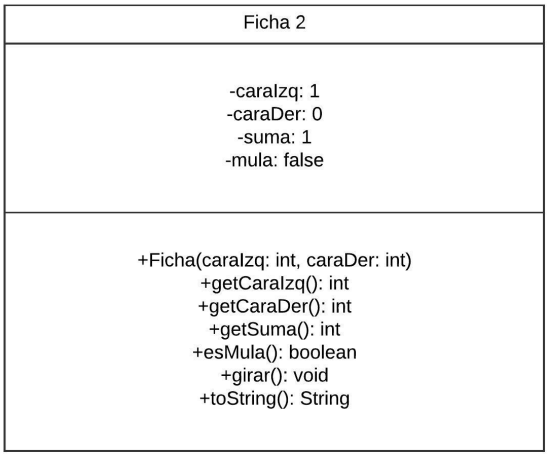

Figura 9: UML de objeto *Ficha2*

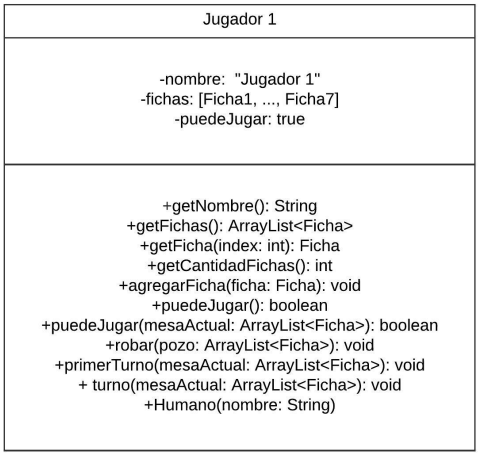

Figura 10: UML de objeto *Jugador1*

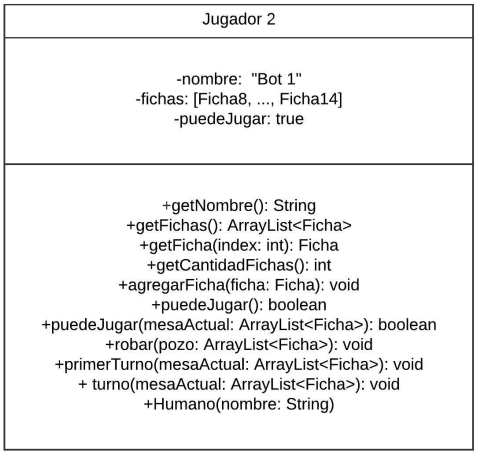

Figura 11: UML de objeto *Jugador1*

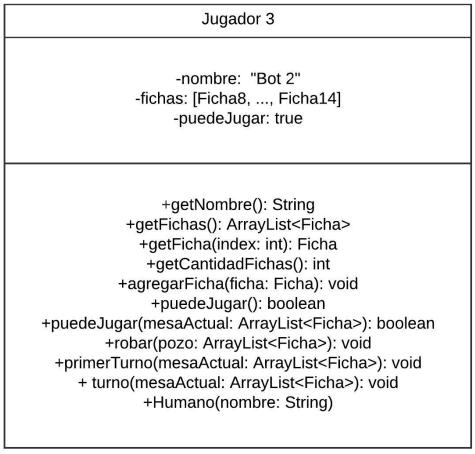

Figura 12: UML de objeto *Jugador1*

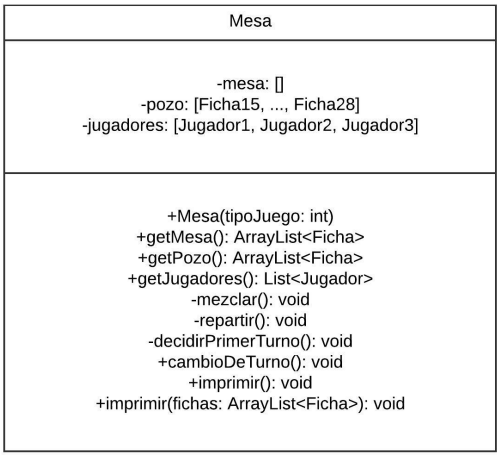

Figura 13: UML de objeto *Mesa*

14.3. Diagrama UML de casos de uso

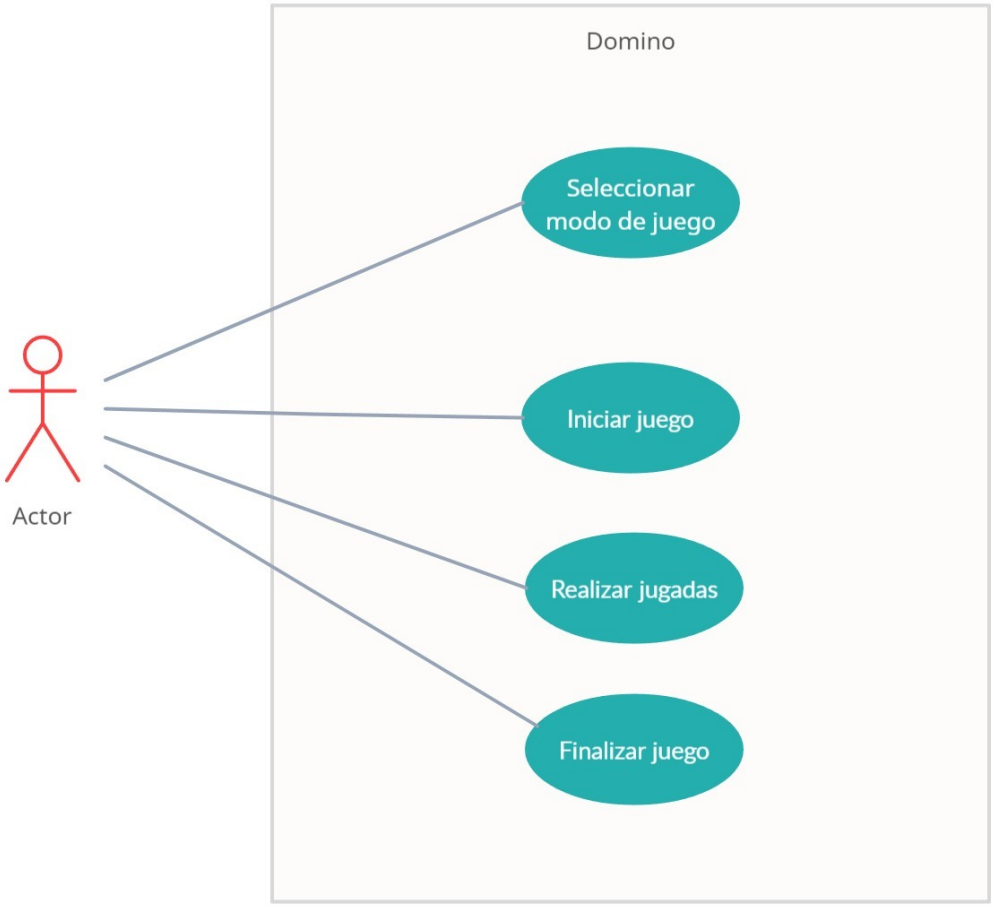

Figura 14: UML de casos de uso

14.4. Diagrama UML de estados de hilos

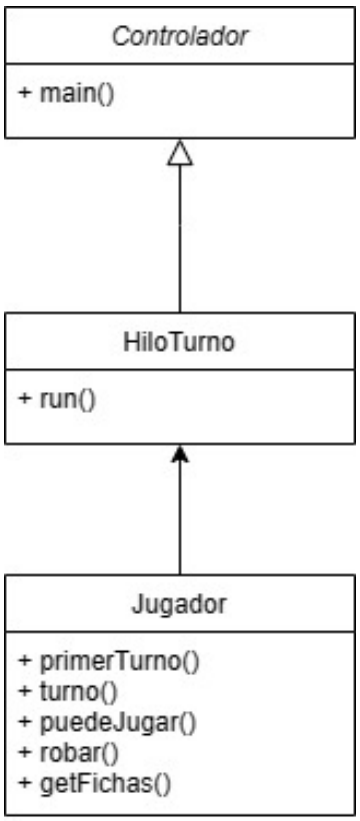

Figura 15: UML de estados de hilos

14.5. Diagrama UML de MVC

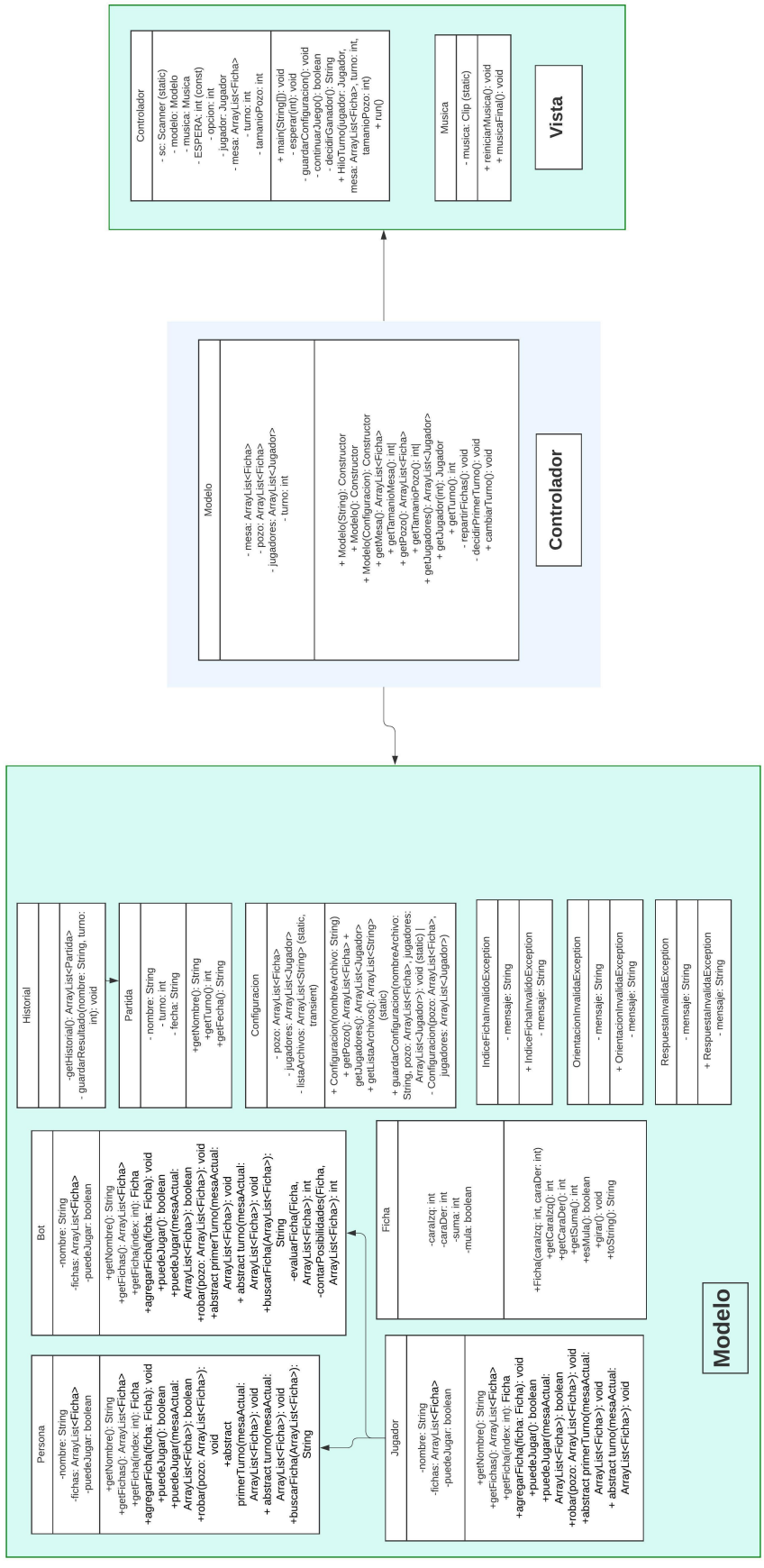

Figura 16: UML de MVC

14.6. Diagrama UML de actividades

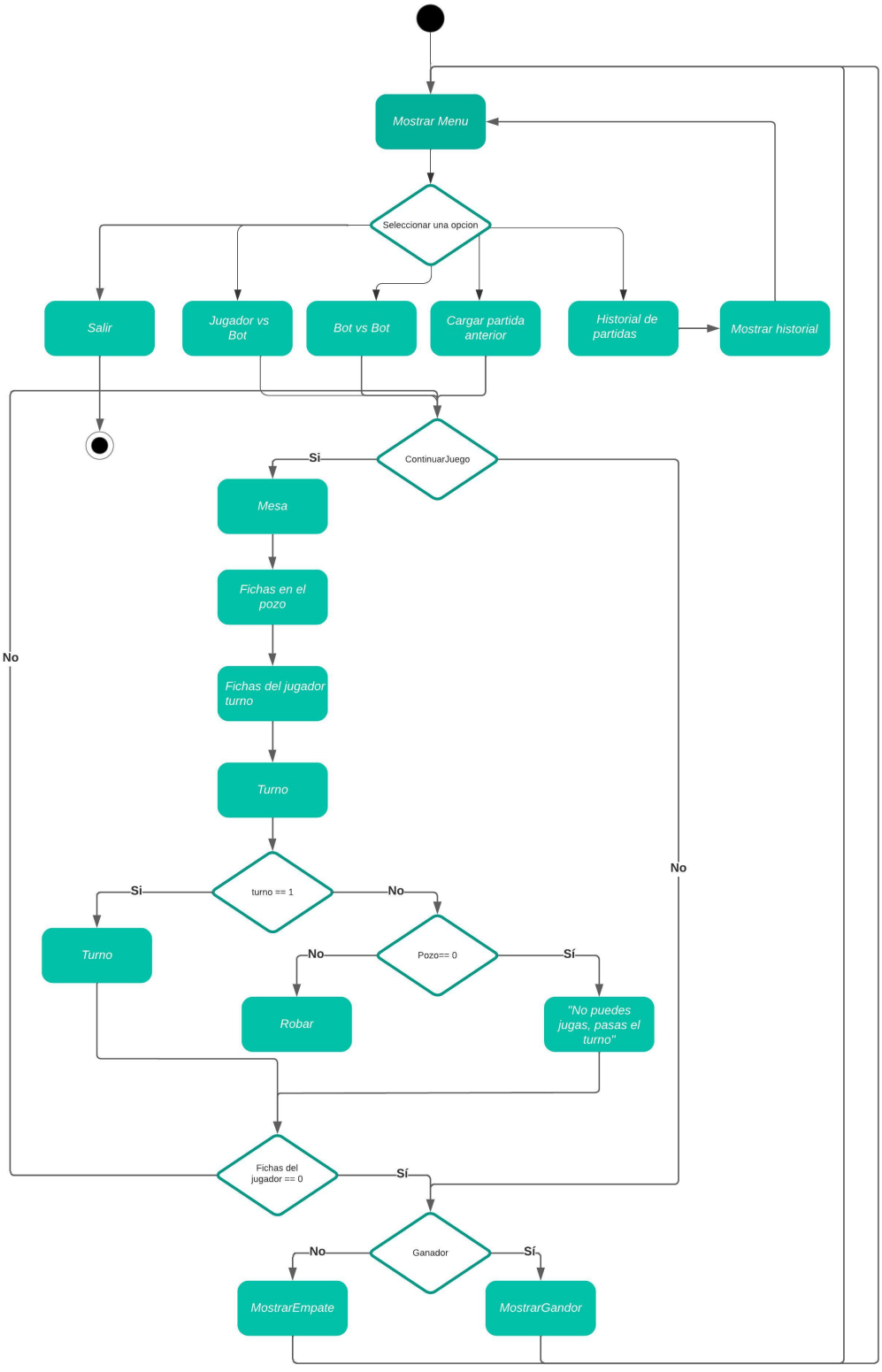

Figura 17: UML de actividades

14.7. Diagrama UML de componentes

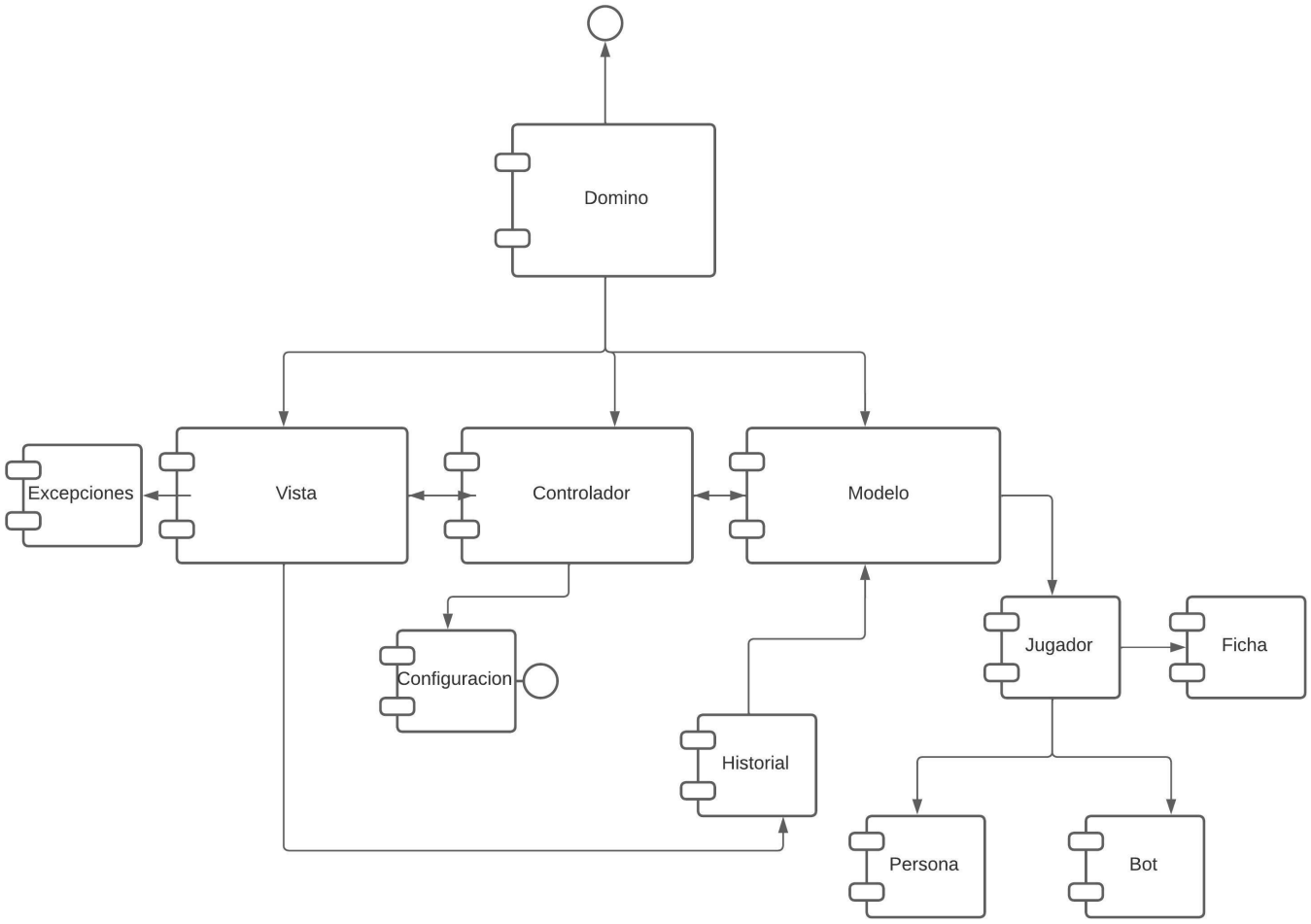

Figura 18: UML de componentes

14.8. Código fuente

14.8.1. Ficha.java

```
package domino;

import java.io.Serializable;

/**
 * Representa una ficha de domino con dos caras.
 * Cada cara tiene un numero, y la ficha puede ser una "mula" si ambas caras son iguales.
 */
public class Ficha implements Serializable {
  private int caraIzq;
  private int caraDer;
  private int suma;
  private boolean mula;

  /**
    * Constructor que recibe las dos caras de la ficha.
    * @param caraIzq Numero de la cara izquierda.
```

```
    * @param caraDer Numero de la cara derecha.
    */
  public Ficha(int caraIzq, int caraDer) {
    this.caraIzq = caraIzq;
    this.caraDer = caraDer;
    this.suma = caraIzq + caraDer;
    this.mula = (caraIzq == caraDer);
  }

  /**
    * Obtiene el numero de la cara izquierda.
    * @return Numero de la cara izquierda.
    */
  public int getCaraIzq() {
    return caraIzq;
  }

  /**
    * Obtiene el numero de la cara derecha.
    * @return Numero de la cara derecha.
    */
  public int getCaraDer() {
    return caraDer;
  }

  /**
    * Obtiene la suma de las caras.
    * @return Suma de las caras.
    */
  public int getSuma() {
    return suma;
  }

  /**
    * Comprueba si la ficha es una mula (ambas caras son iguales).
    * @return `true` si la ficha es una mula, `false` si no.
    */
  public boolean esMula() {
    return mula;
  }

  /**
    * Gira la ficha intercambiando las caras izquierda y derecha.
    */
  public void girar() {
    int aux = caraIzq;
    caraIzq = caraDer;
    caraDer = aux;
  }

  /**
    * Representacion en cadena de la ficha con el formato [caraIzq|caraDer].
    * @return Cadena que representa la ficha.
    */
  @Override
  public String toString() {
    return "[" + caraIzq + "|" + caraDer + "]";
  }
}
```

14.8.2. Jugador.java

```
package domino;

import java.util.ArrayList;
import java.io.Serializable;
```

```java
/**
 * Clase abstracta que representa un jugador de domino.
 */
public abstract class Jugador implements Serializable {
  protected String nombre;
  protected ArrayList<Ficha> fichas = new ArrayList<>();
  protected boolean puedeJugar = true;

  /**
    * Obtiene el nombre del jugador.
    * @return Nombre del jugador.
    */
  public String getNombre() {
    return nombre;
  }

  /**
    * Obtiene las fichas del jugador.
    * @return Lista de fichas del jugador.
    */
  public ArrayList<Ficha> getFichas() {
    return fichas;
  }

  /**
    * Obtiene una ficha especifica del jugador.
    * @param i Indice de la ficha.
    * @return Ficha del jugador.
    */
  public Ficha getFicha(int i) {
    return fichas.get(i);
  }

  /**
    * Agrega una ficha a las fichas del jugador.
    * @param ficha Ficha a agregar.
    */
  public void agregarFicha(Ficha ficha) {
    fichas.add(ficha);
  }

  /**
    * Indica si el jugador puede jugar.
    * @return `true` si el jugador puede jugar, `false` si no.
    */
  public boolean puedeJugar() {
    return puedeJugar;
  }

  /**
    * Verifica si el jugador puede jugar con las fichas disponibles.
    * @param mesa Fichas de la mesa actual.
    * @return `true` si el jugador puede jugar, `false` si no.
    */
  public boolean puedeJugar(ArrayList<Ficha> mesa) {
    puedeJugar = fichas.stream().anyMatch(ficha ->
      ficha.getCaraIzq() == mesa.get(0).getCaraIzq() ||
      ficha.getCaraDer() == mesa.get(0).getCaraIzq() ||
      ficha.getCaraIzq() == mesa.get(mesa.size() - 1).getCaraDer() ||
      ficha.getCaraDer() == mesa.get(mesa.size() - 1).getCaraDer()
    );
    return puedeJugar;
  }

  /**
    * Roba una ficha del pozo y la agrega a las fichas del jugador.
    * @param pozo Pozo de fichas.
    */
  public void robar(ArrayList<Ficha> pozo) {
```

```
    fichas.add(pozo.get(0));
    pozo.remove(0);
  }

  /**
    * Realiza la jugada del primer turno, eligiendo la ficha mas alta.
    * @param mesa Fichas de la mesa.
    */
  public void primerTurno(ArrayList<Ficha> mesa) {
    int indice = -1;
    int max = -1;

    for (Ficha ficha : fichas) {
      if (ficha.esMula() && ficha.getSuma() > max) {
        max = ficha.getSuma();
        indice = fichas.indexOf(ficha);
      }
    }

    if (indice != -1)
      System.out.println("\n" + nombre + " juega la mula mas alta: " + fichas.get(indice));
    else {
      for (Ficha ficha : fichas) {
        if (ficha.getSuma() > max) {
          max = ficha.getSuma();
          indice = fichas.indexOf(ficha);
        }
      }
      System.out.println("\n" + nombre + " juega la ficha mas alta: " + fichas.get(indice));
    }

    mesa.add(fichas.get(indice));
    fichas.remove(indice);
  }

  /**
    * Realiza la jugada en un turno normal.
    * @param mesa Fichas de la mesa.
    */
  public final void turno(ArrayList<Ficha> mesa) {
    int indice = -1;
    char orientacion;
    String respuesta;
    boolean validacion;

    do {
      respuesta = buscarFicha(mesa);
      orientacion = respuesta.charAt(0);
      indice = Integer.parseInt(respuesta.substring(1));
      validacion = validarFicha(mesa, indice, orientacion);
      if (!validacion)
        System.out.println("Ficha invalida\n");
    } while (!validacion);

    System.out.println("\n" + nombre + " juegas la ficha: " + (indice + 1) + " - " + fichas.get(indice));

    jugarFicha(mesa, indice, orientacion);
  }

  /**
    * Metodo abstracto para buscar la ficha a jugar.
    * @param mesa Fichas de la mesa.
    * @return Representacion de la jugada (orientacion + indice).
    */
  protected abstract String buscarFicha(ArrayList<Ficha> mesa);

  /**
    * Valida si una ficha puede ser jugada en la mesa en la orientacion indicada.
    * @param mesa Fichas de la mesa actual.
```

```
    * @param indice indice de la ficha a jugar.
    * @param orientacion Orientacion de la ficha ('i' o 'd').
    * @return true si la ficha es valida para jugar, false si no.
    */
  private boolean validarFicha(ArrayList<Ficha> mesa, int indice, char orientacion) {
    switch (orientacion) {
      case 'i':
        if (fichas.get(indice).getCaraIzq() == mesa.get(0).getCaraIzq()) {
          fichas.get(indice).girar();
          return true;
        }
        if (fichas.get(indice).getCaraDer() == mesa.get(0).getCaraIzq())
          return true;
        return false;

      case 'd':
        if (fichas.get(indice).getCaraDer() == mesa.get(mesa.size() - 1).getCaraDer()) {
          fichas.get(indice).girar();
          return true;
        }
        if (fichas.get(indice).getCaraIzq() == mesa.get(mesa.size() - 1).getCaraDer())
          return true;
        return false;

      default:
        return false;
    }
  }

  /**
    * Juega la ficha en la mesa segun la orientacion.
    * Este metodo es sincronizado para evitar que dos jugadores jueguen la misma ficha.
    * @param mesa Fichas de la mesa.
    * @param indice Indice de la ficha a jugar.
    * @param orientacion Orientacion de la ficha ('i' o 'd').
    */
  private synchronized void jugarFicha(ArrayList<Ficha> mesa, int indice, char orientacion) {
    if (orientacion == 'i')
      mesa.add(0, fichas.get(indice));
    else if (orientacion == 'd')
      mesa.add(fichas.get(indice));
    fichas.remove(indice);
  }
}
```

14.8.3. Persona.java

```
package domino;

import java.util.ArrayList;
import java.util.Scanner;

/**
 * Esta clase representa un jugador humano de domino.
 */
public class Persona extends Jugador {
  /**
    * Constructor de la clase Persona.
    * @param nombre Nombre del jugador.
    */
  public Persona(String nombre) {
    this.nombre = nombre;
  }

  /**
    * Metodo sobreescrito para permitir al jugador humano seleccionar una ficha
    * para jugar y la orientacion en la que desea colocarla.
```

```
    * @param mesaActual Lista de fichas en la mesa actual.
    * @return Cadena que representa la accion del jugador (orientacion + indice).
    */
  @Override
  protected String buscarFicha(ArrayList<Ficha> mesaActual) {
    Scanner sc = new Scanner(System.in);
    int indice = -1;
    char orientacion = ' ';

    do {
      try {
        System.out.println("\nQue ficha deseas jugar? (1 - " + fichas.size() + ")");
        System.out.print("$ ");
        indice = sc.nextInt() - 1;
        if (indice < 0 || indice >= fichas.size())
          throw new IndiceFichaInvalidoException("Indice de ficha no valido. Debe ser entre 1 y " + fichas.size());
      } catch (IndiceFichaInvalidoException e) {
          System.out.println(e.getMessage() + "\n");
      }
    } while (indice < 0 || indice >= fichas.size());

    do {
      try {
        System.out.println("Donde la quieres jugar? (i o d)");
        System.out.print("$ ");
        orientacion = sc.next().charAt(0);
        if (orientacion != 'i' && orientacion != 'd')
          throw new OrientacionInvalidaException("Orientacion no valida. Debe ser 'i' o 'd'.");
      } catch (OrientacionInvalidaException e) {
        System.out.println(e.getMessage() + "\n");
      }
    } while (orientacion != 'i' && orientacion != 'd');

    return orientacion + "" + indice;
  }
}
```

14.8.4. Bot.java

```
package domino;

import java.util.ArrayList;

/**
 * Esta clase representa un jugador bot de domino.
 * Extiende la clase abstracta Jugador.
 */
public class Bot extends Jugador {
  /**
    * Constructor de la clase Bot.
    * @param nombre Nombre del bot.
    */
  public Bot(String nombre) {
    this.nombre = nombre;
  }

  /**
    * Metodo para que el bot elija una ficha para jugar en la mesa.
    * Evalua sus fichas disponibles y selecciona la que maximiza sus posibilidades de juego.
    * @param mesa La lista de fichas en la mesa.
    * @return Intruccion que representa la eleccion de la ficha y su orientacion ('i' o 'd').
    */
  @Override
  protected String buscarFicha(ArrayList<Ficha> mesa) {
    Ficha mejorFicha = null;
    int mejorPuntuacion = Integer.MIN_VALUE;
```

```
    for (Ficha ficha : fichas) {
      int puntuacion = evaluarFicha(ficha, mesa);
      if (puntuacion > mejorPuntuacion) {
        mejorPuntuacion = puntuacion;
        mejorFicha = ficha;
      }
    }

    if (mejorFicha != null) {
      int indice = fichas.indexOf(mejorFicha);
      if (mejorFicha.getCaraIzq() == mesa.get(0).getCaraIzq())
          return "i" + indice;
      else if (mejorFicha.getCaraDer() == mesa.get(0).getCaraIzq())
          return "i" + indice;
      else if (mejorFicha.getCaraIzq() == mesa.get(mesa.size() - 1).getCaraDer())
          return "d" + indice;
      else if (mejorFicha.getCaraDer() == mesa.get(mesa.size() - 1).getCaraDer())
          return "d" + indice;
    }

    return elegirFichaGenerica(mesa);
  }

  /**
    * Evalua una ficha especifica en funcion de las posibilidades de juego en la mesa.
    * @param ficha Ficha a evaluar.
    * @param mesa Lista de fichas en la mesa.
    * @return Cantidad de posibilidades de juego para la ficha.
    */
  private int evaluarFicha(Ficha ficha, ArrayList<Ficha> mesa) {
    int cantidadPosibilidades = contarPosibilidades(ficha, mesa);
    return cantidadPosibilidades;
  }

  /**
    * Cuenta la cantidad de posibilidades de juego para una ficha en la mesa.
    * @param ficha Ficha a evaluar.
    * @param mesa Lista de fichas en la mesa.
    * @return Cantidad de posibilidades de juego para la ficha.
    */
  private int contarPosibilidades(Ficha ficha, ArrayList<Ficha> mesa) {
    int cantidadPosibilidades = 0;

    for (Ficha mesaFicha : mesa) {
      if (ficha.getCaraIzq() == mesaFicha.getCaraIzq() || ficha.getCaraDer() == mesaFicha.getCaraDer())
        cantidadPosibilidades++;
      if (ficha.getCaraIzq() == mesaFicha.getCaraDer() || ficha.getCaraDer() == mesaFicha.getCaraIzq())
        cantidadPosibilidades++;
    }

    return cantidadPosibilidades;
  }

  /**
    * Elige una ficha generica si el bot no puede seleccionar una especifica.
    * @param mesa Lista de fichas en la mesa.
    * @return Intruccion que representa la eleccion de la ficha y su orientacion ('i' o 'd').
    */
  private String elegirFichaGenerica(ArrayList<Ficha> mesa) {
    for (Ficha ficha : fichas) {
      if (ficha.getCaraIzq() == mesa.get(0).getCaraIzq())
      return "i" + fichas.indexOf(ficha);
      else if (ficha.getCaraDer() == mesa.get(0).getCaraIzq())
        return "i" + fichas.indexOf(ficha);
      else if (ficha.getCaraIzq() == mesa.get(mesa.size() - 1).getCaraDer())
        return "d" + fichas.indexOf(ficha);
      else if (ficha.getCaraDer() == mesa.get(mesa.size() - 1).getCaraDer())
        return "d" + fichas.indexOf(ficha);
    }
```

```
    return null;
  }
}
```

14.8.5. Modelo.java

```
package domino;

import java.util.ArrayList;
import java.util.Random;
import java.util.stream.Collectors;

/**
 * Esta clase representa el modelo del juego de domino.
 */
public class Modelo {
  private ArrayList<Ficha> mesa = new ArrayList<Ficha>();
  private ArrayList<Ficha> pozo = new ArrayList<Ficha>();
  private ArrayList<Jugador> jugadores = new ArrayList<Jugador>();
  private int turno;

  /**
    * Constructor para una partida entre una persona y un bot.
    * @param nombre Nombre del jugador persona.
    */
  public Modelo(String nombre) {
    jugadores.add(new Persona(nombre));
    jugadores.add(new Bot("Bot"));

    repartirFichas();
    decidirPrimerTurno();
    turno = 1;
  }

  /**
    * Constructor para una partida entre dos bots.
    */
  public Modelo() {
    jugadores.add(new Bot("Bot 1"));
    jugadores.add(new Bot("Bot 2"));

    repartirFichas();
    decidirPrimerTurno();
    turno = 1;
  }

  /**
    * Constructor para cargar una partida desde una configuracion.
    * @param conf Configuracion de la partida guardada.
    */
  public Modelo(Configuracion conf) {
    pozo = conf.getPozo();
    jugadores = conf.getJugadores();
    turno = 1;
  }

  /**
    * Obtiene las fichas de la mesa.
    * @return Lista de fichas en la mesa.
    */
  public ArrayList<Ficha> getMesa() {
    return mesa;
  }

  /**
    * Obtiene el tamanio de la mesa.
```

```
   * @return Tamanio de la mesa.
   */
  public int getTamanioMesa() {
    return mesa.size();
  }

  /**
   * Obtiene las fichas en el pozo.
   * @return Lista de fichas en el pozo.
   */
  public ArrayList<Ficha> getPozo() {
    return pozo;
  }

  /**
   * Obtiene el tamanio del pozo.
   * @return Tamanio del pozo.
   */
  public int getTamanioPozo() {
    return pozo.size();
  }

  /**
   * Obtiene la lista de jugadores.
   * @return Lista de jugadores.
   */
  public ArrayList<Jugador> getJugadores() {
    return jugadores;
  }

  /**
   * Obtiene un jugador especifico.
   * @param i Indice del jugador.
   * @return Jugador en la posicion especificada.
   */
  public Jugador getJugador(int i) {
    return jugadores.get(i);
  }

  /**
   * Obtiene el turno actual.
   * @return Numero del turno.
   */
  public int getTurno() {
    return turno;
  }

  /**
   * Reparte las fichas a los jugadores.
   */
  private void repartirFichas() {
    Random random = new Random();

    for (int i = 0; i <= 6; i++)
      for (int j = i; j <= 6; j++)
        pozo.add(new Ficha(i, j));

    pozo = pozo.stream().sorted((ficha1, ficha2) -> random.nextInt(3) - 1)
        .collect(Collectors.toCollection(ArrayList::new));

    for (int i = 0; i < 7; i++)
      jugadores.forEach(jugador -> jugador.agregarFicha(pozo.remove(0)));
  }

  /**
   * Decide quiden empieza el primer turno.
   */
  private void decidirPrimerTurno() {
    int max = -1, posicion = -1;
```

```
    for (Jugador jugador : jugadores) {
      for (int i = 0; i < 7; i++) {
        if (jugador.getFicha(i).esMula() && jugador.getFicha(i).getSuma() > max) {
          max = jugador.getFicha(i).getSuma();
         posicion = jugadores.indexOf(jugador);
        }
      }
    }

    if (posicion != -1) {
      if (posicion == 1)
          cambiarTurno();
      return;
    }

    max = -1;
    for (Jugador jugador : jugadores) {
      for (int i = 0; i < 7; i++) {
        if (jugador.getFicha(i).getSuma() > max) {
          max = jugador.getFicha(i).getSuma();
          posicion = jugadores.indexOf(jugador);
        }
      }
    }

    if (posicion == 1)
      cambiarTurno();
  }

  /**
    * Cambia el turno entre los jugadores.
    */
  public void cambiarTurno() {
    Jugador aux = jugadores.get(0);
    jugadores.set(0, jugadores.get(1));
    jugadores.set(1, aux);
    turno++;
  }
}
```

14.8.6. Historial.java

```
package domino;

import java.util.ArrayList;
import java.util.StringTokenizer;
import java.util.Date;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.PrintWriter;
import java.io.FileWriter;

/**
 * Gestiona el historial de partidas guardando y recuperando informacion sobre las mismas.
 */
public class Historial {
  /**
    * Obtiene el historial de partidas guardadas.
    * @return Lista de objetos ‘Partida‘ que representan el historial.
    */
  public static ArrayList<Partida> getHistorial() {
    ArrayList<Partida> historial = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader("historial/historial.txt"))) {
      br.lines().forEach(linea -> {
        StringTokenizer st = new StringTokenizer(linea, "|");
```

```
        String nombre = st.nextToken();
        int turno = Integer.parseInt(st.nextToken());
        String fecha = st.nextToken();
        historial.add(new Partida(nombre, turno, fecha));
      });
    } catch (Exception e) {
      System.out.println("Error al leer el historial");
    }
    return historial;
  }

  /**
   * Guarda el resultado de una partida en el historial.
   * @param nombre Nombre del ganador de la partida.
   * @param turno Numero de turnos que duro la partida.
   */
  public static void guardarResultado(String nombre, int turno) {
    try (PrintWriter ps = new PrintWriter(new FileWriter("historial/historial.txt", true))) {
      Date fecha = new Date();
      ps.println(nombre + "|" + turno + "|" + fecha.toString());
    } catch (Exception e) {
      System.out.println("Error al guardar el resultado");
    }
  }
}
```

14.8.7. Partida.java

```
package domino;

/**
 * Esta clase representa una partida de domino con informacion sobre el nombre del jugador,
 * el turno en el que finalizo la partida y la fecha en la que se jugo.
 */
public class Partida {
  private String nombre;
  private int turno;
  private String fecha;

  /**
   * Constructor de la clase Partida.
   * @param nombre Nombre del jugador.
   * @param turno Turno en el que finalizo la partida.
   * @param fecha Fecha en la que se jugo la partida.
   */
  public Partida(String nombre, int turno, String fecha) {
    this.nombre = nombre;
    this.turno = turno;
    this.fecha = fecha;
  }

  /**
   * Obtiene el nombre del jugador.
   * @return Nombre del jugador.
   */
  public String getNombre() {
    return nombre;
  }

  /**
   * Obtiene el turno en el que finalizo la partida.
   * @return Turno en el que finalizo la partida.
   */
  public int getTurno() {
    return turno;
  }
```

```
  /**
    * Obtiene la fecha en la que se jugo la partida.
    * @return Fecha en la que se jugo la partida.
    */
  public String getFecha() {
    return fecha;
  }
}
```

14.8.8. Configuracion.java

```
package domino;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Collectors;

import java.io.File;
import java.io.Serializable;
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.FileInputStream;

/**
 * Esta clase representa la configuracion de una partida de domino.
 * Contiene informacion sobre el pozo de fichas y los jugadores.
 */
public class Configuracion implements Serializable {
  private ArrayList<Ficha> pozo = new ArrayList<>();
  private ArrayList<Jugador> jugadores = new ArrayList<>();
  private static transient ArrayList<String> listaArchivos= new ArrayList<>();

  /**
    * Constructor de la clase Configuracion.
    * Carga la configuracion desde un archivo dado.
    * @param nombreArchivo Nombre del archivo de configuracion.
    */
  public Configuracion(String nombreArchivo) {
    try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("configuraciones/" + nombreArchivo))) {
      Configuracion conf = (Configuracion) ois.readObject();
      this.pozo = conf.getPozo();
      this.jugadores = conf.getJugadores();
    } catch (Exception e) {
      System.out.println("Error al cargar la configuracion");
    }
  }

  /**
    * Constructor privado de la clase Configuracion.
    * Se utiliza para crear una nueva configuracion.
    * @param pozo Lista de fichas en el pozo.
    * @param jugadores Lista de jugadores.
    */
  private Configuracion(ArrayList<Ficha> pozo, ArrayList<Jugador> jugadores) {
    this.pozo = pozo;
    this.jugadores = jugadores;
  }

  /**
    * Obtiene la lista de fichas en el pozo.
    * @return Lista de fichas en el pozo.
    */
  public ArrayList<Ficha> getPozo() {
    return pozo;
  }
```

```
  /**
    * Obtiene la lista de jugadores.
    * @return Lista de jugadores.
    */
  public ArrayList<Jugador> getJugadores() {
    return jugadores;
  }

  /**
    * Obtiene la lista de nombres de archivos de configuracion disponibles.
    * @return Lista de nombres de archivos de configuracion.
    */
  public static ArrayList<String> getListaArchivos() {
    listaArchivos.clear();
    File carpeta = new File("configuraciones");

    if (carpeta.exists() && carpeta.isDirectory()) {
      File[] archivos = carpeta.listFiles();

      if (archivos != null) {
        listaArchivos.addAll(Arrays.stream(archivos)
            .map(File::getName)
            .collect(Collectors.toList()));
        return listaArchivos;
      } else
        System.out.println("La carpeta esta vacia.");
    } else
      System.out.println("La carpeta no existe o no es un directorio.");

    return null;
  }

  /**
    * Guarda una nueva configuracion en un archivo dado.
    * @param nombreArchivo Nombre del archivo de configuracion.
    * @param pozo Lista de fichas en el pozo.
    * @param jugadores Lista de jugadores.
    */
  public static void guardarConfiguracion(String nombreArchivo, ArrayList<Ficha> pozo, ArrayList<Jugador> jugadores) {
    try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("configuraciones/" + nombreArchivo + ".txt"))) {
      oos.writeObject(new Configuracion(pozo, jugadores));
    } catch (Exception e) {
      System.out.println("Error al guardar la configuracion");
    }
  }
}
```

14.8.9. Musica.java

```
package domino;

import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;

/**
 * Esta clase gestiona la reproduccion de la musica del juego de domino.
 */
public class Musica {
  private static Clip musica = null;

  /**
    * Reinicia la musica del juego.
    */
  public void reiniciarMusica() {
    try {
      if (musica != null)
        musica.stop();
```

```
      musica = AudioSystem.getClip();
      musica.open(AudioSystem.getAudioInputStream(Controlador.class.getResource("/recursos/MoonlightSonata.wav")));
      musica.loop(Clip.LOOP_CONTINUOUSLY);
    } catch (Exception e) {}
  }

  /**
   * Reproduce la musica de victoria al finalizar el juego.
   */
  public void musicaFinal() {
    try {
      if (musica != null)
        musica.stop();

      musica = AudioSystem.getClip();
      musica.open(AudioSystem.getAudioInputStream(Controlador.class.getResource("/recursos/Victoria.wav")));
      musica.start();
    } catch (Exception e) {}
  }
}
```

14.8.10. Controlador.java

```
package domino;

import java.util.Scanner;
import java.util.ArrayList;

/**
 * Esta clase actua como el controlador principal del juego de domino.
 */
public class Controlador {
  private static Scanner sc = new Scanner(System.in);
  private static Modelo modelo;
  private static Musica musica = new Musica();
  private static final int ESPERA = 3000;
  private static int opcion = 0;

  /**
   * Metodo principal que inicia y controla el juego de domino.
   * Permite a los jugadores iniciar una nueva partida, cargar una partida
   * guardada o ver el historial.
   * Ademas, gestiona el desarrollo del juego, la logica del turno y decide al
   * ganador.
   */
  public static void main(String[] args) {
    while (true) {
      Vista.limpiarPantalla();

      Vista.mostrarMenu();
      opcion = sc.nextInt();

      Vista.limpiarPantalla();
      switch (opcion) {
        case 1:
          Vista.pedirNombre();
          String nombre = sc.next();
          modelo = new Modelo(nombre);
          guardarConfiguracion();
          break;
        case 2:
          modelo = new Modelo();
          guardarConfiguracion();
          break;
        case 3:
          Vista.mostrarHistorial(Historial.getHistorial());
```

```
        Vista.continuar();
        continue;
      case 4:
        ArrayList<String> configuraciones = Configuracion.getListaArchivos();
        if (configuraciones == null) {
          Vista.continuar();
          continue;
        } else {
          Vista.mostrarArchivos(configuraciones);
          System.out.println("\nIngrese el numero de la partida que desea cargar");
          System.out.print("$ ");
          int respuesta = sc.nextInt();
          if (respuesta > 0 && respuesta <= configuraciones.size()) {
            Configuracion conf = new Configuracion(
                configuraciones.get(respuesta - 1));
            modelo = new Modelo(conf);
          } else
            continue;
        }
        break;
      default:
        System.exit(0);
    }

    musica.reiniciarMusica();
    while (continuarJuego()) {
      Vista.limpiarPantalla();

      Vista.mostrarMesa(modelo.getMesa());
      Vista.mostrarPozo(modelo.getTamanioPozo());

      Vista.mostrarTurno(modelo.getJugador(0).getNombre());
      Vista.mostrarFichas(modelo.getJugador(0).getFichas());

      HiloTurno hiloJugador = new HiloTurno(modelo.getJugador(0), modelo.getMesa(), modelo.getTurno(),
          modelo.getTamanioPozo());
      hiloJugador.start();

      try {
        hiloJugador.join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }

      esperar(ESPERA);

      if (modelo.getJugador(0).getFichas().isEmpty())
        break;

      modelo.cambiarTurno();
    }

    Vista.limpiarPantalla();
    Vista.mostrarMesa(modelo.getMesa());

    musica.musicaFinal();
    String resultado = decidirGanador();
    if (resultado.equals("Empate"))
      Vista.mostrarEmpate();
    else
      Vista.mostrarGanador(resultado);

    Historial.guardarResultado(resultado, modelo.getTurno());
    Vista.continuar();
  }
}

/**
 * Representa un hilo para manejar el turno de un jugador en un juego de domino.
```

```
  */
public static class HiloTurno extends Thread {
  private Jugador jugador;
  private ArrayList<Ficha> mesa;
  private int turno;
  private int tamanioPozo;

  /**
   * Construye un nuevo objeto HiloTurno.
   *
   * @param jugador   el jugador asociado con el hilo
   * @param mesa      la lista de fichas de domino en la mesa
   * @param turno     el numero de turno actual
   * @param tamanioPozo el tamanio del monton de robo
   */
  public HiloTurno(Jugador jugador, ArrayList<Ficha> mesa, int turno, int tamanioPozo) {
    this.jugador = jugador;
    this.mesa = mesa;
    this.turno = turno;
    this.tamanioPozo = tamanioPozo;
  }

  /**
   * Ejecuta la logica del turno del jugador.
   */
  @Override
  public void run() {
    if (turno == 1) {
      jugador.primerTurno(mesa);
      esperar(ESPERA);
    } else {
      if (jugador.puedeJugar(mesa)) {
        if (jugador instanceof Bot)
          esperar(ESPERA);
        jugador.turno(mesa);
      } else {
        if (tamanioPozo == 0)
          System.out.println("No puedes jugar, pasas el turno");
        else {
          esperar(ESPERA);
          System.out.println("No puedes jugar, robas una ficha");
          esperar(ESPERA);
          jugador.robar(modelo.getPozo());
          Vista.mostrarFichas(jugador.getFichas());
        }
      }
    }
  }
}

/**
 * Hace que el programa espere un tiempo determinado.
 *
 * @param tiempo Tiempo en milisegundos.
 */
private static void esperar(int tiempo) {
  try {
    Thread.sleep(tiempo);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}

/**
 * Pregunta al jugador si desea guardar la configuracion de la partida.
 * Si la respuesta es afirmativa, solicita un nombre y guarda la configuracion.
 */
private static void guardarConfiguracion() {
  System.out.println("\nQuieres guardar la configuracion de la partida? (s/n)");
```

```
    System.out.print("$ ");
    char respuesta = sc.next().charAt(0);
    respuesta = Character.toLowerCase(respuesta);
    if (respuesta == 's') {
      System.out.println("\nIngrese el nombre de la partida");
      System.out.print("$ ");
      String nombre = sc.next();
      Configuracion.guardarConfiguracion(nombre, modelo.getPozo(), modelo.getJugadores());
    }
  }

  /**
   * Verifica si el juego debe continuar, basandose en si los jugadores pueden
   * jugar o si hay fichas en el pozo.
   *
   * @return `true` si el juego debe continuar, `false` si no.
   */
  private static boolean continuarJuego() {
    return (modelo.getJugador(0).puedeJugar() ||
        modelo.getJugador(1).puedeJugar()) ||
        modelo.getTamanioPozo() != 0;
  }

  /**
   * Decide al ganador basandose en las condiciones del juego.
   *
   * @return Nombre del ganador o "Empate".
   */
  private static String decidirGanador() {
    if (!modelo.getJugador(0).puedeJugar() &&
          !modelo.getJugador(1).puedeJugar() &&
          modelo.getTamanioPozo() == 0) {
      int fichasJugador1 = modelo.getJugador(0).getFichas().size();
      int fichasJugador2 = modelo.getJugador(1).getFichas().size();

      if (fichasJugador1 < fichasJugador2)
        return modelo.getJugador(0).getNombre();
      else if (fichasJugador1 > fichasJugador2)
        return modelo.getJugador(1).getNombre();
      else
        return "Empate";
    } else
      return modelo.getJugador(0).getNombre();
  }
}
```

14.8.11. Vista.java

```
package domino;

import java.util.ArrayList;
import java.util.Scanner;

/**
 * Clase que maneja la interfaz y la presentacion del juego de domino.
 */
public class Vista {
  /**
   * Metodo para limpiar la pantalla.
   */
  public static void limpiarPantalla() {
    // Windows
    try {
      new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();
    } catch (Exception e) {}

    // Linux
```

```
    try {
      new ProcessBuilder("clear").inheritIO().start().waitFor();
    } catch (Exception e) {}
  }

  /**
   * Metodo para mostrar un mensaje y esperar la confirmacion del usuario.
   */
  public static void continuar() {
    Scanner sc = new Scanner(System.in);
    System.out.print("\n\nPresione enter para continuar...");
    sc.nextLine();
  }

  /**
   * Metodo para pedir el nombre al usuario.
   */
  public static void pedirNombre() {
    System.out.println("%%%%%%%%%%%%%%% Ingrese su nombre %%%%%%%%%%%%%%%");
    System.out.print("$ ");
  }

  /**
   * Metodo para mostrar el menu del juego.
   */
  public static void mostrarMenu() {
    System.out.println("%%%%%%%%%%%%%%% Bienvenido a Domino %%%%%%%%%%%%%%%");
    System.out.println("Escoja el modo de juego:");
    System.out.println("1. Jugador vs Bot");
    System.out.println("2. Bot vs Bot");
    System.out.println("3. Historial de partidas");
    System.out.println("4. Cargar partida anterior");
    System.out.println("5. Salir");
    System.out.print("$ ");
  }

  /**
   * Metodo para mostrar la cantidad de fichas en el pozo.
   * @param pozo Numero de fichas en el pozo.
   */
  public static void mostrarPozo(int pozo) {
    System.out.println("\nNumero de fichas en el Pozo: " + pozo);
  }

  /**
   * Metodo para mostrar el turno del jugador.
   * @param nombre Nombre del jugador actual.
   */
  public static void mostrarTurno(String nombre) {
    System.out.println("\n%%%%% Turno de " + nombre + " %%%%%");
  }

  /**
   * Metodo para mostrar las fichas en la mesa.
   * @param mesa Lista de fichas en la mesa.
   */
  public static void mostrarMesa(ArrayList<Ficha> mesa) {
    System.out.println("Mesa actual:");
    for (Ficha ficha : mesa)
      System.out.print(ficha + "\t");
    System.out.println();
  }

  /**
   * Metodo para mostrar las fichas del jugador.
   * @param fichas Lista de fichas del jugador.
   */
  public static void mostrarFichas(ArrayList<Ficha> fichas) {
    System.out.println("\nTus fichas:");
```

```
    for (int i = 1; i <= fichas.size(); i++)
      System.out.print(" " + i + "\t");
    System.out.println();
    for (Ficha ficha : fichas)
      System.out.print(ficha + "\t");
    System.out.println();
  }

  /**
    * Metodo para mostrar al ganador del juego.
    * @param nombre Nombre del jugador ganador.
    */
  public static void mostrarGanador(String nombre) {
    System.out.println("\n\n%%%%%%%%%%%%%%%% " + nombre + " ha ganado! %%%%%%%%%%%%%%%%");
  }

  /**
    * Metodo para mostrar un mensaje de empate.
    */
  public static void mostrarEmpate() {
    System.out.println("\n\n%%%%%%%%%%%%%%%% Empate! %%%%%%%%%%%%%%%%");
  }

  /**
    * Metodo para mostrar el historial de partidas.
    * @param historial Lista de partidas en el historial.
    */
  public static void mostrarHistorial(ArrayList<Partida> historial) {
    System.out.println("%%%%%%%%%%%%%%%% Historial de partidas %%%%%%%%%%%%%%%%");
    System.out.println("Ganador\t\tTurno\t\tFecha");
    for (Partida partida : historial)
      System.out.println(partida.getNombre() + "\t\t" + partida.getTurno() + "\t\t" + partida.getFecha());
  }

  /**
    * Metodo para mostrar los archivos de configuracion disponibles.
    * @param archivos Lista de nombres de archivos de configuracion.
    */
  public static void mostrarArchivos(ArrayList<String> archivos) {
    System.out.println("%%%%%%%%%%%%%%%% Archivos de configuracion %%%%%%%%%%%%%%%%");
    for (int i = 0; i < archivos.size(); i++)
      System.out.println((i + 1) + ". " + archivos.get(i));
    System.out.println("Cualquier otro numero para volver al menu principal");
  }
}
```

14.8.12. Excepciones.java

```
package domino;

/**
 * Excepcion lanzada cuando se recibe una respuesta invalida.
 */
class RespuestaInvalidaException extends Exception {
  /**
    * Constructor de la excepcion.
    * @param mensaje Mensaje que describe la naturaleza de la respuesta invalida.
    */
  public RespuestaInvalidaException(String mensaje) {
    super(mensaje);
  }
}

/**
 * Excepcion lanzada cuando se recibe un indice de ficha invalido.
 */
class IndiceFichaInvalidoException extends Exception {
```

```
  /**
    * Constructor de la excepcion.
    * @param mensaje Mensaje que describe la naturaleza del indice de ficha invalido.
    */
  public IndiceFichaInvalidoException(String mensaje) {
    super(mensaje);
  }
}

/**
 * Excepcion lanzada cuando se recibe una orientacion invalida.
 */
class OrientacionInvalidaException extends Exception {
  /**
    * Constructor de la excepcion.
    * @param mensaje Mensaje que describe la naturaleza de la orientacion invalida.
    */
  public OrientacionInvalidaException(String mensaje) {
    super(mensaje);
  }
}
```

14.9. Pruebas de funcionamiento

Estando en la carpeta codigo y ejecutando el comando *java -jar domino/Domino.jar* se inicia el juego. Al iniciar el juego se muestra el menu de opciones.

```
%%%%%%%%%%%%%%%% Bienvenido a Domino %%%%%%%%%%%%%%%%
Escoja el modo de juego:
1. Jugador vs Bot
2. Bot vs Bot
3. Historial de partidas
4. Cargar partida anterior
5. Salir
$
```

Figura 19: Menu de opciones de juego

En caso de que se seleccione la opcion 1 se pide el nombre del jugador y se luego se pregunta si se quiere guardar la configuracion de la partida. En caso de que se seleccione la opcion 2 se inicia una partida entre dos bots. En caso de que se seleccione la opcion 3 se muestra el historial de partidas guardadas. En caso de que se seleccione la opcion 4 se muestra la lista de archivos de configuracion disponibles.

```
%%%%%%%%%%%%%%%% Ingrese su nombre %%%%%%%%%%%%%%%%
$ Alan

¿Quieres guardar la configuracion de la partida? (s/n)
$ s

Ingrese el nombre de la partida
$ Conf
```

Figura 20: Ingresar nombre de jugador y guardar configuracion

Al iniciar una partida se muestra la mesa, el pozo y las fichas del jugador. En caso de que sea el primer turno del jugador tira automaticamente la mula más alta o la ficha más alta.

```
Mesa actual:


Numero de fichas en el Pozo: 14

%%%%% Turno de Bot 1 %%%%%

Tus fichas:
  1       2       3       4       5       6       7
[2|4]   [0|0]   [1|5]   [6|6]   [0|3]   [2|5]   [4|4]

Bot 1 juega la mula mas alta: [6|6]
```

Figura 21: Inicio del juego

Los siguientes turnos son diferentes dependiendo del tipo de jugador. En caso de que sea un bot se espera 3 segundos antes de que juegue su ficha. En caso de que sea una persona se le pide que seleccione la ficha que quiere jugar y el lado donde la quiere jugar.

```
Mesa actual:
[2|2]

Numero de fichas en el Pozo: 14

%%%%% Turno de Alan %%%%%

Tus fichas:
  1       2       3       4       5       6       7
[0|0]   [2|6]   [2|4]   [1|3]   [1|4]   [0|4]   [0|5]

Que ficha deseas jugar? (1 - 7)
$ 2
Donde la quieres jugar? (i o d)
$ i

Alan juegas la ficha: 2 - [6|2]
```

Figura 22: Turno de jugador Persona

```
Mesa actual:
[6|2]   [2|2]   [2|0]   [0|4]

Numero de fichas en el Pozo: 14

%%%%% Turno de Bot %%%%%

Tus fichas:
  1       2       3       4       5
[0|1]   [1|5]   [0|3]   [3|6]   [5|6]

Bot juegas la ficha: 4 - [3|6]
```

Figura 23: Turno de jugador Bot

En caso de que un jugador necesite robar una ficha se muestra un mensaje indicandolo y se vuelven a imprimir sus fichas.

```
%%%%% Turno de Bot %%%%%

Tus fichas:
  1
[5|6]
No puedes jugar, robas una ficha

Tus fichas:
  1       2
[5|6]   [6|6]
```

Figura 24: Mensaje de robar ficha

Al ganar el jugador se muestra un mensaje de victoria y se termina el programa.

```
%%%%%%%%%%%%%%%% Alan ha ganado! %%%%%%%%%%%%%%%%


Presione enter para continuar...
```

Figura 25: Mensaje de victoria

Al abrir el historial de partidas se muestra una tabla con el nombre del ganador, el turno en el que gano y la fecha en la que se jugo la partida.

```
%%%%%%%%%%%%%%%% Historial de partidas %%%%%%%%%%%%%%%%
Ganador         Turno           Fecha
Bot 1           14              Tue Dec 12 16:25:41 CST 2023
Bot 2           30              Tue Dec 12 16:52:18 CST 2023
Empate          43              Tue Dec 12 16:57:21 CST 2023
Bot 2           18              Tue Dec 12 16:59:21 CST 2023
Bot 2           36              Tue Dec 12 17:00:18 CST 2023
Bot 1           30              Tue Dec 12 17:00:35 CST 2023
Bot 1           14              Tue Dec 12 17:00:56 CST 2023
Bot 2           33              Tue Dec 12 17:01:13 CST 2023
Bot 1           13              Tue Dec 12 17:02:52 CST 2023
Bot 1           27              Tue Dec 12 17:10:19 CST 2023
Bot 2           50              Tue Dec 12 17:11:18 CST 2023
Bot 1           28              Tue Dec 12 17:11:36 CST 2023
Bot 2           21              Tue Dec 12 17:12:47 CST 2023
Bot 2           13              Tue Dec 12 17:17:55 CST 2023
Bot 2           13              Tue Dec 12 17:19:18 CST 2023
Bot 1           14              Tue Dec 12 17:21:08 CST 2023
Bot 2           14              Tue Dec 12 17:21:40 CST 2023
Bot 1           21              Tue Dec 12 17:22:36 CST 2023
```

Figura 26: Historial de partidas

Si se quiere cargar una partida guardada se muestra la lista de archivos de configuracion disponibles y se pide que se seleccione el archivo de configuracion que se quiere cargar.

```
%%%%%%%%%%%%%%%% Archivos de configuracion %%%%%%%%%%%%%%%%
1. Test.txt
2. Test2.txt
3. Test3.txt
4. TestJar.txt
Cualquier otro numero para volver al menu principal

Ingrese el numero de la partida que desea cargar
$
```

Figura 27: Lista de archivos de configuracion

14.10. Documentacion de JavaDoc

Al ejecutar el comando *javadoc -d documentacion domino/*.java* se genera la documentacion de JavaDoc en la carpeta *documentacion*. Algunos ejemplos de la documentacion generada se muestran a continuación y lo restante se encuentra en la dirección https://dominov2.netlify.app.

Figura 28: Página principal

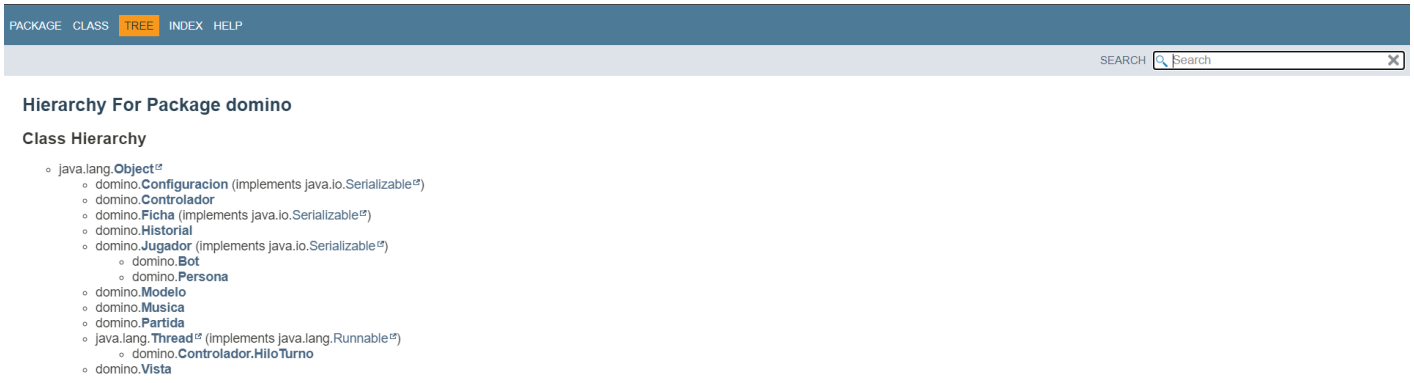

Figura 29: Árbol de jerarquia de clases

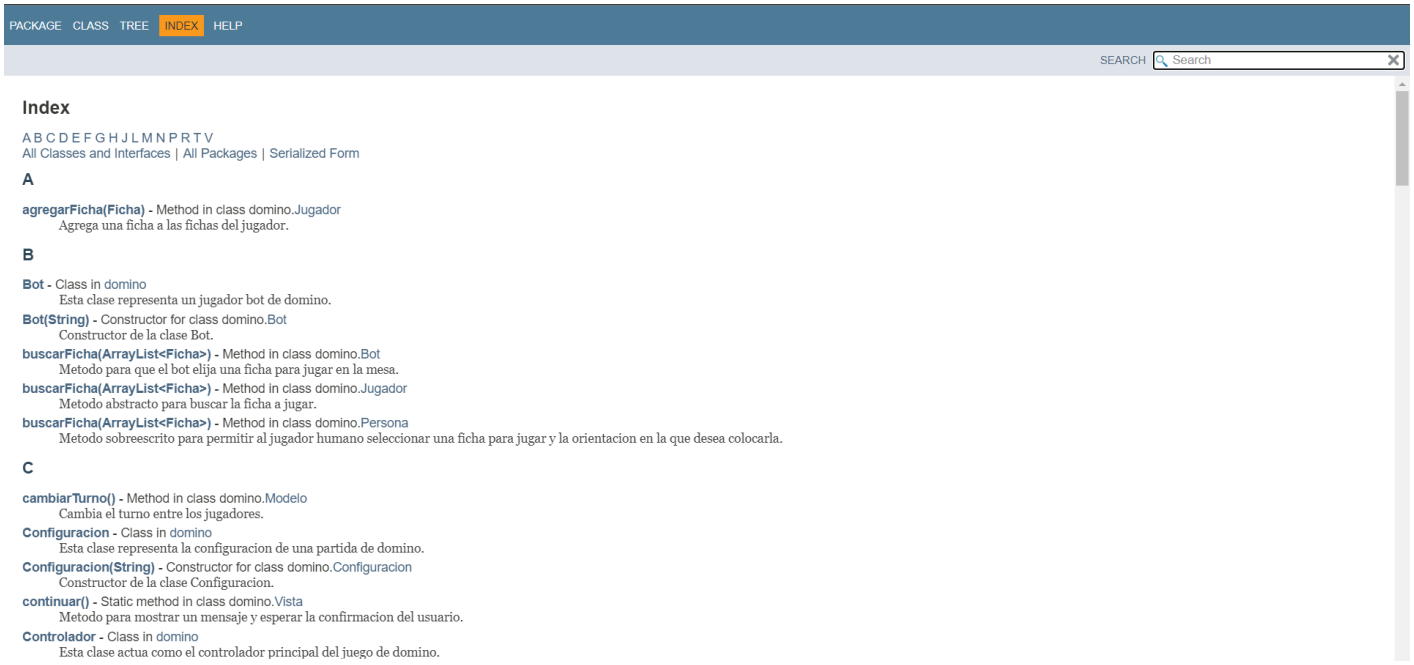

Figura 30: Índice alfabetico

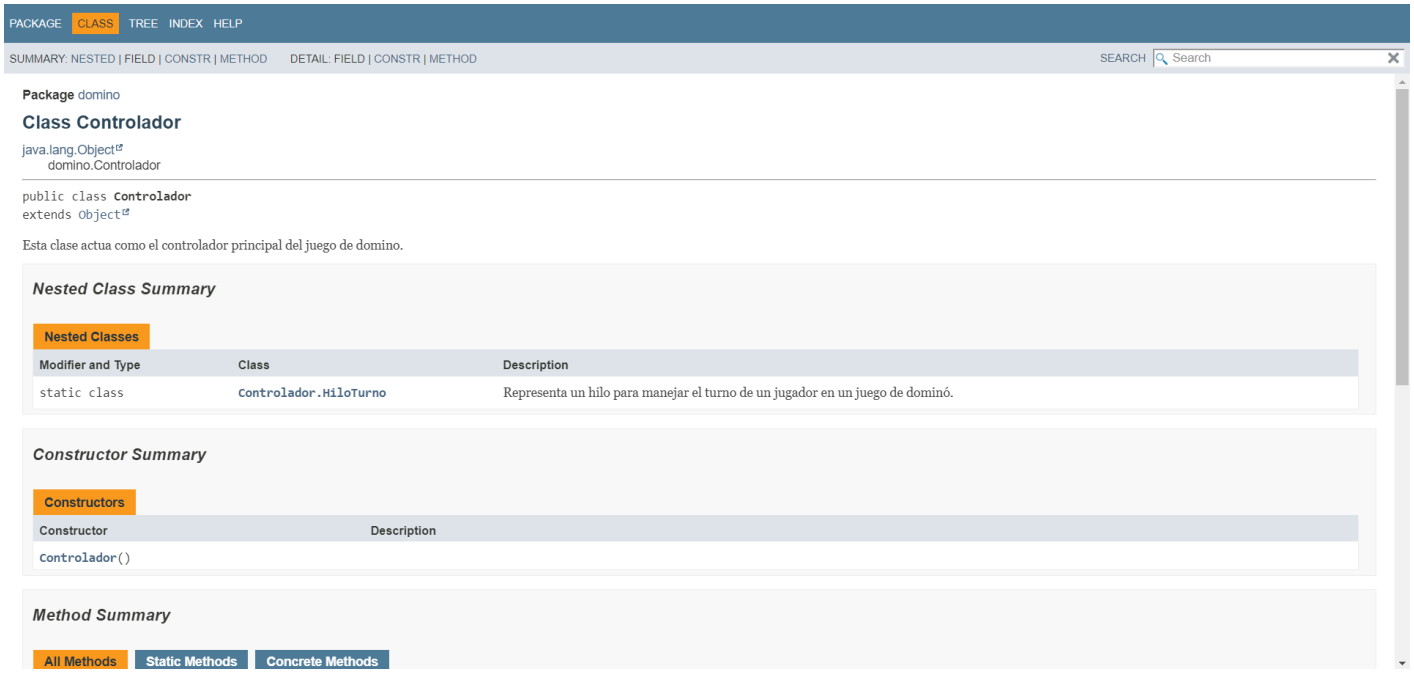

Figura 31: Página de la clase *Controlador*