

Diseño del Juego «Escaleras y Toboganes»

1. Características básicas del juego

Escaleras y Toboganes es un juego de mesa clásico de carreras cuya meta es ser el primer jugador en alcanzar la última casilla del tablero. Se juega típicamente entre **2 a 4 jugadores** (mínimo dos, aunque no existe un máximo estricto en la versión digital), usando un tablero numerado, fichas de colores y un dado de 6 caras. Cada jugador tiene una ficha que comienza en la casilla inicial (generalmente la número 1) y avanza según el valor obtenido al lanzar el dado en su turno. El tablero contiene varias *escaleras y toboganes* (también conocidos como serpientes o “chutes”) que conectan dos casillas: las escaleras permiten subir a una casilla más alta y los toboganes hacen descender a una casilla más baja. El objetivo es llegar a la casilla final (la de mayor número) antes que los demás, aprovechando las escaleras para progresar más rápido y evitando los toboganes que retrasan el avance.

Ejemplo de tablero físico del juego clásico, con casillas numeradas del 1 al 100, varias escaleras (de color verde) que adelantan al jugador y serpientes (toboganes, en rosa) que lo hacen retroceder. Cada jugador avanza su ficha según el dado; las fichas de distintos colores representan a los diferentes jugadores.

Componentes principales: Un tablero cuadrículado (usualmente de 100 casillas dispuestas en 10x10), un dado de seis caras (para obtener resultados de 1 a 6 en cada tiro) y fichas o avatares para representar a cada jugador. En la versión digital, el tablero y las fichas se representan gráficamente, y el dado es virtual (simulado por el software). Las escaleras y toboganes están predefinidos en posiciones fijas del tablero (p. ej., una escalera que inicia en la casilla 4 y sube hasta la 14, un tobogán de la 17 a la 7, etc.), proporcionando atajos o penalizaciones durante el recorrido. **Duración:** Al ser un juego principalmente de azar (alta dependencia de la suerte al lanzar el dado), la duración es variable pero típicamente ronda 15 a 30 minutos en su versión física. La versión digital puede incluir opciones de tiempo límite por turno para agilizar la partida. En cuanto a **edad y complejidad**, es un juego sencillo orientado a todo público (niños desde ~5 años en adelante), ya que no requiere habilidades especiales más allá de contar casillas.

2. Comportamiento del sistema desde la perspectiva del usuario

Desde el punto de vista del usuario, el sistema ofrece una experiencia completa que va desde el acceso (registro/login) hasta la finalización de una partida. A continuación se describen los **casos de uso principales** y el **viaje típico del usuario** a través de la aplicación:

- **Registro e inicio de sesión:** Un nuevo usuario puede registrarse proporcionando datos básicos (nombre de usuario, correo, contraseña, etc.), tras lo cual el sistema crea su cuenta. Un usuario existente inicia sesión con sus credenciales. (Un *usuario no registrado* puede tener acceso limitado, p. ej. jugar como invitado o solo visualizar las reglas, dependiendo de las políticas definidas). Este proceso garantiza la identificación del jugador y, en la versión académica, se puede implementar con autenticación basada en token JWT para las solicitudes posteriores.
- **Creación o unión a partida:** Una vez autenticado, el usuario accede al lobby de juego. Allí puede **crear una nueva partida** (convirtiéndose en anfitrión/administrador de la misma) o **unirse a una partida existente**. Si crea una partida, el sistema le asigna un identificador único o código de invitación para que otros jugadores se unan. Si desea unirse, puede elegir entre partidas listadas en estado “esperando jugadores” o ingresar un código de invitación. Este flujo corresponde a casos de uso como “*Crear partida*” y “*Unirse a partida*”. Al unirse, el jugador espera en la sala de pre-juego hasta que se alcance el número requerido de jugadores y el anfitrión inicie el juego.
- **Inicio de la partida:** Cuando la partida tiene los jugadores necesarios, el administrador la inicia. El sistema muestra el **tablero de juego** a todos los participantes, posicionando las fichas de cada jugador en la casilla inicial. A cada jugador se le asigna un color o avatar distinto para su ficha, y se determina aleatoriamente quién comienza el primer turno (salvo que se defina otra regla, como que el creador inicia). En este punto, todos los jugadores ven la misma interfaz de tablero y conocen el orden de turnos (el sistema podría mostrar una lista de turno actual y siguientes).
- **Desarrollo del juego (turnos y acciones del usuario):** El juego procede por turnos rotativos. En su turno, el **jugador activo** realiza la acción principal: lanzar el dado. En la interfaz, esto se traduce en hacer clic en un botón “Lanzar” (o “Roll”) para generar un número aleatorio de 1 a 6. El sistema anima el dado virtual y luego mueve automáticamente la ficha del jugador el número de casillas correspondientes. Si al detenerse cae en el inicio de una escalera o tobogán, el sistema muestra una animación de ascenso o descenso y actualiza la posición final de la ficha acorde a la regla (subir o bajar). Estas acciones son percibidas por todos los jugadores en tiempo real. Tras resolver el movimiento (y cualquier evento extra, como cartas sorpresa si aplican), el turno concluye y pasa al siguiente jugador. Durante su turno, el jugador también puede realizar acciones secundarias si están disponibles (por ejemplo, usar una carta sorpresa en mano, aunque en este juego las decisiones estratégicas son mínimas dado el factor azar). Los jugadores **fuera de turno** básicamente observan el estado, pueden consultar el historial de movimientos o chatear si la funcionalidad existe, pero no pueden mover fichas.
- **Interacciones adicionales en partida:** El sistema asegura que solo el jugador cuyo turno está activo pueda lanzar el dado (los botones de acciones aparecen habilitados solo para él, o la acción `rollDice` solo es aceptada desde su cliente). Puede haber retroalimentación visual como un indicador resaltando el jugador en

turno o un cronómetro regresivo si hay límite de tiempo para lanzar. Otros posibles casos de uso durante la partida incluyen “*Enviar mensaje de chat*” (si se implementa chat en la sala) o “*Abandonar partida*” (si un jugador se desconecta antes de terminar, el sistema podría reemplazarlo por un jugador virtual o marcar la partida como inconclusa dependiendo de las reglas definidas).

- **Finalización de la partida:** El juego termina cuando un jugador alcanza la última casilla del tablero (p. ej., casilla 100) y es declarado ganador. El sistema notifica a todos el resultado (mostrando, por ejemplo, un mensaje “**¡Jugador X ha ganado!**” con efectos de celebración). También puede mostrar la clasificación final (orden de llegada de los demás jugadores si se decide continuar hasta que todos lleguen, aunque en este juego usualmente termina al primer ganador). Tras esto, la partida se marca como finalizada en el servidor. Los usuarios pueden entonces optar por salir al menú principal (landing) o iniciar una nueva partida. Este flujo cubre casos de uso de “*Finalizar juego*” y “*Mostrar resultados*”. En la interfaz, podría presentarse una pantalla de fin de juego con el podio de ganadores y opciones para volver a la página de inicio o jugar otra vez.

En resumen, desde el login hasta la finalización, el usuario experimenta una **navegación fluida**: autenticación -> entrar al lobby -> unirse o crear partida -> jugar turnos con animaciones y actualizaciones en tiempo real -> ver el resultado final. Todo ello con retroalimentación constante de la interfaz (mensajes de error si unirse a una partida falla, avisos de eventos como “subiste por una escalera”, etc.). Además, el sistema debe contemplar estados intermedios o alternos, por ejemplo: si un jugador intenta unirse a una partida llena, se le informa; si el anfitrión abandona antes de iniciar, se elige un nuevo administrador o se cancela la partida; si un jugador se desconecta en mitad del juego, el sistema podría pausar la partida brevemente o continuar con un bot sustituto, según reglas definidas en el alcance del proyecto.

3. Reglas de negocio implementadas por el servidor

En el servidor del juego se definen las **reglas de negocio** que garantizan que la dinámica respete las normas de *Escaleras y Toboganes*. Algunas de las principales reglas y lógica que el servidor implementa son:

- **Validación de turno:** El servidor mantiene el orden de turnos y solo procesa acciones de juego (como tirar el dado) si provienen del jugador que tiene el turno actualmente. Si por alguna razón llegan peticiones fuera de turno (mal funcionamiento del cliente o intento malicioso), estas se rechazan. Por ejemplo, si el jugador 2 intenta lanzar el dado cuando es turno del jugador 1, el servidor ignora esa acción o devuelve un error. De esta forma se asegura la secuencialidad correcta de la partida.
- **Lanzamiento de dado y cálculo de movimiento:** Cuando el jugador en turno solicita lanzar el dado (vía mensaje `rollDice` del cliente), el servidor genera un número aleatorio entre 1 y 6 (simulando la tirada de dado). Con ese resultado,

calcula la nueva posición potencial de la ficha sumando el valor obtenido a la posición actual del jugador. **Ejemplo:** El jugador está en la casilla 27 y obtiene un 5, entonces su posición base candidata es 32. El servidor luego comprueba el tablero para determinar si esa casilla tiene una escalera, un tobogán u otro evento especial.

- **Aplicación de escaleras y toboganes:** Esta es una regla central: si el jugador cae exactamente en la casilla inferior de una escalera, **sube** automáticamente hasta la casilla superior conectada a esa escalera. Del mismo modo, si cae en la casilla donde inicia un tobogán, **desciende** hasta la casilla inferior indicada por dicho tobogán. El servidor, tras calcular la posición base por el dado, consulta una estructura de datos del tablero (p. ej., un mapa o array que asocia casillas de inicio de escalera/tobogán con su destino) para ver si debe modificar la posición. Continuando el ejemplo anterior, si la casilla 32 fuera el inicio de una escalera que va a 48, el servidor actualizaría la posición final del jugador a 48 y marcaría este evento en la respuesta. Estas transiciones se envían al cliente para animación y para registro en el historial. Si varias escaleras/toboganes se encadenaran (teóricamente podría ocurrir que una escalera lleve a otra, aunque usualmente los tableros reales no lo diseñan así), el servidor aplicaría todas las transiciones necesarias hasta llegar a una casilla sin evento.
- **Movimiento de ficha y actualización del estado del juego:** Después de determinar la posición final del jugador en su turno, el servidor actualiza el estado interno de la partida: la nueva posición de la ficha del jugador en la estructura de juego (p. ej., en el objeto Player o en una matriz de posiciones). También registra el movimiento en el historial de la partida (por ejemplo, podría guardar en una lista de log: "Jugador X tiró 5 y pasó de 27 a 48 (subió por una escalera)"). A continuación, verifica condiciones de fin de juego. La regla de victoria estándar es: *si un jugador llega a la última casilla del tablero*, el servidor lo identifica como ganador y marca la partida como terminada. En algunas variantes existe la regla de necesitar sacar el número exacto para ganar (por ejemplo, estar en la casilla 97 y necesitar exactamente un 3 para caer en 100, de lo contrario la ficha no se mueve o rebota hacia atrás). Si se implementa esa regla, el servidor la contempla: si el jugador saca más de lo necesario para llegar a la meta, o bien no avanza ese turno, o bien avanza hasta 100 y retrocede lo sobrante (según la variante escogida). Estas variantes deben definirse claramente; por simplicidad, podríamos omitir la necesidad de tiro exacto (permitiendo ganar con cualquier tiro que lleve al 100 o más).
- **Gestión de turnos y turno extra por eventos:** Por defecto, el turno avanza al siguiente jugador en orden. Sin embargo, se pueden implementar reglas adicionales típicas de juegos de mesa: por ejemplo, **turno extra si saca 6 en el dado** (regla común en muchos juegos: el jugador que obtiene un 6, puede volver a lanzar). Si se adopta, el servidor comprobaría el valor del dado; si es 6, no avanza el turno al siguiente jugador sino que permanece en el mismo jugador (salvo quizás imponer un límite de tres 6 consecutivos, como alguna regla hogareña que dice que *tres 6 seguidos hacen perder el turno*, aunque esto es opcional y no parte del juego clásico estándar). Otra posible regla: *caer en escalera otorga turno extra, caer en tobogán lo quita* (también variación no estándar). En general, estas reglas de bonificación/castigo de turno se pueden definir en el servidor y comunicarse a los

jugadores.

- **Validación de entrada y reglas de integridad:** El servidor también aplica reglas de negocio para validar los datos. Por ejemplo, al registrarse, comprobar que el nombre de usuario no esté duplicado, que las contraseñas cumplan criterios, etc. Al crear una partida, asegurarse que el nombre o código generado sea único. Al unirse a una partida, verificar que haya cupo (no exceder el número máximo de jugadores definido, e.g., 4 jugadores) y que la partida esté en estado correcto (no unirse a una partida ya iniciada o finalizada). Al procesar movimientos, garantizar que un jugador no pueda ocupar la misma casilla exacta que otro *si la regla lo prohíbe* – en el juego de escaleras y serpientes tradicional, dos fichas pueden coexistir en la misma casilla sin problema, así que el servidor permite posiciones iguales, pero esto es parte de las reglas (a diferencia de juegos como Ludo donde se pueden bloquear). Aquí asumiremos que compartir casilla es aceptable.
- **Actualización del estado para todos los clientes:** Cada vez que ocurre un cambio (movimiento de ficha, cambio de turno, alguien gana), el servidor envía mensajes de actualización a **todos** los clientes en la partida para que sus interfaces reflejen el nuevo estado. Por ejemplo, después de que el jugador A se mueve a 48, el servidor envía a todos los jugadores un mensaje indicando la nueva posición de la ficha de A y anunciando que ahora es el turno del siguiente jugador. Este concepto corresponde a la sincronización en tiempo real, posiblemente usando WebSockets u otra técnica.

En síntesis, el servidor actúa como árbitro que *valida acciones, aplica las reglas del juego* (movimientos y efectos de casillas), *mantiene el estado consistente* y *decide las condiciones de victoria*. Todo esto, implementado en lógica de backend (p. ej., en métodos de clases como Game, TurnManager, etc.) de manera que ningún cliente pueda romper las reglas a su favor.

4. Protocolo de comunicación preliminar cliente-servidor (JSON)

Para la interacción en tiempo real entre clientes (jugadores) y el servidor, se define un **protocolo de comunicación** basado en mensajes JSON intercambiados probablemente sobre WebSockets (para actualizar a todos los jugadores en tiempo real) o HTTP/REST en caso de acciones puntuales. Cada mensaje JSON incluye un campo que indica el tipo de acción o evento, y campos adicionales con los datos necesarios. La comunicación es esencialmente de tipo *request-response* en algunos casos (el cliente pide realizar una acción y el servidor responde con el resultado) y *broadcast* en otros (el servidor envía a todos los clientes notificaciones de lo ocurrido en el juego).

Se definen los principales tipos de mensajes (el formato exacto puede variar, pero a modo de ejemplo):

- **Login:** credenciales de usuario para autenticación. Ejemplo de petición: `{"tipo": "Login", "msg": { "user": "juan", "pass": "1234", "app": "EscalerasToboganes" }}` y posible respuesta: `{"tipo": "re_Login", "msg": { "token": "<token-autenticación>" }}` con un token para próximas comunicaciones
- **newUser (Registro):** contiene datos de registro (usuario, email, pass) y la respuesta del servidor indica éxito o error (por ejemplo, `{"tipo": "re_newUser", "msg": {"status": "OK"}}` o detalles del error).
- **createGame:** mensaje para crear una partida nueva. El cliente envía `{"action": "createGame", "msg": {"user": "juan", "token": "<token>"}}` y el servidor responde con algo como `{"action": "re_createGame", "msg": {"gameId": 123, "joinCode": "ABCD" }}`, proporcionando un ID o código de partida.
- **joinGame:** para unirse a una partida existente. Petición: `{"action": "joinGame", "msg": {"gameId": 123, "user": "pedro", "token": "<token>" }}`. Respuesta del servidor: `{"action": "re_joinGame", "msg": {"status": "OK", "players": [...], "board": {...} }}`, confirmando la unión y enviando la lista de jugadores actuales, estado del tablero, etc. (Si la unión falla, `status` podría ser "ERROR" con motivo).
- **startGame:** (opcional, o implícito cuando se alcanza número de jugadores) enviado por el admin para comenzar la partida. Informa a todos los jugadores con un mensaje broadcast `{"action": "gameStarted", "msg": {"gameId": 123, "order": ["juan", "pedro", ...], "board": {...}}}`.
- **rollDice (tirar dado):** este es crucial durante el juego. El cliente cuyo turno es envía, por ejemplo: `{"action": "rollDice", "msg": {"gameId": 123, "player": "juan", "token": "<token>"}}`. El servidor, al recibirlo, valida turno y genera el número. Luego responde a todos los jugadores (broadcast) con algo como: `{"action": "move", "msg": {"player": "juan", "dice": 5, "from": 27, "to": 48, "ladder": true, "nextPlayer": "pedro" }}`. Este mensaje indica que Juan sacó 5 y se movió de la casilla 27 a la 48 subiendo por una escalera, y que el siguiente turno es de Pedro. Cada cliente usaría esta info para animar el movimiento de la ficha de Juan y actualizar su interfaz (por ejemplo, mostrar un icono de escalera y resaltar que ahora juega Pedro).
- **jugada:** Alternativamente, el protocolo podría llamar al movimiento completo como `jugada`. En la tabla de ejemplo se ve un mensaje `jugada` desde cliente con atributos y su respuesta `re_jugada` desde. En nuestro caso lo detallamos con acciones atómicas (rollDice y su consecuencia).

- **chat:** si se implementa chat, mensajes como `{"action": "chat", "msg": {"user": "juan", "text": "¡Buena suerte!"}}` se distribuyen a todos en la partida, posiblemente con respuesta de confirmación.
- **listGames:** para listar partidas disponibles (por ejemplo en el lobby), cliente envía `{"action": "listGames"}` y servidor responde con listado de partidas no iniciadas o públicas.
- **leaveGame/disconnect:** para manejar jugadores que se salen. El cliente notifica `{"action": "leaveGame", "msg": {"gameId": 123, "user": "pedro"}}`, y el servidor responde confirmando y alertando a los demás jugadores de que pedro salió (posiblemente ajustando el estado del juego o terminándolo si era crítico).

En la documentación de referencia se establecen estos tipos de mensajes y sus atributos esperados. Por ejemplo, *joinGame* incluye id de partida, usuario y token; *jugada* incluye la acción del juego; *error* y *chat* llevan mensajes de error o chat respectivamente. Cada petición del cliente suele tener su contraparte de respuesta del servidor (*re_**), como se observa en la tabla: *re_joinPartida*, *re_jugada*, *re_chat*, etc., con los datos o confirmaciones correspondientes.

Es importante notar que durante la partida activa, muchas comunicaciones serán iniciadas por el servidor (no solo respuestas). Cuando un jugador lanza el dado, el servidor envía a *todos* el resultado; cuando alguien gana, el servidor envía un mensaje `{"action": "gameOver", "msg": {"winner": "juan"}}` a todos, etc. Para facilitar esto, lo ideal es usar WebSockets con un canal por partida donde servidor y clientes intercambian JSON de forma asíncrona.

Todo el protocolo está pensado para ser **sin estado en cada mensaje** más allá del token de sesión y el contexto de partida, ya que el estado de juego se mantiene en el servidor. De esta manera, si un cliente se reconecta, podría recibir el estado completo nuevamente (mediante mensajes de sincronización). El formato JSON hace que los mensajes sean legibles y fáciles de depurar, y la estructura tipo `{ tipo: "...", msg: { ... } }` separa el tipo de evento de los datos asociados, lo cual coincide con buenas prácticas en este tipo de aplicaciones.

Ejemplo concreto (rollDice):

json

CopiarEditar

```
// Petición del cliente (Jugador 1 lanza dado)
{
  "action": "rollDice",
  "gameId": 123,
  "player": "Jugador1",
  "token": "abcd1234"
```

```

}

// Respuesta del servidor (difundida a todos los jugadores)
{
  "action": "move",
  "gameId": 123,
  "msg": {
    "player": "Jugador1",
    "dice": 5,
    "from": 27,
    "to": 48,
    "event": "ladder",      // escalera (podría ser "slide" para
                             tobogán, o null si nada)
    "nextPlayer": "Jugador2"
  }
}

```

En este ejemplo, el servidor informa que Jugador1 sacó 5 y se movió de 27 a 48 gracias a una escalera, y que Jugador2 tiene el siguiente turno. Este tipo de mensajes JSON cubre las necesidades del juego en red: control de turnos, movimientos y eventos especiales, actualizaciones de estado y chat. En la implementación real, se definiría con más detalle el esquema JSON y se podrían usar librerías de serialización/deserialización en el servidor para mapear estos mensajes a métodos (por ejemplo, un mensaje `rollDice` invoca al método correspondiente en el controlador del juego).

5. Diseño de la página *Landing* (inicio)

La **landing page** o página de inicio es la primera vista con la que interactúa el usuario al acceder a la aplicación web del juego. Su diseño debe ser claro, atractivo y brindar acceso a las principales secciones: información general, reglas, y el ingreso al juego (login/registro).

En esta landing page se suelen incluir las siguientes **secciones** y elementos:

- **Encabezado (Header):** En la parte superior, un título llamativo con el nombre del juego “*Escaleras y Toboganes*”. Puede acompañarse de un pequeño eslogan (ej. “¡Llega a la cima evitando los toboganes!”) y eventualmente el logo de la universidad o del proyecto. También en el header o navbar se incluirían botones de navegación rápida, como “Inicio”, “Acerca de”, “Reglas” y “Entrar”.
- **Sección "Acerca de":** Un bloque que describe brevemente el juego y el proyecto. Aquí se explica qué es Escaleras y Toboganes, quizás su origen clásico y el objetivo del juego, adaptado al contexto digital. Podría mencionar que es un proyecto académico de la PUC Chile, el propósito educativo, etc. Esta sección orienta al

usuario nuevo sobre de qué trata el sitio.

- **Sección "Reglas del juego":** Un resumen de las reglas principales (similar a la sección 1 de este documento pero más breve y orientado al jugador). Incluir puntos clave: cómo se juega, qué hacen las escaleras y toboganes, cómo ganar. Esta sección permite que alguien que no conozca el juego pueda aprender lo básico sin leer un documento aparte. Podría mostrarse con texto y pequeños íconos (por ejemplo, un icono de escalera y uno de tobogán junto a la explicación de sus efectos).
- **Botones de acceso (Login/Registro):** Destacados en la página, probablemente dos botones: "Iniciar Sesión" y "Registrarse". Estos botones llevan a los formularios correspondientes de login o alta de usuario. Alternativamente, si el usuario ya tiene sesión iniciada (cookie/token activa), la landing podría detectarlo y en lugar de esos botones mostrar directamente un botón "Ir al Lobby / Mis Partidas".
- **Opción de juego rápido/invitado:** Dependiendo de los requisitos, se podría ofrecer un botón "Jugar como Invitado" que permita probar el juego sin registro (generalmente entraría al lobby con un nombre temporal "Invitado#" y funcionalidades limitadas). Si se decide incluir, estaría también visible en la landing.
- **Diseño visual:** La landing debe ser **responsiva** y agradable. Podría tener un fondo temático (por ejemplo, una ilustración del tablero o dibujos de escaleras y toboganes de manera difuminada). Se pueden usar colores vivos asociados al juego (colores primarios intensos, similares a los del tablero físico: rojo, azul, verde, amarillo) de forma equilibrada. Importante mantener buen contraste para la legibilidad.

La composición general podría ser, por ejemplo, un diseño de una sola página estilo *scroll*: primero un banner superior con el logo y los botones de login/registro, más abajo las secciones "Acerca de" y "Reglas" en columnas o tarjetas. También podría tener una imagen ilustrativa del tablero incrustada. En la parte inferior, un pie de página simple con créditos del proyecto (curso, universidad, año, nombres de integrantes).

En cuanto a la **usabilidad**, los botones de "Entrar" y "Registro" deben destacar (posiblemente de color diferente, p. ej., un botón de color verde "Jugar ahora" que motive a la acción). La navegación "Acerca" y "Reglas" puede ser anclas que hagan scroll a esas secciones en la misma página.

Para el diseño detallado de esta interfaz, se pueden utilizar herramientas de prototipado como *Balsamiq* (para un wireframe de baja fidelidad) o *Figma* / *Adobe XD* (para un diseño de alta fidelidad con el estilo final). Estas herramientas permiten planificar la ubicación de cada elemento antes de implementarlo. En la documentación, se podría incluir un mockup hecho con Balsamiq que muestre la estructura: header con título, columna izquierda "Acerca de" con texto, columna derecha "Reglas" con una lista de reglas, y centrado abajo los botones de acción.

En resumen, la landing page da la **primera impresión** del juego: transmite la temática lúdica y guía al usuario a registrarse o iniciar sesión rápidamente para empezar a jugar. Un usuario no registrado puede leer de qué trata el juego y decidir crear una cuenta para unirse a la diversión.

6. Mockup de pantalla de registro de usuario

La pantalla o formulario de **registro de usuario** permite a un nuevo jugador crear una cuenta en la plataforma. El diseño de este formulario debe ser sencillo, claro en los requisitos de cada campo, y con validaciones para evitar datos incorrectos.

Campos típicos del formulario de registro:

- **Nombre de usuario:** Un campo para elegir un alias único. Se validará que no esté ya en uso (consulta al servidor en vivo o al enviar el formulario) y quizás restricciones de longitud (por ejemplo 3 a 15 caracteres) y caracteres permitidos (solo letras/números, sin espacios, etc.).
- **Correo electrónico:** Usado para identificación única y eventualmente recuperación de contraseña. Validación de formato (`usuario@dominio.com`) al momento de introducir (indicando al usuario si el formato no es válido).
- **Contraseña:** Campo de contraseña, idealmente con criterios de seguridad (mínimo 6 u 8 caracteres, incluir mayúsculas, números, etc. según políticas). Incluir confirmación de contraseña (un segundo campo para reingresarla y verificar que coincidan).
- **Otros campos opcionales:** Podría pedirse el nombre real, pero en un juego no es indispensable. Quizá un avatar o selección de color preferido (que podría usarse para su ficha en el juego), aunque esto podría hacerse después del registro también.

El formulario estaría dispuesto con etiquetas claras (“Nombre de usuario:”, “Email:”, “Contraseña:”, “Confirmar Contraseña:”) y campos de entrada alineados. Debajo o al lado de cada campo, se podrían mostrar *mensajes de validación* en tiempo real. Por ejemplo: si el usuario teclea un nombre de usuario ya existente, aparece un mensaje “⚠ Nombre de usuario no disponible”; o si la contraseña y su confirmación no coinciden, aparece “⚠ Las contraseñas no coinciden”. Estos mensajes ayudan al usuario a corregir antes de enviar.

Botones: Al final del formulario habrá un botón destacado “Registrarse” para enviar. También un enlace o botón secundario “¿Ya tienes cuenta? Inicia sesión” que lleve al formulario de login (facilitando la navegación en caso de estar en la pantalla equivocada). Inversamente, en la pantalla de login habrá un enlace “Crear nueva cuenta” para llegar a este registro.

Flujo de confirmación: Tras llenar y enviar, el sistema mostrará feedback. Si todo está correcto, se podría redirigir automáticamente al lobby (y considerar al usuario logueado), o bien mostrar un mensaje tipo “Registro exitoso, ahora puedes iniciar sesión”. En algunos

sistemas, envían un correo de confirmación, pero quizá para este proyecto se omita por simplicidad. Si hay errores (por ejemplo email ya registrado), el sistema retornará al formulario con un mensaje de error apropiado en la parte superior o junto al campo conflictivo.

Desde el punto de vista de interfaz, se busca que el proceso sea **ágil y claro**: pocos campos, solo lo necesario, y validaciones inmediatas. Un mockup hecho con *Figma* o *Adobe XD* podría ilustrar esta pantalla. Por ejemplo, un diseño limpio con fondo claro, un recuadro central con el título “Crear Cuenta”, los campos en una columna, y el botón “Registrarse” en color azul o verde. Los placeholders dentro de los campos podrían guiar sobre el formato (“ejemplo@correo.com” en el email, etc.).

Adicionalmente, se pueden implementar ayudas como: mostrar u ocultar la contraseña (un icono de ojo tachado para mostrar la contraseña en texto plano momentáneamente si el usuario quiere verificar lo que escribió), o una barra de “fortaleza de contraseña” debajo del campo de contraseña indicando si es débil, media o fuerte.

Desde el punto de vista del sistema, al enviarse este formulario, en el backend se crea el nuevo registro de usuario en la base de datos (tabla Users en PostgreSQL). Por eso es importante validar bien antes de enviarlo, para minimizar rechazos. Si el registro es exitoso, el usuario quizás inicia sesión automáticamente (auto-login) y pasa directamente al menú del juego.

En resumen, el mockup de registro presenta una **UX intuitiva**: solicita información mínima y guía al usuario con mensajes claros. Herramientas de diseño de interfaz como *Justinmind* o *Balsamiq* pueden emplearse para iterar sobre este formulario y probar la disposición de los elementos antes de codificarlo.

7. Lógica de inscripción o unión a partida (Lobby de juego)

Una vez el usuario ha iniciado sesión, entra al **lobby** o menú principal de juego, donde puede inscribirse en una partida existente o crear una nueva. Aquí se manejan roles y tiempos de espera, así como la distinción de privilegios del administrador de la partida.

Crear partida (rol Administrador): Cuando un usuario decide crear una partida, invoca la función correspondiente (por ej. clic en “Crear partida nueva”). El servidor crea un objeto Partida (Game) con un ID único y lo asocia al usuario como administrador. El administrador puede tener capacidades especiales, como iniciar la partida manualmente cuando estén listos o expulsar jugadores problemáticos antes de empezar. Al crear la partida, ésta se pone en estado “pendiente” o “en espera” de jugadores. El sistema podría proveer al admin un código de invitación o URL corta para compartir con amigos, o enlistar la partida en una sala pública para que otros la vean.

Unirse a partida (rol Jugador común): Un usuario puede unirse introduciendo un código o seleccionando una partida listada. Al unirse, el servidor verifica que haya cupo disponible. Si todo ok, añade al usuario a la lista de jugadores de esa partida. El cliente del jugador entonces cambia su vista al *sala de espera* de esa partida, mostrando quiénes están

conectados. Cada jugador recién unido recibe la información actual (quién es el admin, cuántos jugadores hay / faltan, etc.).

Número de jugadores y comienzo: Se define un número mínimo y máximo de jugadores por partida (por ejemplo, mínimo 2, máximo 4 jugadores, en concordancia con el juego clásico). La partida no inicia hasta que al menos el mínimo esté presente. El máximo impide que se unan más de los permitidos. El lobby podría mostrar “Jugadores: 3/4” indicando vacantes. Una vez alcanzado el mínimo, si el admin desea puede arrancar inmediatamente o esperar a llenar el cupo. También podría haber un temporizador de cuenta regresiva para inicio automático (p.ej., “La partida comienza en 30 segundos...” una vez hay 2 jugadores, dando chance a más gente de unirse).

Manejo de timeout (tiempos de espera): Para no dejar partidas colgadas indefinidamente, se implementan algunas lógicas de timeout:

- Si alguien crea una partida y ningún segundo jugador se une en un período determinado (por ejemplo 2 minutos), la partida podría *cancelarse automáticamente* y notificar al admin que no hubo jugadores (devolviéndolo al lobby general). Esto evita saturar el sistema con partidas vacías.
- Durante la partida, también pueden haber timeouts por inactividad de un jugador en su turno (por ejemplo, máximo 60 segundos para lanzar el dado). Si se agota, el servidor puede forzar la acción (p. ej., tirar el dado automáticamente por él para no frenar el juego). Esta regla de inactividad garantiza el flujo continuo.
- En la sala de espera previa al inicio, se podría imponer un límite de espera. Por ejemplo, si tras X minutos no se completa el mínimo, la partida se disuelve. Sin embargo, dado que mínimo 2 para jugar, bastaría con el creador y uno más para iniciar manualmente.

Diferenciación de roles: El *administrador* (host) de la partida suele estar identificado en la UI de la sala de espera con algún icono o marca (por ejemplo, una estrella junto a su nombre). Solo él ve ciertos controles especiales: un botón “Iniciar partida” (enabled cuando hay mínimo de jugadores), y quizás la opción de expulsar a algún jugador antes de empezar (click en el nombre de un jugador -> “Expulsar”, útil si alguien desconocido se une a partida privada y no era deseado). Los *jugadores* normales no tienen esos controles; simplemente esperan y quizá pueden indicar que están listos con algún botón “¡Listo!” (aunque no es imprescindible en este juego).

Confirmación de inicio: Cuando el admin inicia la partida, todos los jugadores reciben una notificación y entran a la vista del tablero. En ese momento la inscripción se cierra (nadie más puede unirse) y la lista de jugadores queda fijada. El servidor asigna aleatoriamente el orden de turnos si no se hizo antes.

Abandonos antes de iniciar: Si un jugador que se había unido decide salir antes de que arranque el juego, puede hacerlo (un botón “Salir de la partida”). Entonces el servidor lo remueve de la lista y notifica a los demás (“Jugador X ha salido de la sala”). Si el que se va era el único o era el administrador, habría que manejarlo:

- Si el admin sale *antes* de iniciar, podría transferirse el rol admin a otro que quede (por ejemplo, al siguiente jugador que entró) o simplemente cancelarse la partida (dado que sin admin no puede empezar, quizá se decide anular la partida y avisar a todos).
- Si un jugador no-admin sale, el admin puede continuar esperando otros o iniciar con menos jugadores de los planificados.

Abandonos durante la partida: (Aunque esto se sale un poco de “inscripción”, es parte de la lógica del lobby/partida). Si alguien se desconecta en medio del juego, se puede manejar de varias formas según la especificación del proyecto: o bien se pausa y termina la partida (declarándola inconclusa), o bien el jugador queda “inactivo” y su ficha permanece pero ya no juega (los demás pueden seguir, ignorando al ausente), o incluso se podría implementar un bot que lance el dado por él para terminar la partida. Esto debe definirse; quizá en un contexto académico simple, se notifica que “Jugador X se ha desconectado” y se continúa con los restantes, o se finaliza la partida abruptamente si era un juego estricto.

Toda esta lógica de lobby e inicio de partida puede resumirse en un **diagrama de estados**: *Sala pública* (viendo lista de partidas) -> *Sala de espera de partida (estado pendiente)* -> *Partida en juego (estado en curso)* -> *Partida finalizada*. El usuario atraviesa esos estados según sus acciones.

En implementación, se suele usar una vista tipo **lista/tabla** de partidas en el lobby. Cada entrada muestra el nombre o ID de la partida, número de jugadores dentro vs máximo, y un botón “Unirse”. También un botón “Crear nueva partida”. El **uso de frameworks** en frontend (como React/Vue) podría facilitar la actualización en tiempo real de esa lista a medida que se crean o inician partidas (por WebSocket o polling).

Resumiendo: la inscripción a partidas distingue al creador (admin) y a los participantes, maneja los tiempos para que el juego arranque oportunamente y asegura roles claros. Esta parte del sistema garantiza que las partidas comiencen de forma organizada con los jugadores correctos y las expectativas alineadas.

8. Navegación e interacción del usuario (menú, botones, hot-keys, drag-and-drop)

La aplicación proporciona **mecanismos de navegación** intuitivos y soporta diversas interacciones para mejorar la experiencia de usuario durante el juego.

Menú principal y navegación: Arriba se suele tener un menú (navbar) accesible en todas las pantallas principales. Este menú podría incluir:

- El logo/título del juego que al clicarlo lleva a la página de inicio (landing).
- En sesiones autenticadas, opciones como “Lobby/Partidas” (vista principal de juego), “Perfil” (si hubiera ajustes de usuario) y “Cerrar Sesión”.

- En la pantalla de juego, quizás un botón de menú o icono (ej. ☰ “hamburger menu”) para desplegar opciones como “Abandonar partida” o “Configuración”.

La navegación entre secciones (landing, reglas, juego) debe ser consistente. Por ejemplo, tras login exitoso se redirige automáticamente al lobby; desde el lobby se puede volver al inicio con un enlace; etc. Se debe evitar que el usuario se quede “atrapado” en una pantalla sin salida.

Botones de acción: En la interfaz gráfica se identificarán claramente los botones que ejecutan acciones:

- Botón **“Lanzar Dado”**: Visible solo para el jugador en turno en la partida. Generalmente es un botón grande y destacado (p.ej., color azul con la etiqueta “¡Lanzar!”) ubicado en la parte inferior de la pantalla de juego. Cuando se pulsa, se deshabilita hasta que la animación termina y el servidor valida, para evitar doble envíos.
- Botón **“Terminar Turno”**: Si hubiera acciones opcionales que un jugador pudiera hacer antes de ceder el turno (en este juego no muchas, salvo usar carta sorpresa), podría haber un botón para terminar su turno voluntariamente. Pero en Escaleras y Toboganes tradicional, el turno termina automáticamente tras mover.
- **Botones de menú contextual**: Por ejemplo, un botón de **“Reglas”** podría estar siempre disponible para consultar durante el juego (abrir un pop-up con las reglas resumidas).
- Botones de **confirmación/cancelación**: En ciertos diálogos (ej. al abandonar partida, “¿Seguro que deseas salir?” con [Sí] [No]).

Todos los botones importantes deben tener atajos de teclado (hot-keys) para usuarios avanzados:

- **Hot-keys**: Se pueden definir teclas rápidas: por ejemplo, tecla **R** para “Roll” (lanzar dado) cuando sea tu turno, así no tener que clicar el botón. Tecla **C** para abrir/close chat, etc. En la implementación, se registran eventos de teclado y se mapean a acciones cuando corresponda. Esto mejora accesibilidad también.
- Durante la navegación web normal, se pueden aprovechar atajos del navegador (aunque no siempre recomendable override). Un hotkey útil puede ser Esc para cerrar diálogos.

Drag-and-Drop (arrastrar y soltar): En la interacción del tablero, aunque el movimiento de fichas es automático, podríamos permitir *drag & drop* de la ficha como una forma manual de moverla tras tirar el dado, solo por sensación táctil. Por ejemplo, tras obtener el número, el usuario podría arrastrar su ficha hasta la casilla destino (el sistema validaría que sea la correcta calculada). Sin embargo, esto podría generar inconsistencias si el jugador suelta en

casilla errónea, así que normalmente se evitan errores moviendo automáticamente. Alternativamente, el drag-and-drop se puede usar en otros aspectos:

- Gestión de cartas sorpresa: si un jugador tiene cartas en mano (como power-ups), podría arrastrar una carta desde su mano hasta el tablero o hasta un ícono de descarte para usarla.
- En la interfaz de configuración del perfil: arrastrar un avatar a cierto lugar, etc., pero es marginal.

Otro lugar donde *drag-and-drop* es relevante es en el **tablero Kanban (Trello)** del equipo de desarrollo (como se menciona en el punto 14). En Trello, las tareas se arrastran de una columna a otra. Esto no es para el jugador del juego sino para los desarrolladores, pero vale la pena mencionar que el equipo usó drag-and-drop ahí.

Interacciones adicionales en UI:

- *Hover tooltips*: Al pasar el mouse sobre ciertos elementos, aparece información. Por ejemplo, sobre la imagen de una escalera en el tablero se podría mostrar “Escalera: te lleva arriba”, o sobre un jugador en la lista “Jugador X (puntaje actual Y)”. Estas ayudan a la comprensión sin saturar la pantalla de texto permanente.
- *Indicadores visuales*: El jugador en turno puede tener un indicador flotante (como una mano animada apuntando su ficha, o un borde iluminado en su avatar). Esto guía a todos sobre quién debe jugar.
- *Sonidos*: Es parte de la interacción (aunque no visual, sí de UX). Sonidos de dado rodando, de escalera (un sonido ascendente) o de tobogán (sonido descendente) aportan inmersión. Se permitiría mutear desde menú si se implementa.

Pantallas clave y mockups:

- *Lobby/Lista de partidas*: mostrada post-login, con lista y botones crear/unir. Interacción: clic en partida -> unirse.
- *Sala de espera*: lista de jugadores en esa partida, con indicador de quién es admin. Botón de iniciar para admin.
- *Pantalla de juego principal*: el tablero ocupa la mayor parte. A un lado o arriba, un **panel de información**: muestra todos los jugadores con sus nombres, posiblemente sus posiciones actuales (“Juan: casilla 48”) y algún indicador de turno. También puede mostrar un registro breve de últimos movimientos (“Juan subió a 48 por escalera”).
- *Historial detallado*: quizá un botón que expande el historial completo de movimientos en una barra lateral o ventana modal.

- *Chat*: si hay chat, un panel o ventana en que los jugadores escriben mensajes durante la partida, con scroll.
- *Pantalla de final de juego*: mostrando el ganador. Un botón “Salir” y otro “Jugar otra vez” (que llevaría al lobby o reinicia una nueva partida con mismos jugadores si se desea).

Para ilustrar estas interacciones, se pueden preparar **mockups** de las pantallas clave usando herramientas de diseño UI como *Lucidchart* (para flujos) y *Adobe XD* o *Figma* (para pantallas). Por ejemplo, un mockup del tablero podría mostrar el botón “ROLL!” (Lanzar) centrado abajo, las fichas de colores en la casilla 1 al inicio, y en la esquina superior un ícono de menú (☰).

En general, la aplicación debe responder a eventos del usuario de forma clara: cuando algo es clickable, destacarlo (cursor pointer, efecto hover). Si una acción está inhabilitada (botón gris con tooltip “Espera tu turno”), el usuario lo entiende. Los *hot-keys* y *drag-and-drop* complementan la experiencia para usuarios avanzados o en desktop, mientras que en móvil todo debería ser táctil (botones suficientemente grandes para tocar, y en vez de hover, se usarían quizás iconos siempre visibles por la falta de hover en táctil).

La **consistencia** es clave: mismos colores para acciones similares, misma ubicación de menús en todas las vistas, etc., de modo que el usuario navegue sin confundirse. Todos estos elementos se reflejan en los prototipos de alta fidelidad que guían la implementación final.

9. Mockups del tablero de juego (vista pública/privada, turnos, historial, avance)



El **tablero de juego** es la vista central durante la partida. Debe presentar la información de forma accesible tanto a los jugadores como a potenciales espectadores (si se permitiera un modo espectador, aunque no se ha mencionado, lo tratamos como “pública” la vista común del tablero).

Interfaz ilustrativa de la partida en curso: se muestra el tablero numerado con gráficos de serpientes (toboganes) y escaleras. Cada jugador está representado por un avatar sobre la casilla correspondiente (en la imagen, se ven fichas con caritas de colores). En la parte superior se indica la posición (POS) relativa: quién va 1ro, 2do, etc., y en la parte inferior están los controles del jugador: a la izquierda un menú (icono de 3 líneas) para opciones, y al centro el botón “ROLL!” para lanzar el dado. Esta UI es responsiva y mantiene a todos los jugadores informados del progreso

Vista pública vs privada: En un juego como Escaleras y Toboganes, prácticamente toda la información es pública, ya que no hay elementos ocultos (no hay cartas en mano ni información secreta de jugadores). Por ello, la **vista pública** del tablero (lo que todos ven) contiene todo: posiciones de todos los peones, resultado de dados tirados, etc. ¿Qué sería “vista privada”? Podría referirse simplemente a elementos UI que solo el jugador activo ve o

controla (por ejemplo, solo el jugador actual ve habilitado el botón de lanzar dado, los demás lo ven deshabilitado). Si se introdujeran *cartas sorpresa*, esas sí serían información privada: cada jugador tendría en su interfaz una mano de cartas que los demás no ven. En tal caso, la vista privada de cada uno incluiría, digamos, 2 cartas con efectos desconocidos para otros hasta que se jueguen. Aparte de eso, no hay distinción mayor: todos comparten casi la misma vista del tablero.

Turnos indicados en la interfaz: Es fundamental mostrar de forma destacada de quién es el turno actual y quién sigue. Esto puede hacerse de varias formas no excluyentes:

- Resaltando la ficha o el nombre del jugador actual (por ejemplo, la ficha podría brillar, o el nombre del jugador en un panel lateral aparece subrayado con “ Es tu turno!”).
- Mostrando una etiqueta “Turno de: Juan” en algún lugar fijo.
- Ordenando la lista de jugadores según turno: p. ej., en un panel lateral listar “Turno: Juan () | Siguiente: Pedro” etc.
- Sonidos o animaciones: quizá un pequeño resplandor alrededor del avatar del jugador en turno.

Además, cuando alguien lanza el dado, se puede temporalmente mostrar un **dado girando** en pantalla y luego el número obtenido grande en el centro o cerca del jugador, para que todos lo vean claramente. Esto acompañado de texto en el historial: “Juan lanzó 5”.

Historial de acciones: Un elemento importante es registrar y mostrar las acciones ocurridas en la partida (para referencia y para espectadores):

- Puede implementarse un **panel de historial** tipo log de texto. Por ejemplo, en un recuadro a la derecha se van agregando líneas: “Juan sacó 5 y subió de 27 a 48 (escalera)”, siguiente línea: “Pedro sacó 2 y bajó de 31 a 14 (tobogán)”, etc. Este historial podría limitarse a últimos X eventos visibles, con scroll para ver más.
- Otra opción visual es un **timeline** o registro por turnos: enumerar Turno 1: Juan -> movió a 48, Turno 2: Pedro -> movió a 14, etc.
- Si la interfaz está muy recargada, se puede ocultar el historial completo detrás de un botón “Ver historial” que abra un pop-up con la lista completa de movimientos.
- Este registro también es útil para depuración y fairness, para que si alguien duda de qué pasó en un turno anterior, pueda revisarlo.

Indicador de avance/progreso: Aunque en un tablero se ve quién está más adelante (por el número de casilla de cada ficha), a veces es útil un indicador global. Podría ser algo como una *barra de progreso* por jugador: cada jugador tiene una barrita que va del inicio a meta y su porcentaje avanza según su casilla. O más simple, como se ve en la imagen de

ejemplo, un display de posiciones “1st, 2nd” con los avatares【40†】. En la captura, “POS 1st (ficha roja), 2nd (ficha azul)” indica quién lidera. Esto puede recalcularse dinámicamente tras cada turno. Sin embargo, en Escaleras y Toboganes las posiciones pueden ser engañosas (alguien atrás podría ganar de repente con una escalera larga), pero igualmente sirve de referencia en cada momento.

Diseño del tablero en pantalla:

- El tablero ocupa la mayor parte de la ventana, preferentemente cuadrado para mantener proporciones. Las casillas deben ser lo suficientemente grandes para contener una ficha (y en caso de coincidir dos fichas en la misma, que se vean ambas quizás superpuestas ligeramente).
- Las escaleras y toboganes pueden representarse con imágenes (escalera dibujada conectando dos casillas, tobogán/serpiente dibujado entre otras dos). Estas imágenes no deben tapar los números de casilla; generalmente se ponen semi-transparente debajo o con colores que contrasten.
- Cada casilla está numerada; los números pueden mostrarse en la esquina de la casilla para que se lean. Alternativamente, podría haber una vista opcional “mostrar números” que el jugador active si necesita, para no sobrecargar visualmente (pero dado que es digital, mostrar los números está bien).
- **Fichas/avatares:** Pueden usar pequeñas figuras (por ejemplo, círculos de color sólido, o iconos personalizados). En digital, podríamos incluso usar mini-avatares con imagen (la cara del jugador si subió foto, o un emoji). Importante que se distingan claramente. Si dos fichas caen en la misma casilla, pueden mostrarse ligeramente offset (desplazadas) para que ambas sean visibles, o una encima de otra con borde distinto.
- **Animaciones:** Cada movimiento de ficha debe ser animado: la ficha recorre las casillas intermedias rápidamente para dar la sensación de movimiento (como si contara 1,2,3,...). Luego, si hay escalera, animar la ficha subiendo por la escalera dibujada; si hay tobogán, animar deslizándose hacia abajo. Esto mejora la comprensión inmediata de qué ocurrió (más que teletransportar la ficha sin animación).
- **Interfaz durante el turno del otro:** Los jugadores que no están en turno pueden tener su interfaz semi-bloqueada para evitar confusiones. Por ejemplo, el botón de tirar dado está oculto o gris. Pueden de todos modos interactuar con chat o consultar reglas. Pueden mover la cámara si el tablero fuera grande (en 100 casillas tal vez quepa completo en pantalla sin scroll; si fuera un tablero más grande, se permitiría paneo/zoom).
- **Información privada (en caso de cartas):** Si implementamos cartas sorpresa, cada jugador tendría en su vista una mano (pequeñas cartas mostradas en la esquina). Solo ellos las ven. Si usan una carta, la jugarían (click o drag sobre tablero), y el efecto se refleja públicamente (“Jugador X usó carta Y”). Las cartas no usadas

permanecen ocultas.

Soporte multiplataforma: En escritorio, se ve todo en un pantallazo. En dispositivos móviles, quizás la UI deba adaptarse: el tablero podría mostrarse ajustado a pantalla y el panel de jugadores/historial ocultable en un botón (para aprovechar pantalla pequeña). Los mockups deberían contemplar versión desktop y versión móvil (media queries CSS o diseños adaptativos).

Para elaborar los **mockups del tablero**, se pueden usar herramientas de diseño gráfico. *Draw.io* o *Lucidchart* permitirían hacer un diagrama del tablero con posiciones y fichas, mientras que *Figma* podría usarse para un prototipo visual con los elementos UI (botones, paneles). Por ejemplo, un mockup en Figma presentaría una pantalla con:

- Un área central cuadrada marrón claro (tablero) con cuadritos y algunas escaleras dibujadas.
- Un panel lateral derecho con fondo semitransparente negro donde se listan jugadores y log.
- Botón “Lanzar” grande abajo quizás de color verde.
- Icono de menú y quizás icono de chat en esquinas.

De esta forma, podemos **visualizar** cómo luce la interfaz final y ajustar detalles antes de implementarla. La prioridad es que el tablero sea **claramente entendible** (los jugadores ven inmediatamente dónde están todos y qué sucede en cada turno) y que los controles sean **fáciles de usar** incluso para niños, dado el perfil de este juego.

10. Descripción preliminar de objetos e instancias (modelo de clases)

Para estructurar la lógica del juego en el servidor, se han identificado las principales **clases (objetos)** que participan en el sistema, con sus atributos y responsabilidades clave. A continuación se describen estos elementos:

- **Game (Partida):** Representa una partida en curso o creada. Es la clase central que coordina el juego. Sus atributos incluyen un identificador único (**gameId**), el estado de la partida (pendiente, en juego, finalizada), la colección de jugadores participantes, el tablero asociado y probablemente el turno actual. Sus métodos cubren la inicialización y flujo principal: por ejemplo **start()** para iniciar la partida (coloca a todos en casilla 1, asigna orden de turnos), **endGame()** para finalizar (determina ganador, realiza limpieza). También podría tener métodos para procesar acciones de jugador delegando a otras clases, e.j. **rollDice(player)** que internamente usa el dado, mueve la ficha y actualiza estado.

- Player (Jugador):** Representa a un jugador (usuario) dentro de una partida. Atributos: un identificador o referencia al User real (si hay persistencia de usuarios), un nombre/alias, posiblemente un color o avatar asignado, y su posición actual en el tablero (número de casilla en la que está su ficha). También un flag `isAdmin` para saber si es administrador de esa partida. Puede tener un historial personal (e.g., tiradas realizadas). Métodos: `moveTo(cell)` para moverse a una casilla específica (usado cuando el Game determina la nueva posición), actualizar su posición; quizás `rollDice()` que llamaría al dado global (o TurnManager) para obtener un valor. Si se implementan cartas sorpresa, Player podría tener una lista de `cards` en mano y métodos para `useCard()`.
- Board (Tablero):** Contiene la representación del tablero de juego. Sus atributos podrían incluir el tamaño (ej. 100 casillas), y estructuras para las escaleras y toboganes presentes. Por ejemplo, una lista de objetos Ladder, una lista de objetos Slide, y/o un mapa clave-valor que dado un número de casilla devuelva la casilla destino si es especial. También podría tener una matriz o lista de `Cell` (casillas). Métodos: `init()` o constructor que construye las escaleras y toboganes en las casillas apropiadas (ya sea fijo o aleatorio si se quisieran tableros distintos), métodos para consultar eventos en una casilla (e.g., `getDestinationIfAny(cellNum)` que devuelve otro número si hay escalera/tobogán en cellNum). El Board no necesariamente maneja la lógica de movimiento (eso lo hace Game), pero sí conoce la configuración espacial del juego.
- Cell (Casilla):** Clase opcional para modelar cada casilla del tablero. Podría tener atributos como número de casilla, y booleanos o referencias para si tiene inicio de escalera o tobogán. En un diseño simple, quizás no necesitamos instanciar 100 objetos Cell, bastaría con estructuras en Board. Pero si se usa, Cell puede tener `next` o `shortcut` pointer: por ejemplo, `Cell.ladderTo` apunta a la casilla superior si es escalera (null si no), `Cell.slideTo` apunta a la casilla inferior si es tobogán. Esto haría fácil preguntar: *si `cell.ladderTo` != null entonces hay escalera*. También puede tener coordenadas x,y para representación gráfica si se usara en frontend (en el server no hace falta).
- Ladder (Escalera):** Objeto que encapsula una escalera. Sus atributos básicos son `start` (casilla donde inicia) y `end` (casilla donde termina). En una escalera válida, `end > start`. No suele haber métodos complejos; tal vez solo un constructor y alguna función utilitaria. Podría ser una subclase de una clase genérica `BoardElement` o `Jump` junto con Slide, dado que ambos comparten comportamiento similar (un salto entre casillas). En diseño OO, podríamos tener una clase base `Jump` con start/end, de la cual Ladder y Slide heredan, o simplemente mantenerlas separadas.
- Slide (Tobogán/Serpiente):** Similar a Ladder, con `start` y `end` (aquí `end < start` siempre). Representa una bajada. Igual que Ladder, mayormente datos. En algunos diseños, Ladder y Slide no son necesarias como clases separadas; el Board

puede guardar pares de enteros para atajos. Sin embargo, si planeamos extensibilidad, tenerlas como clases permite añadir atributos (ej. quizás cada escalera/tobogán tiene un texto o nombre, o un efecto especial distinto, etc.). Por ahora, se asumen simples.

- **Dice (Dado):** Esta clase modela el dado de 6 caras. Atributo: número de caras (6 por defecto). Método principal: `roll()` que devuelve un entero aleatorio entre 1 y 6. Puede implementarse como clase estática o utilitaria, pero como objeto permite tener distintas configuraciones. En nuestro caso, `Dice.roll()` usa la librería `random` del lenguaje servidor para generar la tirada.
- **TurnManager (GestorTurnos):** Maneja la secuencia de turnos de los jugadores. Atributos: puede tener una cola o lista de jugadores en el orden actual, y quizás trackear el jugador actual. Ejemplo: `currentIndex` apuntando al índice del jugador en turno en la lista ordenada. Métodos: `nextTurn()` que avanza al siguiente jugador cíclicamente (o según reglas, saltando si alguien abandonó, etc.), `getCurrentPlayer()`. Podría implementar la regla de repetir turno si sacó 6 (por lo que `nextTurn()` quizás recibe un parámetro flag para no avanzar en ese caso). En versiones más complejas, TurnManager también controlaría temporizadores (contando tiempo por turno, y si expira, forzar `nextTurn`). En algunos diseños, TurnManager puede ser absorbido por Game (pues Game sabe quién es el siguiente), pero separarlo ayuda a organizar si hay lógica de turnos elaborada.
- **Card (Carta Sorpresa):** Si incorporamos eventos aleatorios adicionales mediante cartas, habría una clase Card. Atributos: un identificador o tipo, descripción del efecto. Ejemplos de cartas: “Avanza 3 casillas extra”, “Intercambia posición con el jugador más cercano”, “Pierdes el próximo turno”, etc. Método: `applyEffect(Game game, Player player)` que aplique el efecto en el contexto del juego. Estas cartas podrían estar en un mazo del Board o Game; se robarían cuando un jugador cae en cierta casilla especial (no mencionada explícitamente en requisitos pero insinuada en “cartas sorpresa”). Si se incluye esta mecánica, habría también un `Deck` (mazo) que administra las cartas disponibles.
- **ChatMessage:** (si chat implementado) una clase simple con autor, texto, timestamp, para manejar la cola de chat de la partida.

Estas son las clases principales en el dominio del juego. También existen otras en la periferia del sistema:

- **User:** si distinguimos Player (en juego) de User (cuenta global). User contendría info de cuenta (username, email, contraseña hash). Un User puede tener cero o más Player (en distintas partidas). La relación se maneja en la base de datos pero en backend podríamos tener la entidad User para autenticación.
- **Lobby/MatchMaker:** Podría ser una clase que gestiona la lista de partidas activas, crea nuevas, lista disponibles. En un servidor sencillo, esto puede ser simplemente

métodos de un controlador que operan sobre colecciones estáticas de Game.

- **Server/Controller:** Clases encargadas de recibir los mensajes JSON, interpretar y delegar en las instancias Game correspondientes. Por ejemplo, un controlador que cuando recibe `rollDice` busca el objeto Game con ese gameId, y llama `game.processRoll(playerId)`.

En la siguiente sección de Diagrama UML, se reflejan las relaciones entre varias de estas clases, sus atributos y métodos principales, para clarificar la estructura de diseño.

11. Eventos aleatorios: escaleras, toboganes, cartas sorpresa

En el juego, los **eventos aleatorios** introducen emoción y cambios bruscos en la posición de los jugadores. Los clásicos son las escaleras y toboganes (serpientes), a los que en nuestra versión agregamos la idea de “cartas sorpresa” como elemento adicional opcional.

- **Escaleras:** Como ya se mencionó, son atajos positivos. Están representadas por pares de casillas (inicio y fin) donde $\text{inicio} < \text{fin}$. **Mecánica:** Si un jugador termina su movimiento exactamente en la casilla de inicio de una escalera, inmediatamente “sube” por ella hasta la casilla de fin. Esto ocurre de manera automática, sin intervención del jugador (regla intrínseca del tablero). La presencia de escaleras es predefinida al inicio de la partida (el tablero tiene, digamos, 8 escaleras colocadas en posiciones fijas o aleatorias según diseño). Desde el punto de vista del sistema, subir por una escalera es un evento determinístico dado el estado (posición), pero desde el punto de vista del jugador es un evento aleatorio favorable (depende de caer justo ahí, lo cual es cuestión de suerte). Las escaleras permiten que un jugador que estaba rezagado pegue un salto hacia adelante en el recorrido.
- **Toboganes (Serpientes):** Son el contrapunto negativo. También pares de casillas con $\text{inicio} > \text{fin}$. Si un jugador termina en la casilla inicial de un tobogán, su ficha “se desliza” hacia abajo hasta la casilla fin (más baja). Esto típicamente atrasa al jugador, a veces deshaciendo el avance de varias tiradas. Igual que las escaleras, las posiciones de toboganes son fijas en el tablero. Son eventos no evitables: si caes ahí, sufres la consecuencia. En la implementación, el servidor tras mover la ficha chequea `if (board.hasSlideAt(player.position)) then player.position = slide.end`. Estos toboganes introducen incertidumbre, ya que incluso quien lidere puede caer en uno y perder ventaja.

Ambos elementos (escaleras y toboganes) aportan **alta variabilidad** al juego, haciendo que no todo dependa de que alguien saque números altos consistentemente. En la versión física su distribución es conocida (muchas veces, escalera larga beneficia mucho, tobogán largo castiga mucho). Podríamos programar distintos tableros con diferentes distribuciones para variedad.

- **Cartas sorpresa:** Para innovar más allá del juego clásico, podemos incluir casillas especiales que al caer en ellas hacen que el jugador tome una *carta sorpresa*. Estas cartas podrían equivaler a eventos aleatorios adicionales (similar a tarjetas de la suerte en Monopoly, por ejemplo). La mecánica sería: ciertas casillas marcadas (por ejemplo, casilla 8 y casilla 22, etc. definidas en el tablero) son “Casilla Sorpresa”. Si un jugador cae ahí, el sistema le asigna aleatoriamente una carta de un mazo prediseñado. **Ejemplos de cartas sorpresa:**
 - **“Avanza 3 casillas”:** en ese mismo momento, el jugador mueve 3 casillas más adelante (y si con eso cae en escalera/tobogán, también se aplica). Es un pequeño bonus.
 - **“Retrocede 2 casillas”:** lo opuesto, un pequeño castigo.
 - **“Pierdes el próximo turno”:** se marca al jugador para saltarlo una vez en el orden de turnos.
 - **“Intercambia posición con el jugador que va primero”:** una carta potente que podría cambiar significativamente la situación (el jugador que la saca cambia su ficha a donde esté el líder, y el líder pasa a su casilla).
 - **“Escudo”:** una carta que el jugador conserva y puede usar para evitar el efecto de un tobogán la próxima vez que caiga en uno, por ejemplo.
 - **“Dado especial”:** la próxima tirada de ese jugador será con un dado de 8 caras en lugar de 6 (o repetirá si saca par, etc.), dando una chance de avance extra.

Estas son ideas; en la implementación podríamos tener un conjunto de, digamos, 10 cartas posibles. Al caer en la casilla sorpresa, el servidor elige una al azar (usando RNG) de las disponibles no usadas (o pueden ser infinitas si se permiten repeticiones). Luego envía un mensaje al jugador: “Has obtenido la carta X”. Si la carta es de efecto inmediato (ej. avanzar 3), se aplica en el acto (y se notifica a todos en el historial: “Jugador X obtuvo carta 'Avanza 3' y se mueve 3 casillas más adelante”). Si es guardable (ej. escudo), se almacena en el inventario del Player y se podría mostrar un iconito en su interfaz. El jugador la podrá usar en otro momento (lo cual requiere un botón “Usar carta X” durante su turno o en reacción a algo).

La introducción de cartas aumenta la complejidad de las reglas de negocio, pues el servidor debe manejar la lógica de cada efecto. En el diseño, cada tipo de carta se implementaría en la clase Card con su método efecto. Por simplicidad en un prototipo, podríamos hacer que todas las cartas se apliquen inmediatamente al sacarlas (no guardables) así evitamos estado extra de cartas en mano.

Balance y aleatoriedad: Estas cartas son aleatorias en adquisición (depende de caer en la casilla sorpresa, lo cual es en sí por azar del dado) y en contenido (suerte de qué carta sale). Añaden un nivel de *azar adicional*. Hay que vigilar que no rompan el juego: por

ejemplo, la carta de intercambiar posición es muy fuerte; conviene calibrar cuántas de esas hay. Un enfoque es tener más cartas pequeñas (avanza, retrocede) y pocas disruptivas.

Integración con escaleras y toboganes: Todas estas mecánicas conviven. Es posible que un jugador en un turno experimente varios eventos: ej. lanza dado, cae justo en casilla sorpresa, saca una carta "avanza 3", entonces se mueve 3, y esa nueva casilla resulta ser inicio de escalera, sube aún más. El servidor debe manejar la secuencia: carta -> mover adicional -> escalera -> etc., antes de concluir el turno. Debemos garantizar que la interfaz comunique todo (por ejemplo: "¡Sorpresa! Avanza 3 casillas... subes por escalera a 90").

Implementación en servidor: Escaleras y toboganes están en Board predefinidos. Las cartas sorpresa, si se incluyen, también se predefinen qué casillas las dan. Podría modelarse con una lista `specialCells` en Board: e.g., `{8: "drawCard", 22: "drawCard", 50: "drawCard"}`. Cuando un jugador termina en una casilla que en ese map tiene "drawCard", el Game sabe que debe sacar carta. Se integraría así:

pseudo

CopiarEditar

```
if board.special[cell] == "drawCard":
    card = deck.drawRandom()
    applyCardEffect(card, player)
```

Si la carta es inmediata, `applyCardEffect` podría llamar recursivamente a la misma rutina de movimiento (si dice avanzar X, se mueve). Si es diferida, se guarda en `player.cards`.

Jugabilidad: Estas cartas pueden hacer el juego menos predecible y más divertido en digital. Para el proyecto, al menos describirlas muestra creatividad en el diseño. En la práctica, se podría activar/desactivar esta opción en la partida (modo clásico vs modo extendido).

En resumen, *escaleras y toboganes* son eventos posicionales predeterminados que aceleran o retrasan a los jugadores y ocurren automáticamente al caer en ciertas casillas. Las *cartas sorpresa* son eventos aleatorios adicionales introducidos por casillas especiales o alguna otra condición, que pueden tener efectos variados en el juego, implementadas para aportar novedad. Todas estas mecánicas deben estar correctamente implementadas en las reglas del servidor para que el juego las aplique de forma justa y coherente.

12. Modelo Entidad-Relación (orientado a PostgreSQL)

Para la persistencia de datos (particularmente de usuarios y partidas) se propone un **modelo entidad-relación (ER)** adecuado para PostgreSQL. Este modelo define las tablas principales y sus relaciones en la base de datos.

Diagrama Entidad-Relación del sistema (simplificado). Las entidades principales son Usuario, Partida, Jugador, Movida, Escalera y Tobogán. Se observan claves primarias (PK)

en cada tabla y claves foráneas (FK) que establecen las relaciones. Por ejemplo, un Usuario puede estar asociado a múltiples Jugadores (participaciones en partidas), y cada Jugador registra cuál Usuario y cuál Partida representa. Las entidades Ladder (Escalera) y Slide (Tobogán) están vinculadas a Partida para permitir configurar tableros específicos. Move (Movida) registra el historial de jugadas de cada partida. Este modelo sería implementado en PostgreSQL, respetando integridad referencial y permitiendo consultas eficientes del estado del juego.

En detalle, las principales entidades y sus campos:

- **User (Usuario):** Tabla para los usuarios registrados. Campos clave: `user_id` (PK, llave primaria única por usuario, quizás serial), `username` (nombre de cuenta, único), `email` (único), `password` (almacenar hash de contraseña). También se pueden guardar fecha de registro, último login, etc., pero lo básico son esas credenciales. Un usuario puede participar en múltiples partidas a lo largo del tiempo.
- **Game (Partida):** Tabla de partidas. Campos: `game_id` (PK único por partida), código o nombre de partida (p. ej., `join_code` si se usa para invitaciones), `status` (estado actual: pending, started, finished, etc.), y posiblemente `created_at`, `finished_at` timestamps. Podríamos guardar también `max_players` permitido. Si se quiere soportar diferentes tableros, se podría guardar un `board_type` o incluso una referencia a una tabla Board (no incluida en diagrama para simplicidad; aquí asumimos un único tipo de tablero, y guardamos escaleras/toboganes en tablas separadas atadas al Game).
- **Player (Jugador):** Esta entidad relaciona Usuarios con Partidas, y almacena información de cada *participación de un usuario en una partida*. Campos: `player_id` (PK, puede ser serial), `game_id` (FK referencia a Game), `user_id` (FK referencia a User). Además, atributos propios del rol en esa partida: `is_admin` (boolean, true si este jugador es el creador/administrador de la partida), `position` (int, posición actual en el tablero; el servidor puede actualizar este campo después de cada turno si se desea persistir el estado, aunque podría ser redundante con estado en memoria). También se podría guardar algo como `joined_at` (timestamp de cuando se unió a la partida). La clave foránea `game_id, user_id` conjunta podría ser única para evitar duplicados (un mismo user no puede unirse dos veces a la misma partida). La relación lógica es: **User 1:N Player, Game 1:N Player**, esto permite modelo muchos a muchos entre User y Game mediante Player.
- **Move (Movida/Jugada):** Registra las acciones de juego (historial de tiradas). Cada registro sería una tirada de dado y su resultado. Campos: `move_id` (PK), `game_id` (FK a Game, la partida a la que pertenece la movida), `player_id` (FK a Player, quién hizo la jugada), `dice_roll` (valor del dado obtenido), `from_cell` y `to_cell` (casilla origen y destino luego de aplicar escalera/tobogán, etc.), `timestamp` (momento de la jugada). Esto permitiría reconstruir el historial o mostrarlo. Se puede guardar también qué tipo de evento ocurrió (`event` string: "ladder", "slide", "normal",

“cardX” para saber si subió/bajó/usó carta). Con estos datos guardados, se podrían hacer estadísticas (ej: cuántas escaleras tomó cada uno, cuántos toboganes). La relación es **Game 1:N Move** (una partida tiene muchas movidas) y **Player 1:N Move** (un jugador hace muchas movidas).

- **Ladder (Escalera):** Representa una escalera específica en una partida (si las posiciones de escaleras pueden variar por partida). Campos: `ladder_id` (PK), `game_id` (FK a Game, indicando en qué partida/tabla aplica; si todos los juegos usan mismo tablero fijo, se podría omitir `game_id` y tener una tabla estática, pero si algún día se quiere tableros distintos o aleatorios, mejor asociarlas a game), `start_cell` (casilla de inicio), `end_cell` (casilla de fin superior). La relación es **Game 1:N Ladder**. Antes de iniciar la partida, se insertan en esta tabla todas las escaleras del tablero de esa partida. Si todas las partidas comparten mismo set de escaleras estándar, podríamos almacenar solo una vez y referenciar por board type, pero dado el alcance del proyecto, registrar por partida da flexibilidad.
- **Slide (Tobogán):** Similar a Ladder. Campos: `slide_id` (PK), `game_id` (FK), `start_cell`, `end_cell`. Representa serpientes/toboganes. Relación **Game 1:N Slide**.

Cabe destacar que en un tablero clásico, Ladder y Slide en realidad podrían combinarse en una sola entidad "Atajo" con campo tipo (ladder/slide), ya que son casi idénticos en estructura. Pero mantenerlas separadas puede simplificar consultas (saber cuántas escaleras vs serpientes hay, etc.).

- **Card (Carta):** Si implementamos cartas sorpresa, podríamos tener tabla Card (con definiciones de cada tipo de carta, pero eso es estático) y quizás una tabla `CardDraw` para registrar cuándo un jugador sacó qué carta en qué partida. Dado que es un extra, no lo incluimos en el diagrama para no recargar; se podría incorporar si se detalla esa funcionalidad.
- **Otros elementos:** No mostrados explícitamente:
 - Si tuviéramos un sistema de chat persistente, una tabla `ChatMessage(game_id, player_id, message, timestamp)` para guardar mensajes.
 - Tabla `Board` si se quisieran predefinir configuraciones de tablero (diferentes distribuciones de escaleras/toboganes) y luego Game reference a Board. Por sencillez asumimos un único layout manejado por Ladder/Slide por Game.
 - Tabla `Token/Session` para manejar las sesiones de login (o podríamos usar JWT sin tabla de sesión).

Integridad referencial:

- Al crear una Player, debe existir el User y el Game referidos (FK).
- Moves solo existen con player válido y game válido asociados.
- Ladder/Slide deben referenciar a un Game existente. Podríamos agregar constraint de que `start_cell < end_cell` en Ladder y `start_cell > end_cell` en Slide para mantener consistencia.
- Si un Game se borra (en caso de limpieza), habría que borrar en cascada sus Players, Moves, Ladder, Slide.

Consultas típicas soportadas por este modelo:

- Obtener todos los jugadores de una partida: `SELECT * FROM Player WHERE game_id = X;` (join con User para nombres).
- Obtener historial de una partida: `SELECT * FROM Move WHERE game_id = X ORDER BY timestamp;`
- Verificar credenciales de usuario: `SELECT * FROM User WHERE username = ?;`
- Contar escaleras y toboganes de un tablero: `SELECT COUNT(*) FROM Ladder WHERE game_id=X;`
- Ranking o estadísticas: se podría contar cuántas partidas ganó un usuario, pero para eso habría que almacenar ganador en Game (campo `winner_player_id` por ejemplo). No se mostró, pero podríamos añadirlo: `Game.winner_id` (FK a Player o User del ganador).

El modelo está **orientado a PostgreSQL**, usando tipos básicos (INTEGER para ids y posiciones, VARCHAR para nombres, etc., TIMESTAMP para fechas). PostgreSQL soporta serial (autoincrement) para PKs, o usar UUIDs si se prefiere. Podrían crearse *índices* en campos de búsqueda común (username, game status, etc.).

Herramientas como *dbdiagram.io* o *drawSQL* podrían emplearse para dibujar este diagrama ER durante la planificación, facilitando visualizar las relaciones. En nuestro caso, el diagrama muestra claramente las relaciones de uno a muchos:

- Un **Usuario** puede estar en muchas partidas (vía Player).
- Una **Partida** tiene muchos Jugadores, muchos Moves, muchas Escaleras y Toboganes.
- Cada **Jugador** pertenece a una Partida y está ligado a un Usuario.

- Cada **Move** pertenece a una Partida y a un Jugador específico (que a su vez sabe el usuario).

Este modelo asegura que podamos persistir la información necesaria para reanudar partidas si fuese necesario o para análisis posterior, aunque en un juego simple tiempo real, muchas de estas cosas podrían mantenerse solo en memoria. Pero dado que es un proyecto académico, es valioso almacenar resultados de partidas, logs y usuarios registrados.

13. Diagrama de clases UML (atributos, métodos y asociaciones)

A continuación se presenta un **diagrama de clases UML** que ilustra las principales clases del sistema, sus atributos más relevantes y métodos, así como las relaciones (asociaciones) entre ellas. Este diagrama se centró en la lógica del juego (modelo del dominio), excluyendo clases de infraestructura o UI.

*Diagrama UML de clases principales para el juego Escaleras y Toboganes. Se incluyen las clases Game, TurnManager, Player, Board, Cell, Ladder, Slide, Dice y sus relaciones. Por ejemplo, Game tiene una asociación 1.. con Player (muchos jugadores en una partida) y una composición 1 con Board (contiene un tablero). TurnManager se asocia con Game para manejar el turno actual. Player utiliza la clase Dice para sus lanzamientos. Ladder y Slide están vinculados al Board (como elementos del tablero), y ambos conectan con Cell para indicar sus casillas de inicio/fin. Este diagrama fue diseñado utilizando una herramienta UML (p.ej., Lucidchart o draw.io) siguiendo las convenciones de notación estándar.**

En el diagrama:

- Las **flechas/relaciones** muestran, por ejemplo, que **Game** contiene a **Board** y maneja múltiples **Player**. También que **TurnManager** controla el turno de **Player**.
- Los **atributos** de cada clase (parte inferior de cada caja) reflejan lo discutido: Player tiene name, tokenColor, position; Game tiene players, board, turnManager; etc.
- Los **métodos** principales también se listan: Game.start(), Game.endGame(), Player.rollDice(), Player.move(int), TurnManager.nextTurn(), Dice.roll(), etc.
- Notar que Ladder y Slide podrían haberse modelado como una jerarquía, pero aquí simplemente se ven como clases separadas asociadas a Cell (en este UML simplificado se indica con "link" para marcar la conexión entre una escalera/tobogán y las celdas que conecta).

Este diagrama sirve para validar la arquitectura orientada a objetos. Por ejemplo, muestra que:

- La clase **Game** es agregadora: conoce a sus jugadores, al tablero, y delega en TurnManager y Dice las funciones respectivas. Esto sugiere que en la implementación, Game podría tener métodos como `processTurn()` que usando TurnManager obtenga el jugador actual y luego use Dice y Board para efectuar el movimiento.
- **Player** aquí tiene un método `rollDice()`, pero en la lógica real, sería el Game/TurnManager quien lo llama cuando es el turno del Player. `Player.rollDice()` podría simplemente llamar a `Dice.roll()` y quizás devolver ese valor.
- La relación entre **Board** y **Cell/Ladder/Slide** indica que el Board contiene la estructura del tablero. Por simplicidad, Cell se asocia con Ladder/Slide como "link" en la imagen (es decir, Ladder sabe qué Cell es su inicio y fin, pero podríamos también simplificar y decir `Ladder.start` y `Ladder.end` son enteros índices de Cell).
- **TurnManager** en este diagrama se asocia con Game (posible composición también) y con Player (indicando a quién le toca). `TurnManager.currentPlayer`: Player muestra que mantiene referencia al jugador actual.

El UML es una vista abstracta; en la implementación real algunos detalles pueden cambiar (por ejemplo, podríamos no crear realmente objetos Cell para cada casilla sino manejar enteros, etc.). Sin embargo, es útil para distribuir responsabilidades:

- Game: flujo general, integra todo.
- Player: estado individual.
- Board: configuración estática.
- TurnManager: lógica de turnos separada.
- Dice: aleatoriedad aislada.

Este diagrama fue construido durante la fase de diseño usando herramientas como *draw.io* (también conocida como *diagrams.net*) o *StarUML*, lo que permitió iterar en la estructura antes de codificar. Mantener actualizado el diagrama ayuda a documentar el proyecto para futuros mantenedores o para la memoria del estudiante.

14. Tablero Kanban (Jira/Trello) con tareas, estados y responsables

Para la gestión del proyecto de desarrollo, el equipo utilizó un **tablero Kanban** al estilo Trello/Jira, donde se organizaron todas las tareas en columnas según su estado de avance, y se asignaron responsables a cada tarea. A continuación se muestra una captura de ejemplo de dicho tablero:

Captura de un tablero Kanban utilizado en el proyecto (simulado con Trello). Las columnas típicas representadas son “To Do” (Por hacer), “In Progress” (En progreso), “Review” (En revisión/pruebas) y “Done” (Hecho). Cada tarjeta corresponde a una tarea específica: por ejemplo, en “To Do” se ven tareas pendientes como “Weekly Social Media Posts” (en un contexto de marketing) solo a modo ilustrativo, mientras que en nuestro proyecto podrían ser “Diseñar pantalla de registro” o “Implementar lógica de escaleras”. En “In Progress” aparecen tareas que alguien está desarrollando actualmente (p.ej., “Implement Lead Scoring” como ejemplo genérico; en nuestro caso podría ser “Codificar clase TurnManager”). La columna “Review” tendría tareas completadas a la espera de verificación (testeo, revisión de código) y finalmente “Done” lista las tareas terminadas (ej., “Blog Post 1” completado). Cada tarjeta suele tener asignado el miembro responsable y posiblemente etiquetas de prioridad.

En nuestro proyecto, adaptando ese ejemplo, hubiéramos tenido tarjetas como:

- *To Do*: “Crear modelo ER”, “Diseñar mockups UI”, “Configurar servidor Node/Express”, “Desarrollar componente de chat”.
- *In Progress*: tareas en curso con el nombre del desarrollador asignado. Ej: “Implementar backend registro (Alice)”, “Frontend tablero de juego (Bob)”.
- *Review/Testing*: tras completar, otro miembro o el mismo pasa la tarjeta aquí indicando que necesita prueba. Ej: “Revisar lógica de turnos (Carlos)”.
- *Done*: tareas finalizadas, como “Base de datos desplegada”, “Documento de diseño escrito”.

Cada **tarjeta** en Trello puede tener descripciones detalladas, checklist de subtareas, etiquetas (por ejemplo: “Frontend”, “Backend”, “High Priority”) y comentarios de colaboradores. Los **responsables** aparecen típicamente como pequeños avatares en cada tarjeta. Por ejemplo, la tarea “Diseñar pantalla de registro” podría mostrar el avatar de *Juan* si él la está haciendo.

Usar este tablero permitió al equipo seguir un proceso ágil **Kanban**: las tareas se arrastran (drag-and-drop) de *To Do* a *In Progress* cuando se empiezan, luego a *Done* al completarse, proporcionando una vista visual del progreso. Si se usó **Jira**, el concepto es similar pero con más campos (Jira suele usarse con Scrum también, manejando sprints).

El tablero refleja el flujo de trabajo:

1. **Backlog** (To Do): inicialmente poblado con todas las historias de usuario o tareas identificadas en la planificación.
2. **In Progress**: lo que se está haciendo actualmente. Limita el WIP (work in progress) a un par de tareas por desarrollador para mantener foco.
3. **Review/QA**: una columna extra que Trello no tiene por defecto pero se puede añadir. Sirve para indicar que la tarea está hecha por el dev pero pendiente de revisión de

código o de pruebas integrales.

4. **Done:** finalizadas y verificadas.

Por ejemplo, si *María* termina de implementar las “Reglas de negocio de escaleras/toboganes”, moverá esa tarjeta a Done. Si *José* está a mitad de “Implementar protocolo JSON”, su tarjeta está en In Progress. Este método hace fácil la reunión diaria (daily stand-up) pues todos ven el tablero y comentan bloqueos o próximos pasos.

El tablero Kanban es una herramienta vital para mantener la **organización** del equipo, la **visibilidad** del estado del proyecto para todos los miembros y asegurar que cada tarea tiene un dueño responsable (accountability). Para este proyecto académico, Trello resultó adecuado por su simplicidad y flexibilidad, permitiendo incluso adjuntar nuestros diagramas (como el UML o el ER) a tarjetas de tareas correspondientes.

En conclusión, gracias a este tablero, el equipo pudo dividir el desarrollo en tareas manejables (diseño de UI, programación de lógica, pruebas, documentación), hacer seguimiento del avance y colaborar eficientemente. Este enfoque visual de gestión es altamente recomendado en proyectos de ingeniería de software, y complementa la documentación técnica y de diseño presentada en las secciones anteriores.