Universidade Vigo

ESCOLA SUPERIOR DE ENXEÑARÍA INFORMÁTICA

Memoria do Traballo de Fin de Grao que presenta

D. César Nieto González

para a obtención do Título de Graduado en Enxeñaría Informática

Aplicación de Pizarra de Jugadas para Baloncesto

Xuño, 2025

Traballo de Fin de Grao Nº: El 24/25-41

Titor/a: Celso Campos Bastos
Área de coñecemento: Linguaxes e Sistemas Informáticos
Departamento: Informática

Agradecimientos

Gracias a mi tutor por su orientación, a mi familia, amigos y novia por estar siempre, y sobre todo a mi hermano, que así no se podrá olvidar las jugadas nunca más.

Índice

1. Introducción y Objetivos

1.1. Introducción

En el ámbito del deporte y la formación táctica, la representación visual de jugadas y movimientos se ha convertido en una herramienta clave para entrenadores, docentes y deportistas. Ya sea en un aula o en un entrenamiento, disponer de una pizarra digital que permita construir, guardar y reproducir esquemas tácticos en tiempo real mejora notablemente la comprensión de conceptos estratégicos y facilita la comunicación visual.

Sin embargo, la mayoría de las herramientas actuales son de pago, tienen interfaces limitadas o no permiten trabajar con secuencias dinámicas de movimientos. Esto dificulta su uso en contextos educativos, donde la flexibilidad, la simplicidad de uso y la posibilidad de experimentar con jugadas paso a paso son fundamentales.

Ante esta necesidad, surge el presente Trabajo de Fin de Grado, que propone el desarrollo de *MiPizarra*, una aplicación interactiva diseñada para facilitar la creación, edición y reproducción de jugadas tácticas. Esta herramienta, desarrollada en Java y JavaFX, está pensada para ser multiplataforma y enfocada a usuarios no técnicos, permitiendo representar visualmente desplazamientos, bloqueos, zonas y secuencias completas de juego.

El proyecto no solo se centra en la creación de elementos gráficos, sino también en la gestión inteligente de jugadas como secuencias temporales, la persistencia en formato JSON, el reescalado dinámico de la interfaz y la posibilidad de navegar paso a paso por una jugada completa. Su diseño modular busca facilitar futuras extensiones como exportación multimedia, adaptación a distintos deportes o integración con almacenamiento en la nube.

1.2. Objetivo general

Desarrollar una aplicación multiplataforma de escritorio que permita diseñar, almacenar y reproducir visualmente jugadas tácticas en un entorno gráfico interactivo, facilitando su uso tanto en contextos deportivos como educativos.

1.3. Objetivos específicos

- Diseñar una interfaz gráfica amigable y accesible basada en JavaFX.

- Implementar distintas herramientas de dibujo: líneas con flecha, líneas de bloqueo, líneas discontinuas, zonas circulares y trazado libre.
- Desarrollar un sistema de gestión de jugadas que permita registrar movimientos paso a paso.
- Incorporar funciones de navegación: avanzar, retroceder, reiniciar y reproducir jugadas.
- Aplicar reescalado automático de los elementos gráficos según el tamaño de ventana.
- Gestionar la persistencia de datos utilizando archivos JSON.
- Permitir guardar, cargar y reiniciar jugadas desde la interfaz.
- Añadir funcionalidades de deshacer y rehacer acciones gráficas mediante pilas de estados.
- Garantizar compatibilidad con sistemas Windows, macOS y Linux mediante Java y Maven.
- Documentar el uso, instalación y arquitectura del sistema para facilitar su mantenimiento y mejora futura.

1.4. Metodología empleada

El desarrollo del proyecto se ha llevado a cabo de forma iterativa, dividiendo el trabajo en etapas funcionales bien delimitadas. A pesar de tratarse de un desarrollo individual, se ha aplicado una adaptación del marco ágil SCRUM para organizar el trabajo de forma iterativa e incremental.

Cada etapa (similar a un sprint) abordó un conjunto coherente de funcionalidades o tareas:

1. Análisis preliminar de requisitos.
2. Diseño de arquitectura e interfaz.
3. Implementación de jugadas.
4. Módulos de control de estado e interacción avanzada.
5. Validación, pruebas y documentación.

Esta organización por bloques permitió un control claro del avance, facilitó la detección temprana de errores, y permitió mejorar el producto progresivamente tras cada iteración.

1.5. Planificación

La planificación del proyecto *MiPizarra* se ha desarrollado considerando los tres ejes fundamentales de la gestión de proyectos: **alcance, tiempo y coste**.

Desde el punto de vista del **alcance**, el proyecto se centra en la creación de una aplicación de escritorio multiplataforma que permita a entrenadores, docentes o estudiantes diseñar, reproducir y analizar jugadas tácticas deportivas. El sistema incluye herramientas gráficas de dibujo, navegación paso a paso de movimientos, guardado y carga de jugadas en formato JSON, y funcionalidades de deshacer/rehacer sobre el lienzo táctico. Se excluyen en esta fase inicial funcionalidades web, móviles o colaborativas online, priorizando la estabilidad, claridad de interfaz y utilidad docente.

En cuanto a la **temporalización**, se definió una planificación inicial de 300 horas. Para mejorar la organización y la trazabilidad del trabajo, se dividió el desarrollo en 5 fases bien delimitadas, con una duración semanal y objetivos concretos:

Código	Fase / Subfase	Periodo	Horas
1	Análisis preliminar y requisitos	3–14 marzo 2025	40 h
1.1	Revisión de antecedentes y necesidades del usuario		8 h
1.2	Estudio de soluciones existentes		8 h
1.3	Definición de objetivos generales y específicos		8 h
1.4	Requisitos funcionales y no funcionables iniciales		16 h
2	Diseño de arquitectura e interfaz	17–28 marzo 2025	60 h
2.1	Diseño del lienzo y disposición de elementos		15 h
2.2	Boceto de herramientas de dibujo		15 h
2.3	Estructura de clases gráficas (ElementoPizarra, etc.)		10 h
2.4	Integración con controladores y lógica visual		20 h
3	Implementación de jugadas	31 mar – 11 abril	70 h
3.1	Registro del movimiento desde arrastre		20 h
3.2	Creación y gestión de objetos Movimiento		15 h
3.3	Control de navegación de jugadas (paso a paso)		15 h
3.4	Guardado/cargado de jugadas en formato JSON		20 h

Código	Fase / Subfase	Periodo	Horas
4	Módulos de control de estado e interacción avanzada	14–25 abril 2025	60 h
4.1	Funcionalidad de deshacer y rehacer		20 h
4.2	Control del historial de acciones (DrawState)		10 h
4.3	Reescalado proporcional (scaleX/scaleY) de objetos y ventana		30 h
5	Validación, pruebas y documentación	28 abril – 9 mayo	70 h
5.1	Planificación de pruebas funcionales		15 h
5.2	Ejecución y anotación de resultados de pruebas		15 h
5.3	Evaluación del rendimiento y escalabilidad		15 h
5.4	Elaboración del manual de usuario		10 h
5.5	Formato y revisión final para entrega		15 h

Figura 1: Diagrama de Gantt que representa gráficamente esta planificación.

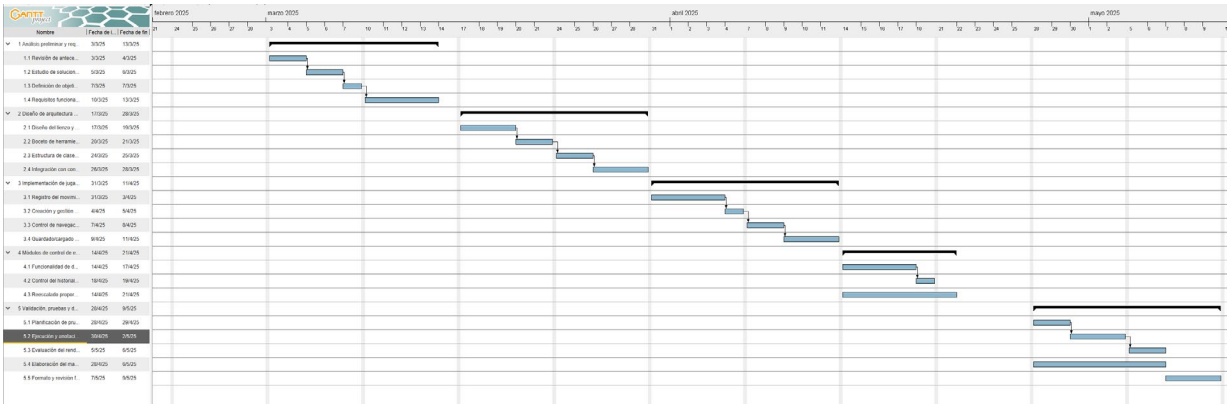
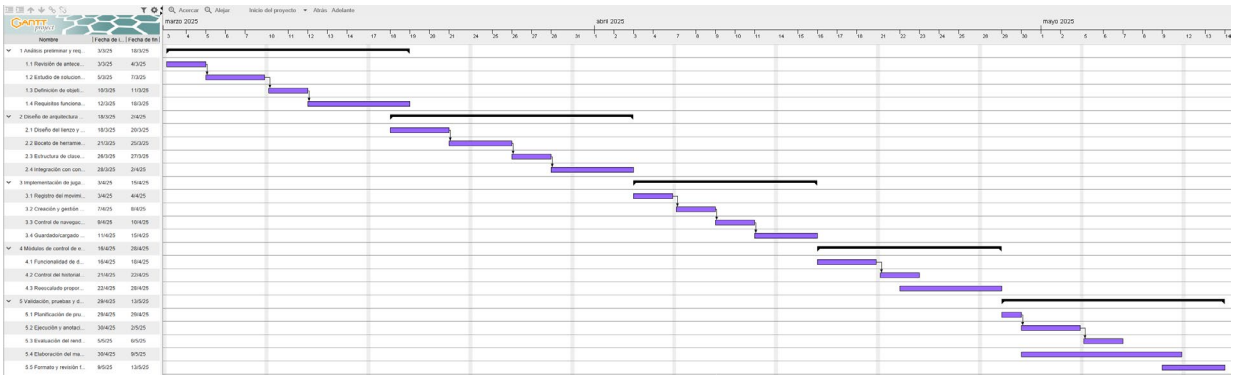


Figura 2: Diagrama de Gantt que representa gráficamente lo sucedido.



A partir de la comparación entre el diagrama de Gantt de planificación y el diagrama de seguimiento, se aprecia que, aunque la estructura general y la secuencia de fases se han respetado, han existido diversas desviaciones temporales a lo largo del desarrollo del proyecto. Estas variaciones responden a factores técnicos, organizativos y prácticos, propios de una ejecución realista y adaptada al progreso efectivo del trabajo.

La **fase 1**, correspondiente al análisis preliminar, presentó una ligera extensión respecto al calendario inicial. La necesidad de revisar con más profundidad algunas fuentes de información y de redefinir ciertos objetivos específicos provocó que la subfase 1.4, centrada en los requisitos funcionales y no funcionales, se alargase hasta el 18 de marzo. Esta decisión se tomó con el objetivo de contar con una base más sólida sobre la que construir el diseño posterior, evitando ambigüedades en fases futuras.

En la **fase 2**, orientada al diseño de la arquitectura e interfaz, se produjo un retraso acumulado de aproximadamente una semana. La principal causa fue la complejidad no prevista en el diseño del lienzo y de las herramientas de dibujo, especialmente en lo relativo a su comportamiento dinámico y a la disposición adaptable de los elementos gráficos. Asimismo, la subfase 2.4, encargada de la integración visual con los controladores, necesitó un esfuerzo adicional para resolver inconsistencias en la sincronización de eventos gráficos, lo cual supuso una extensión de dos jornadas sobre lo previsto.

La **fase 3**, centrada en la implementación de jugadas, comenzó más tarde de lo planificado debido al arrastre de tareas de diseño aún no cerradas. No obstante, se recuperó parcialmente el ritmo mediante el solapamiento de algunas subfases. Por ejemplo, mientras se ultimaban detalles del módulo de arrastre, se inició en paralelo la creación de los objetos Movimiento, gracias a que sus especificaciones eran ya suficientemente estables. Esta reestructuración permitió ajustar el impacto del retraso sin comprometer la calidad del resultado.

En la **fase 4**, se observaron algunas reducciones de tiempo respecto a la previsión inicial. El módulo de historial de acciones, por ejemplo, se benefició del trabajo previo realizado durante la implementación de otras funcionalidades gráficas, permitiendo una integración más ágil. También se optimizó la fase de reescalado de objetos, gracias a la reutilización de estructuras de transformación ya implementadas en tareas anteriores.

Finalmente, la **fase 5**, dedicada a la validación, pruebas y documentación, se desarrolló conforme a lo planificado en sus primeras subfases. Sin embargo, la elaboración del manual de usuario y la revisión final para entrega requirieron una dedicación mayor de la prevista. Esto se debió, por un lado, al esfuerzo invertido en generar una documentación comprensible y estructurada, con capturas y ejemplos detallados; y por otro, a la decisión de realizar una última revisión transversal del código y de las funcionalidades implementadas antes de la entrega final.

En conjunto, el diagrama de seguimiento refleja una ejecución realista del proyecto, con decisiones de reorganización y ajustes puntuales motivados por la lógica del desarrollo. Estos cambios han permitido mantener el cumplimiento de los objetivos planteados, garantizando tanto la funcionalidad como la calidad del resultado final, sin superar el margen temporal global establecido.

En cuanto al coste estimado, se ha realizado una aproximación global que incluye todos los recursos necesarios para el desarrollo del proyecto, aunque se trate de un entorno académico sin implicaciones económicas reales. A efectos de dimensionar el esfuerzo, justificar la dedicación y proporcionar una referencia para su posible extrapolación profesional, se ha considerado una estimación total compuesta por los siguientes componentes:

- **Coste de mano de obra**: Se ha tomado como referencia una tarifa base de 15 €/hora para tareas de análisis, diseño, desarrollo, validación y documentación. Dado que el proyecto ha supuesto una dedicación total de 300 horas, el coste asociado sería de **4.500 €**.
- **Coste de hardware**: Incluye el uso de un ordenador personal ya disponible, por lo que no se ha imputado coste directo. No obstante, en un contexto profesional se podría considerar una amortización proporcional (no contemplada aquí).
- **Coste de software**: Se han empleado herramientas gratuitas o de código abierto (Java, JavaFX, Scene Builder, NetBeans, etc.), por lo que el coste es **nulo** en esta categoría.
- **Gastos generales**: Se podría considerar un pequeño porcentaje adicional (por ejemplo, un 5–10 %) en concepto de electricidad, mantenimiento y posibles materiales de apoyo, elevando el coste estimado a unos **5.000–5.100 €** en total.

Esta valoración no implica un desembolso económico real, pero proporciona un marco de referencia para dimensionar el proyecto en términos de esfuerzo, recursos utilizados y viabilidad futura en otros entornos. Además, permite justificar el alcance y la planificación desarrollada durante las distintas fases del trabajo.

1.6. Estado del Arte

En el contexto actual de las herramientas digitales aplicadas a la representación táctica en deportes colectivos, existen diversas soluciones que permiten visualizar

y planificar jugadas sobre un campo virtual. No obstante, muchas de ellas presentan barreras de acceso, limitaciones funcionales o una orientación específica a usos profesionales de pago. Este apartado recoge una visión general de las herramientas existentes y contextualiza el papel que *MiPizarra* desempeña como alternativa abierta, personalizable y educativa.

Herramientas comerciales existentes

Entre las aplicaciones más reconocidas en el ámbito profesional se encuentran:

TacticalPad
Es una herramienta ampliamente utilizada por cuerpos técnicos profesionales. Permite crear jugadas animadas, guardar esquemas y realizar presentaciones en vídeo. Su punto fuerte es la calidad gráfica y su capacidad de exportación, pero se trata de una solución de pago con funcionalidades limitadas en su versión gratuita.

FastDraw(byFastModelSports)
Es un software centrado en baloncesto, que destaca por su base de datos de jugadas, plantillas específicas y organización por categorías. Está orientado al análisis avanzado, pero su curva de aprendizaje es elevada y no está disponible en español.

YouCoach, CoachBase y similar
Estas herramientas integran pizarras tácticas con bases de datos de ejercicios, y suelen estar vinculadas a plataformas web o móviles con suscripción. Aunque potentes, están menos orientadas al control total por parte del usuario y más enfocadas a su uso guiado dentro de sistemas cerrados.

Herramientas académicas y proyectos libres

En entornos educativos o de código abierto, es habitual encontrar proyectos más limitados, enfocados a la edición estática de jugadas mediante imágenes o PDF. Estas soluciones suelen carecer de interacción directa, historial de acciones, reproducción paso a paso o exportación estructurada. Además, no suelen ofrecer soporte multiplataforma ni estar diseñadas con arquitectura escalable.

Justificación de la propuesta

La aplicación *MiPizarra* nace como una solución pensada específicamente para:

- Contextos **educativos** (docentes, alumnos, proyectos de aula).
- **Entrenadores** que buscan una herramienta sencilla pero flexible para construir, guardar y explicar jugadas.
- **Estudiantes** de informática o tecnología interesados en sistemas interactivos visuales.

Su principal valor añadido reside en:

- La **creación paso a paso de jugadas dinámicas**, no solo disposición estática.
- La **posibilidad de reproducir secuencias de movimiento**, con visualización progresiva.
- La **persistencia de datos mediante JSON**, permitiendo compartir y reutilizar jugadas.
- El uso de tecnologías estándar y accesibles (Java, JavaFX, Scene Builder), sin necesidad de licencias comerciales.
- La implementación de **herramientas de edición avanzadas**, como deshacer/rehacer y escalado automático.

En resumen, *MiPizarra* se posiciona como una alternativa intermedia entre los proyectos limitados sin interactividad real y las plataformas profesionales cerradas, aportando un equilibrio entre funcionalidad, libertad técnica y accesibilidad.

2. Solución propuesta, Arquitectura y Diseño

Solución propuesta, Arquitectura y Diseño recoge los aspectos estructurales y formales del sistema *MiPizarra*, desde la arquitectura general hasta el diseño, implementación y validación de sus componentes. Se documentan las decisiones técnicas adoptadas, los modelos utilizados y la organización modular que permite que el sistema sea escalable, mantenible y adaptable a futuras ampliaciones. Esta parte tiene como objetivo principal proporcionar una visión clara del funcionamiento interno de la aplicación, permitiendo su comprensión, evaluación o modificación por parte de desarrolladores o docentes con perfil técnico.

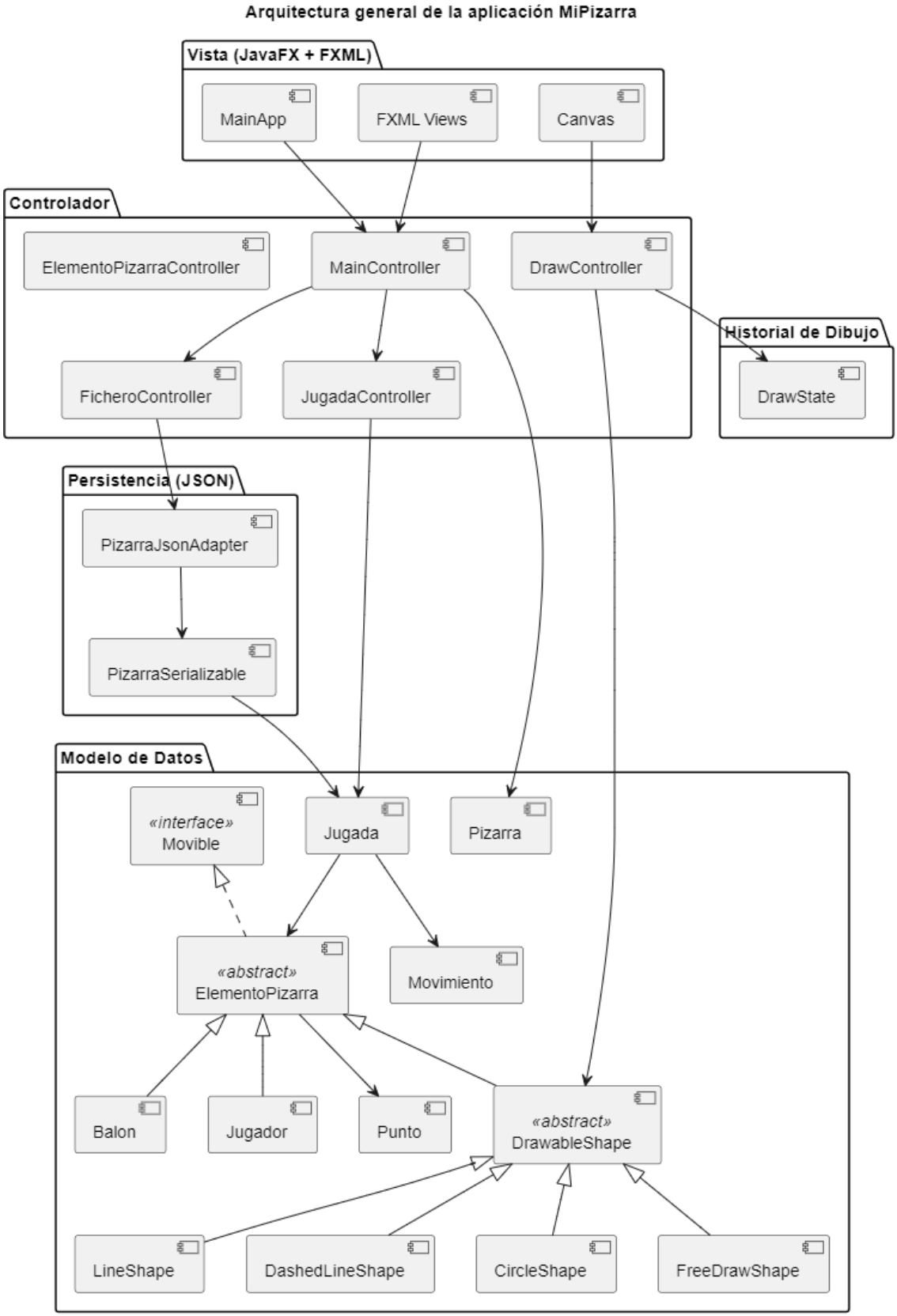
2.1. Visión general y arquitectura

La aplicación *MiPizarra* ha sido diseñada bajo un enfoque modular que favorece la separación de responsabilidades y facilita el mantenimiento del sistema. Se estructura principalmente en tres capas lógicas:

- **Capa de presentación (Vista)**: construida con JavaFX y FXML, es responsable de la interacción con el usuario y del renderizado visual de los elementos sobre la cancha. La interfaz se adapta dinámicamente al tamaño de la ventana mediante mecanismos de reescalado gráfico.
- **Capa de lógica y control**: implementada en clases como MainController, DrawController y FicheroController , esta capa se encarga de gestionar los eventos de usuario, controlar el flujo de ejecución, almacenar el estado de la aplicación y coordinar las distintas funcionalidades (dibujo, movimiento, jugadas, etc.).
- **Capa de datos y modelo**: conformada por clases como DrawableShape, LineShape, Jugador, Jugada y sus derivados. Representa los objetos que se manipulan en el sistema y proporciona una interfaz coherente para su renderizado, movimiento y almacenamiento.

Esta arquitectura está inspirada en el patrón **MVC (Modelo-Vista-Controlador)**, aunque adaptada de forma flexible a las necesidades de una aplicación gráfica interactiva. A ello se suma el uso de mecanismos propios para gestionar acciones (historial de dibujo con DrawState, pilas de undo/redo) y la serialización con **Jackson** para persistir jugadas como archivos JSON.

Figura 3: Arquitectura general.



2.2. Dificultades técnicas encontradas

Durante el desarrollo de *MiPizarra* se han presentado diversos retos técnicos que han exigido una reflexión continua sobre la arquitectura, las decisiones de diseño y la implementación concreta de algunas funcionalidades clave. A continuación se detallan los principales problemas técnicos afrontados y las soluciones adoptadas:

Gestión del escalado dinámico
Uno de los principales desafíos fue conseguir que todos los elementos gráficos (jugadores, balón, líneas, zonas) mantuvieran su proporción y posición relativa al redimensionar la ventana. Se optó por calcular coordenadas relativas al tamaño del campo base, y aplicar transformaciones inversas para mantener la precisión.

Separación entre dibujo libre y elementos interactivos
Fue necesario diseñar una arquitectura donde los trazos gráficos (líneas, flechas) no interfirieran con los elementos interactivos (jugadores, balón). Para ello se separaron en capas lógicas distintas y se estableció un control estricto del foco y del orden de renderizado.

Implementación del historial de dibujo
La creación del sistema de deshacer y rehacer supuso un reto importante. Fue necesario crear una clase DrawState capaz de clonar el estado completo del lienzo, e implementar pilas de historial (undoStack, redoStack) con gestión de límites y sincronización tras cada operación.

Reproducción secuencial de jugadas
Diseñar un sistema que almacenara movimientos paso a paso y permitiera reproducirlos visualmente de forma coherente implicó modelar una estructura de jugada con índices de reproducción, listas de movimientos y control del flujo de avance/retroceso sin pérdida de estado.

Gestión de persistencia JSON con Jackson
Serializar objetos que contenían referencias a imágenes, coordenadas relativas, tipos específicos y listas anidadas supuso una complejidad adicional. Fue necesario crear clases adaptadoras (PizarraSerializable) y aplicar anotaciones para controlar el proceso de serialización y evitar errores.

Sincronización entre Scene Builder, JavaFX y recursos
Garantizar que todos los componentes FXML funcionaran correctamente

en combinación con los recursos gráficos (imágenes, iconos) requirió una estructura de carpetas muy cuidada, rutas relativas precisas y una configuración adecuada del entorno de desarrollo en NetBeans.

Estas dificultades no solo representaron barreras técnicas, sino también oportunidades de aprendizaje clave durante el desarrollo. Su resolución ha fortalecido la robustez y madurez del sistema, y ha permitido implementar funcionalidades avanzadas sin comprometer la estabilidad general de la aplicación.

2.3. Tecnologías y Herramientas Utilizadas

El desarrollo de *MiPizarra* ha requerido la incorporación de múltiples tecnologías y herramientas que han contribuido tanto a la implementación funcional como al diseño visual, la gestión del proyecto y la organización del código. A continuación, se detallan los lenguajes, frameworks, librerías y productos de terceros utilizados a lo largo del proceso de desarrollo.

2.3.1. Lenguajes y frameworks de desarrollo

- **Java**
  Lenguaje principal empleado para toda la lógica de negocio, gestión de eventos, modelo de datos y control de flujo. Su orientación a objetos, robustez y portabilidad lo convierten en una opción ideal para aplicaciones gráficas multiplataforma.

- **JavaFX**
  Biblioteca utilizada para la creación de interfaces gráficas modernas. Permite dibujar sobre un lienzo (Canvas), capturar eventos del usuario y aplicar estilos mediante CSS. Su integración con animaciones y su arquitectura modular fueron claves para representar jugadas tácticas dinámicas.

- **FXML**
  Lenguaje declarativo basado en XML que facilita la construcción estructurada de interfaces gráficas. Se empleó para definir vistas separadas de la lógica de control, mejorando la organización y mantenimiento del código.

- **ApacheMaven**
  Sistema de gestión de proyectos y dependencias. Su uso permitió mantener una estructura clara, automatizar compilaciones y facilitar la integración con bibliotecas externas como Jackson y JavaFX.

2.3.2. Librerías auxiliares y componentes internos

- **JSON**
  Formato estándar empleado para la persistencia de jugadas. Permite guardar de forma estructurada las posiciones, trayectorias y elementos interactivos, facilitando su recuperación y edición en futuras sesiones.
- **Jackson**
  Librería de serialización/deserialización utilizada para convertir objetos Java a archivos .json y viceversa. Su integración simplificó el almacenamiento de jugadas y aseguró compatibilidad incluso con versiones anteriores mediante la anotación @JsonIgnoreProperties.
- **DrawState**
  Sistema propio de gestión de historial de acciones. Cada modificación gráfica se encapsula como un objeto DrawState, lo que permite implementar las funcionalidades de deshacer (undo) y rehacer (redo) de forma segura y sin pérdida de información.
- **JavaFXAnimations**
  Se han integrado animaciones como Timeline, FadeTransition y ParallelTransition para mejorar la interacción visual. Estas animaciones suavizan transiciones en la interfaz y sientan las bases para una reproducción visual fluida de jugadas completas.

2.3.3. Herramientas de desarrollo y diseño

- **NetBeansIDE**
  Entorno de desarrollo utilizado durante todo el proyecto. Su compatibilidad con Java, JavaFX, FXML, Maven y Git permitió una gestión integral del ciclo de desarrollo: desde la edición de código y depuración, hasta la compilación y ejecución del proyecto.
- **SceneBuilder**
  Herramienta visual que facilita la creación de interfaces JavaFX mediante FXML. Gracias a ella fue posible definir interfaces gráficas sin codificación

manual de layouts, acelerando el diseño y garantizando una separación clara entre lógica y presentación.

- **Git/GitHub**
  Se utilizó Git para el control de versiones local, permitiendo experimentar sin riesgo, retroceder cambios y mantener un historial ordenado. GitHub sirvió como repositorio remoto, facilitando el respaldo del proyecto y su posible colaboración futura.

2.3.4. Recursos gráficos y edición multimedia

- **MicrosoftPhotoshop**
  Utilizado para diseñar y adaptar los elementos visuales de la interfaz, como jugadores, el balón o la pista. Su capacidad para trabajar con capas y formatos editables (PSD) permitió una mayor versatilidad en la creación de recursos personalizados.

- **PNG/PSD**
  Todos los elementos gráficos fueron exportados en formato PNG para garantizar una carga rápida y compatibilidad multiplataforma. En algunos casos, se mantuvieron versiones PSD por capas para facilitar futuras modificaciones o adaptaciones visuales.

- **GanttProject**
  Se empleó para diseñar y gestionar la planificación temporal del proyecto. Esta herramienta permitió estructurar las fases de trabajo, asignar recursos, establecer dependencias entre tareas y generar un diagrama de Gantt completo, lo que facilitó la visualización del progreso y los plazos durante todo el desarrollo.
- **PlantUML**
  Herramienta de modelado basada en texto que permitió generar diagramas UML de forma rápida, reproducible y editable. Fue utilizada para representar tanto la estructura de clases del sistema como los principales flujos funcionales a través de diagramas de secuencia. Su integración en el entorno de desarrollo permitió mantener la documentación técnica sincronizada con el código.

3. Conclusiones y Trabajo Futuro

El presente capítulo recoge una reflexión general sobre el trabajo realizado, las aportaciones técnicas conseguidas, las dificultades superadas y las posibles líneas de mejora futuras. A lo largo del desarrollo de *MiPizarra* se han abordado múltiples aspectos del diseño e implementación de una herramienta visual para la representación táctica en deportes colectivos, integrando conceptos de interfaz gráfica, modelado de datos, control de estados, edición visual y persistencia.

La evaluación final de este proyecto se realiza tanto desde una perspectiva técnica como personal, considerando el cumplimiento de los objetivos planteados, el valor añadido de las funcionalidades implementadas, y la viabilidad de continuar su evolución en próximos desarrollos o contextos reales de uso.

3.1. Principales Problemas

Durante el desarrollo de *MiPizarra* se identificaron y resolvieron diversos problemas técnicos que afectaban a funcionalidades clave de la aplicación. A continuación, se describen las principales dificultades encontradas y las soluciones implementadas para garantizar un funcionamiento robusto y una experiencia de usuario fluida.

3.1.1. Escalado incorrecto de elementos gráficos

Uno de los retos más significativos fue lograr que todos los elementos de la pizarra (líneas, trayectorias, zonas, jugadores) se adaptaran correctamente al tamaño de ventana en diferentes resoluciones. Inicialmente, los objetos mantenían coordenadas absolutas que no respondían a los cambios de tamaño, provocando desalineaciones y pérdida de proporciones.

Solución:
Se implementó un sistema de reescalado dinámico basado en los factores scaleX y scaleY, calculados a partir de las dimensiones actuales del lienzo en comparación con su tamaño base. Todas las clases derivadas de DrawableShape fueron modificadas para utilizar estos factores en sus métodos draw(), y se ajustó también el grosor de las líneas (strokeWidth) de forma proporcional. Esto garantizó una visualización coherente y escalable en cualquier contexto de uso (pantallas pequeñas, proyectores, pizarras digitales).

3.1.2. Flechas mal proporcionadas y errores por clics rápidos

En la herramienta de dibujo de líneas con flecha y líneas de bloqueo, surgieron dos problemas recurrentes:

- Las flechas se mostraban desproporcionadas en ventanas pequeñas.
- Los clics rápidos generaban líneas incompletas o con errores de renderizado.

Solución:
Se modificó el método drawArrowHead(...) para que adaptara el tamaño de la flecha al contexto gráfico mediante un escalado relativo. Para los eventos de entrada, se mejoró el control de ratón filtrando correctamente las acciones en onMouseReleased, evitando así registros defectuosos por interacciones rápidas.

3.1.3. Pérdida de movimientos en la gestión de jugadas

En versiones tempranas del sistema, los desplazamientos de elementos sobre el campo no se almacenaban adecuadamente. Esto impedía construir secuencias de jugadas y reproducirlas paso a paso, reduciendo la utilidad del sistema como herramienta didáctica o táctica.

Solución:
Se diseñó la clase Jugada.java como contenedor de una lista ordenada de movimientos (fotogramas), representando cada acción realizada sobre los elementos. Esta lista permite recorrer las posiciones hacia adelante y atrás, reiniciar desde el principio y, en el futuro, añadir reproducción automática. La introducción de esta arquitectura modular supuso un avance decisivo hacia una funcionalidad de jugadas dinámica y reutilizable.

3.1.4. Problemas de persistencia en el guardado y carga de jugadas

Al implementar la opción de guardar jugadas en formato JSON, se identificaron dificultades para serializar objetos complejos y para mantener compatibilidad ante posibles cambios en la estructura interna del modelo.

Solución:
Se integró la librería Jackson, usando anotaciones como @JsonIgnoreProperties(ignoreUnknown = true) para asegurar la tolerancia a versiones antiguas. Además, se encapsuló toda la lógica de persistencia en la clase FicheroController.java, garantizando un aislamiento limpio entre modelo y

almacenamiento. Esto permitió guardar jugadas completas, cargarlas correctamente y reiniciarlas sin afectar la estabilidad de la aplicación.

3.2. Principales aportaciones

El desarrollo de *MiPizarra* ha dado lugar a una serie de aportaciones significativas en el ámbito de las herramientas gráficas para la representación táctica deportiva. Entre las funcionalidades implementadas y mejoras logradas destacan:

- **Sistema de Gestión de Jugadas Dinámicas**: Se ha diseñado una arquitectura que permite construir jugadas paso a paso mediante la captura de movimientos secuenciales. Esto habilita una simulación táctica visual, ideal para entrenadores, docentes y estudiantes.
- **Dibujo Táctico Multitipo**: La aplicación incorpora trazado libre, líneas con flecha, líneas de bloqueo y círculos, ofreciendo múltiples opciones para representar tácticas complejas.
- **Gestión del Historial de Dibujo**: Se ha desarrollado un sistema de deshacer y rehacer acciones mediante pilas de estados (undoStack y redoStack), lo que permite corregir errores fácilmente sin perder el trabajo realizado.
- **Persistencia con JSON**: Se ha implementado el guardado y carga de jugadas utilizando el formato JSON y la librería Jackson, permitiendo reutilizar jugadas, compartirlas y editarlas externamente si es necesario.
- **Escalado Inteligente de Elementos**: Los elementos de la pizarra se adaptan automáticamente a cualquier tamaño de ventana, manteniendo la proporcionalidad y ubicación relativa.
- **Animaciones y Transiciones en JavaFX**: Se han incorporado transiciones suaves y efectos visuales que mejoran la experiencia de uso, y sientan la base para una futura reproducción animada de jugadas.

3.3. Conclusiones

Desde un punto de vista técnico, el desarrollo de *MiPizarra* ha supuesto un proceso completo que ha abarcado desde la definición de requisitos hasta la implementación, pruebas y documentación de un sistema interactivo. La elección de tecnologías como Java, JavaFX, FXML y Jackson ha permitido construir una solución modular, escalable y bien estructurada.

Uno de los mayores retos ha sido la gestión de eventos gráficos y la integración de funcionalidades complejas como la reproducción de jugadas o el historial de acciones de dibujo. La experiencia ha consolidado conocimientos en programación orientada a objetos, diseño de interfaces gráficas, manejo de estructuras de datos y control de versiones con Git.

En el plano personal, el proyecto ha permitido adquirir experiencia en la planificación y ejecución de un sistema completo, así como en la toma de decisiones técnicas, resolución de errores no previstos y mejora progresiva del producto. El trabajo ha fomentado la autonomía, la adaptabilidad y el aprendizaje autodidacta de herramientas como Scene Builder, JSON o animaciones en JavaFX.

3.4. Vías de trabajo futuro

De cara al futuro, existen diversas líneas de mejora y expansión que podrían dotar a *MiPizarra* de mayor funcionalidad y valor añadido:

- **Edición y Gestión Avanzada de Jugadas**: Incorporar opciones para renombrar, duplicar, editar pasos intermedios o insertar nuevos movimientos en jugadas ya existentes.
- **Exportación Multiformato**: Permitir exportar jugadas como imágenes, secuencias de vídeo o presentaciones para su uso en clases, partidos o redes sociales.
- **Soporte Multideporte**: Adaptar la interfaz para representar otros deportes colectivos (fútbol, balonmano, hockey...) mediante plantillas de pista y configuraciones específicas.
- **Etiquetas y Clasificación de Jugadas**: Añadir metadatos como tipo de jugada, categoría, nivel o fecha, para facilitar la búsqueda y organización.
- **Compatibilidad Multiplataforma**: Explorar alternativas para portar la aplicación a sistemas móviles (Android/iOS) o web, utilizando frameworks como JavaFXPorts o tecnologías web híbridas.

Estas mejoras no solo ampliarían las capacidades actuales de *MiPizarra*, sino que permitirían su aplicación real en entornos educativos y deportivos, acercando el sistema a estándares profesionales de análisis y comunicación táctica

4. Especificación de Requisitos

En esta sección se describen los requisitos que debe satisfacer la aplicación *MiPizarra* para cumplir con sus objetivos funcionales y de calidad. La especificación se divide en dos grandes grupos: **requisitos funcionales**, que definen las capacidades que debe ofrecer el sistema, y **requisitos no funcionales**, que detallan condiciones técnicas, de rendimiento y de portabilidad. Asimismo, se presentan algunos **casos de uso** representativos que ilustran la interacción esperada entre el usuario y el sistema.

4.1. Requisitos funcionales

Los requisitos funcionales describen las acciones que el sistema debe ser capaz de ejecutar, y se han identificado mediante un análisis iterativo de las necesidades del usuario y de las funcionalidades diseñadas:

- **RF01**: El sistema permitirá al usuario representar elementos tácticos sobre un lienzo mediante herramientas de dibujo: líneas con flecha, líneas de bloqueo, trayectorias discontinuas, zonas y círculos.
- **RF02**: El usuario podrá interactuar con objetos activos sobre la pista, como jugadores y balón, mediante operaciones de colocación, selección, movimiento y eliminación.
- **RF03**: El sistema registrará automáticamente los desplazamientos realizados cuando esté activo el modo “Jugada”, organizando los movimientos en una secuencia estructurada.
- **RF04**: La jugada registrada podrá reproducirse paso a paso mediante controles de navegación o de forma automática con temporizador.
- **RF05**: El usuario podrá guardar la jugada activa en un archivo con formato .json, manteniendo tanto los elementos como su secuencia de movimientos.
- **RF06**: El sistema permitirá la carga de jugadas previamente guardadas, restaurando el estado de la pizarra y su dinámica asociada.
- **RF07**: La jugada activa podrá reiniciarse, eliminando los movimientos registrados y restaurando el estado inicial del lienzo.
- **RF08**: Los elementos gráficos (formas, jugadores, zonas) se escalarán proporcionalmente al redimensionar la ventana o modificar la resolución.

- **RF09**: Se proporcionará un sistema de gestión del historial de acciones gráficas, permitiendo al usuario deshacer y rehacer trazos manuales.
- **RF10**: La interfaz gráfica ofrecerá controles accesibles e intuitivos, organizados en una barra superior y paneles laterales.

4.2. Requisitos no funcionales

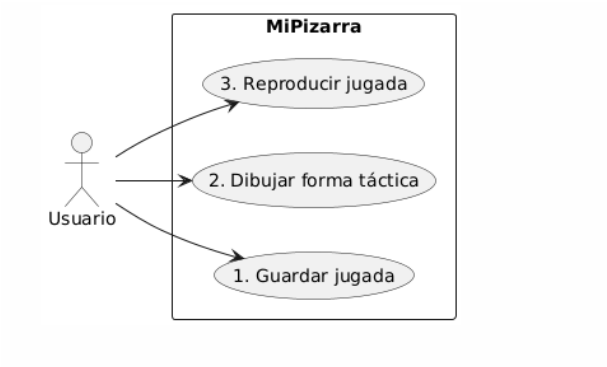
Los requisitos no funcionales establecen propiedades deseables del sistema en términos de usabilidad, compatibilidad, rendimiento y calidad del código:

- **RNF01**: La aplicación deberá ser multiplataforma, ejecutándose correctamente en sistemas Windows, Linux y macOS.
- **RNF02**: El almacenamiento de datos se realizará mediante archivos JSON, garantizando legibilidad, ligereza y compatibilidad con herramientas externas.
- **RNF03**: La interfaz de usuario deberá adaptarse dinámicamente a distintas resoluciones sin pérdida de alineación ni proporciones gráficas.
- **RNF04**: La aplicación deberá responder de forma inmediata a las acciones del usuario, minimizando latencias en operaciones de dibujo o reproducción.
- **RNF05**: La arquitectura del sistema se organizará siguiendo principios de modularidad y separación de responsabilidades, facilitando su mantenibilidad y extensibilidad.

4.3. Casos de uso

A continuación, se presentan tres casos de uso representativos que ilustran interacciones clave entre el usuario y la aplicación:

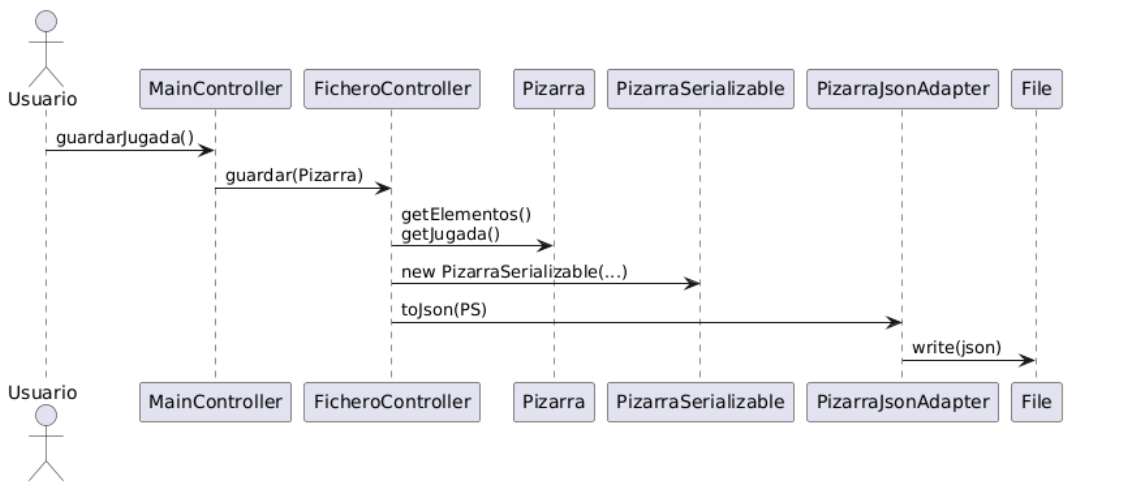
Figura 4: Diagrama de Casos de Uso.



Caso de Uso 1: Guardar jugada

- *Actor*: Usuario
- *Objetivo*: Almacenar en disco la jugada actualmente activa.
- *Precondición*: Existe una jugada registrada.
- *Flujo principal*: El usuario accede al menú de guardado, selecciona una ubicación y un nombre de archivo. El sistema serializa la jugada y la almacena en formato JSON.
- *Resultado esperado*: El archivo es generado y queda disponible para futuras sesiones.

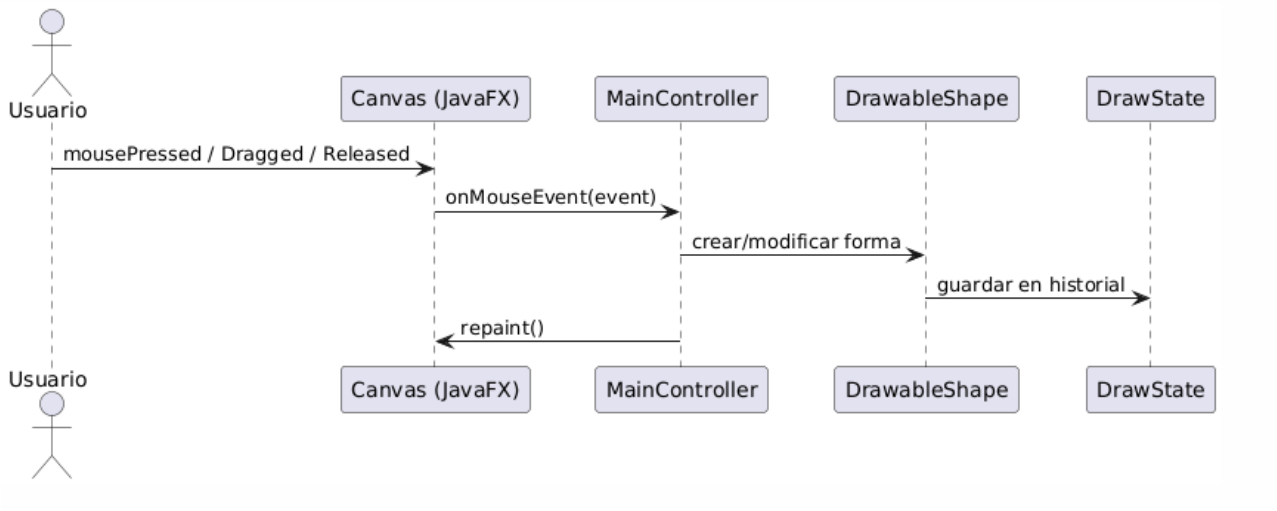
Figura 5: Diagrama de Guardar Jugada.



Caso de Uso 2: Dibujar forma táctica

- *Actor*: Usuario
- *Objetivo*: Representar visualmente una acción táctica mediante una línea o forma.
- *Flujo principal*: El usuario selecciona el tipo de forma y traza un segmento sobre la pista. El sistema interpreta la acción, valida restricciones (por ejemplo, si es una línea de bloqueo) y añade el objeto al lienzo.
- *Resultado esperado*: La forma es visible en la interfaz con el estilo seleccionado.

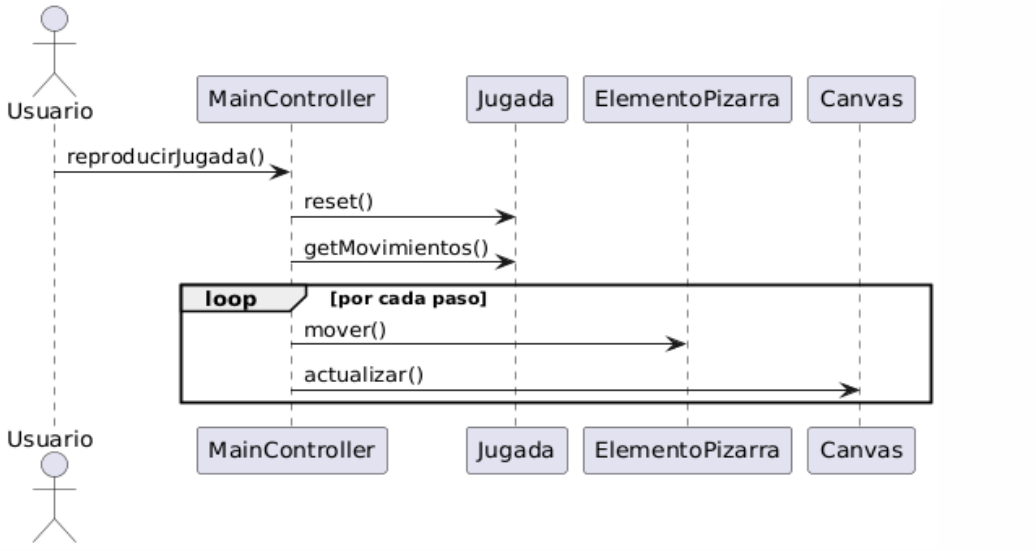
Figura 6: Diagrama de Dibujar forma táctica.



Caso de Uso 3: Reproducir jugada

- *Actor*: Usuario
- *Objetivo*: Visualizar el desarrollo de una jugada secuenciada.
- *Flujo principal*: El usuario pulsa el botón de reproducción. El sistema itera sobre los elementos y aplica las transformaciones correspondientes según la velocidad seleccionada.
- *Resultado esperado*: Los elementos se mueven conforme a la jugada registrada, simulando el desarrollo dinámico.

Figura 7: Diagrama de Reproducir Jugada.



5. Análisis

El análisis del sistema tiene como objetivo descomponer las funcionalidades requeridas en **comportamientos observables** mediante la interacción entre componentes. Para ello, se ha empleado modelado UML a través de **diagramas de secuencia**, que permiten ilustrar el flujo de control entre clases, interfaces gráficas y controladores del sistema, sin entrar todavía en los detalles de implementación interna.

En el caso de *MiPizarra*, se han identificado tres procesos funcionales esenciales que requieren coordinación entre múltiples objetos: la reproducción de jugadas, el dibujo de elementos tácticos, y el registro dinámico de movimientos. Cada uno de ellos se analiza mediante un diagrama de secuencia específico que refleja la lógica temporal del sistema.

5.1. Análisis funcional

La aplicación *MiPizarra* nace con el objetivo de permitir a entrenadores, docentes y estudiantes representar jugadas deportivas sobre un campo virtual de forma gráfica, interactiva y secuencial. Para ello, se ha realizado un análisis funcional que descompone el sistema en funcionalidades concretas, agrupadas por su objetivo principal: edición visual, modelado táctico, gestión de jugadas y persistencia.

A partir de los requisitos definidos en el capítulo 3, se han identificado los siguientes bloques funcionales principales:

Edición visual del campo de juego: El usuario debe poder añadir elementos (jugadores, balón), modificar su posición, y trazar líneas de movimiento o zonas. El sistema distingue entre elementos “movibles” (actúan dentro de la jugada) y elementos “dibujados” (forman parte del esquema gráfico).

Control de jugadas paso a paso: Cada vez que un elemento se mueve, el sistema registra la posición en un paso de jugada. Posteriormente, el usuario puede avanzar y retroceder por estos pasos, visualizando la jugada de forma dinámica. Este comportamiento simula la evolución real de una táctica sobre el campo.

Gestión de acciones gráficas (historial de dibujo): Las acciones de dibujo (líneas, flechas, zonas) pueden ser revertidas o repetidas mediante deshacer y rehacer, utilizando una arquitectura basada en DrawState. Esto permite una edición libre pero controlada.

Persistencia de jugadas: Las jugadas pueden ser guardadas y recuperadas mediante archivos .json, manteniendo tanto la disposición de los elementos como su secuencia de movimientos. Se utiliza la librería Jackson para la serialización/deserialización.

Escalado automático: El sistema ajusta automáticamente las posiciones y dimensiones de todos los elementos cuando el tamaño de la ventana cambia, garantizando coherencia visual sin distorsión.

Interfaz gráfica modular: La aplicación se divide en múltiples pantallas y componentes, todos integrados mediante FXML y Scene Builder. Esto permite mantener una separación clara entre lógica, presentación y eventos.

Este análisis funcional ha guiado la implementación de cada una de las funcionalidades descritas en el apartado 4.2, asegurando que todas ellas respondan a necesidades reales detectadas en el estado del arte y en los requisitos definidos inicialmente.

5.2. Organización de paquetes

El proyecto está estructurado en paquetes que agrupan las clases según su funcionalidad:

- controller: controladores principales de la interfaz y de la lógica de usuario.
- model: clases de dominio que definen las entidades (jugadas, movimientos, elementos).
- shapes: jerarquía de formas dibujables (líneas, círculos, trayectorias).
- util: clases auxiliares como DrawState y herramientas gráficas.
- resources: imágenes, estilos CSS y archivos FXML.

Responsabilidades principales por módulo

Módulo / Clase	Función principal
MainController	Inicializa la interfaz y gestiona los eventos generales
DrawController	Lógica de dibujo libre, selección de herramientas y gestión de trazos
Jugada	Almacena y controla la reproducción de una jugada paso a paso
FicheroController	Guarda y carga jugadas en formato JSON
LineShape, CircleShape, etc.	Representan gráficamente líneas, zonas y otros elementos sobre el campo
DrawState	Encapsula estados del lienzo para deshacer y rehacer acciones

Esta estructura ha permitido un desarrollo organizado, con componentes reutilizables y bien delimitados. Asimismo, posibilita futuras mejoras, como integrar animaciones automáticas, exportaciones multimedia o ampliación a nuevos deportes sin alterar la base ya construida.

5.3. Diagrama de clases

En el caso de *MiPizarra*, se han identificado clases fundamentales como Jugada, ElementoPizarra, DrawableShape, DrawState, MainController y PizarraJsonAdapter, entre otras. Estas clases se organizan en capas lógicas con responsabilidades diferenciadas, que responden a una arquitectura modular basada en el patrón MVC (Modelo-Vista-Controlador). El diagrama facilita la comprensión del sistema desde una perspectiva de diseño técnico y es clave para futuras ampliaciones o mantenimiento del proyecto.

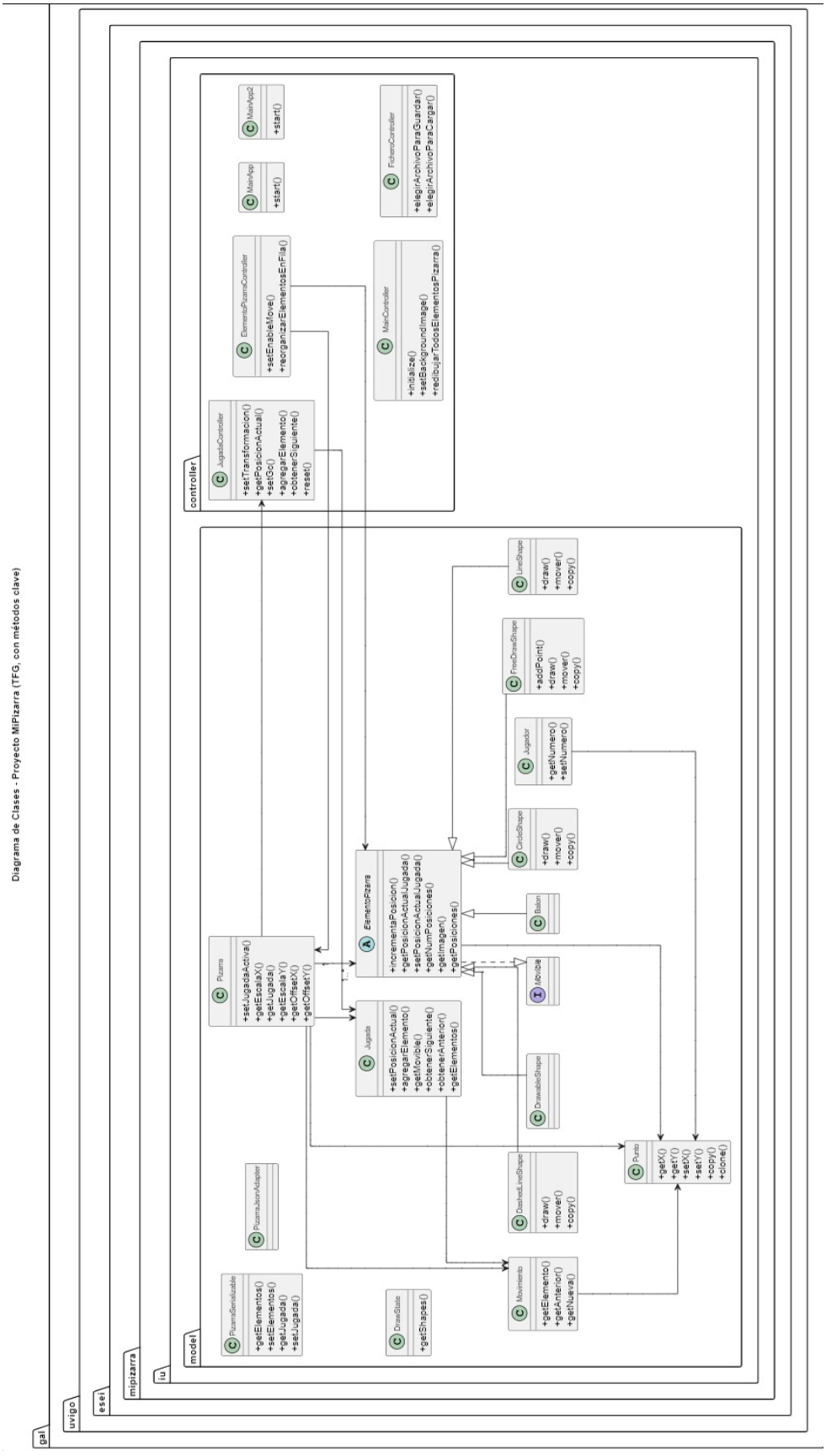
El siguiente diagrama representa la **estructura estática** del sistema y las relaciones entre sus componentes principales. Se ha diseñado para cumplir los principios SOLID y facilitar futuras ampliaciones como nuevos tipos de formas, animaciones, o integración web.

Componentes destacados:

- **Pizarra**: clase principal del modelo. Almacena los elementos (ElementoPizarra), la jugada activa (Jugada) y los factores de escala y desplazamiento del lienzo. Proporciona métodos de redibujado y recuperación de elementos.
- **ElementoPizarra** (abstracta): clase base para cualquier objeto que puede representarse gráficamente y ser animado. Define métodos como draw(), mover(), getPosiciones(), y mantiene el historial posicional.
    - Subclases: Jugador, Balon, CircleShape, LineShape, DashedLineShape, FreeDrawShape. Todas implementan draw() y mover() con lógica específica.
- **Jugada**: gestiona una secuencia ordenada de movimientos. Proporciona métodos para obtener el movimiento actual (getPosicionActual()), avanzar o retroceder, reiniciar y acceder a subconjuntos de la jugada.
- **Movimiento**: representa un desplazamiento de un objeto entre dos posiciones (posicionAnterior, nuevaPosicion). Asociado siempre a un ElementoPizarra.
- **DrawState**: encapsula una instantánea del estado gráfico del lienzo para habilitar operaciones de deshacer/rehacer. Gestionado como pila.
- **PizarraSerializable** y **PizarraJsonAdapter**: clases que permiten transformar el estado interno de la pizarra en una estructura JSON. Permiten desacoplar la lógica de persistencia del resto del modelo.
- **Controladores**:
    - MainController: inicializa la interfaz, escucha eventos globales.
    - FicheroController: gestiona los flujos de guardado y carga.
    - JugadaController y ElementoPizarraController: responsables de la lógica de movimiento, agrupación, y reproducción de jugadas.

Este diseño modular permite integrar nuevas clases sin romper la jerarquía existente, y soporta extensión de comportamiento por herencia.

Figura 8: Diagrama de Clases.



5.4. Diagramas de secuencia

Los diagramas de secuencia permiten describir de forma detallada cómo se produce la interacción entre objetos del sistema durante la ejecución de casos de uso concretos. Representan el flujo de mensajes entre instancias a lo largo del tiempo, mostrando el orden y los métodos implicados en cada operación.

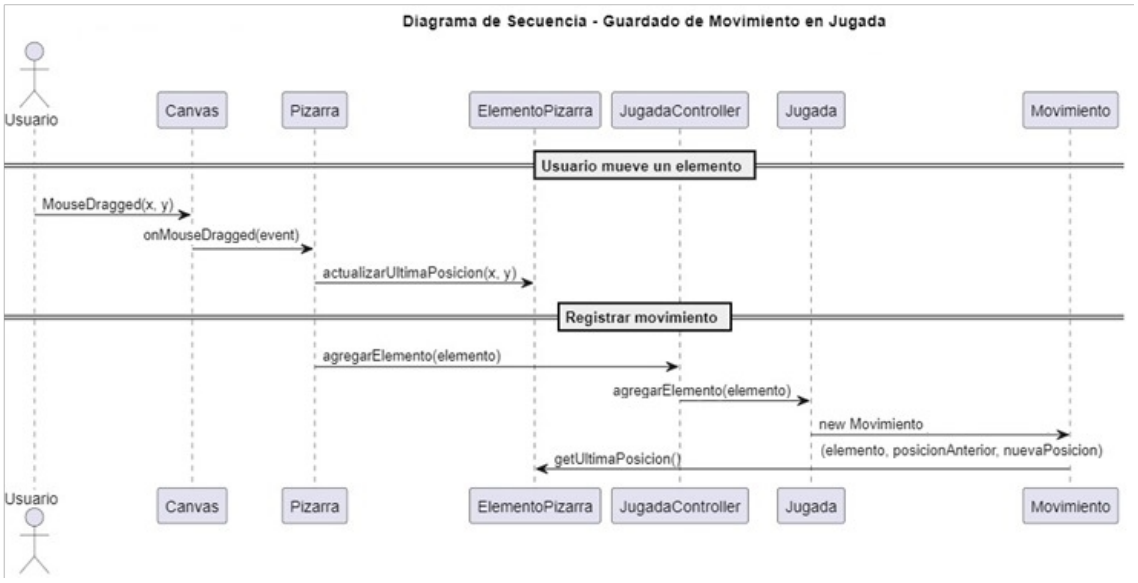
5.4.1. Registro de movimiento durante una jugada

Durante el uso del sistema, el usuario puede mover elementos sobre el campo mientras el modo “Jugada” está activo. Cada uno de estos movimientos genera un evento que es capturado por Pizarra, el cual delega la actualización posicional a los objetos ElementoPizarra.

Simultáneamente, se crea un nuevo objeto Movimiento, que encapsula la posición anterior y la nueva, y se registra en la instancia activa de Jugada. De esta forma, el sistema va construyendo progresivamente una estructura secuencial que permite la navegación paso a paso o la reproducción animada.

Este análisis demuestra la necesidad de un almacenamiento intermedio consistente y desacoplado entre vista y modelo.

Figura 9: Diagrama de Secuencia-Guardado de Movimiento en Jugada.



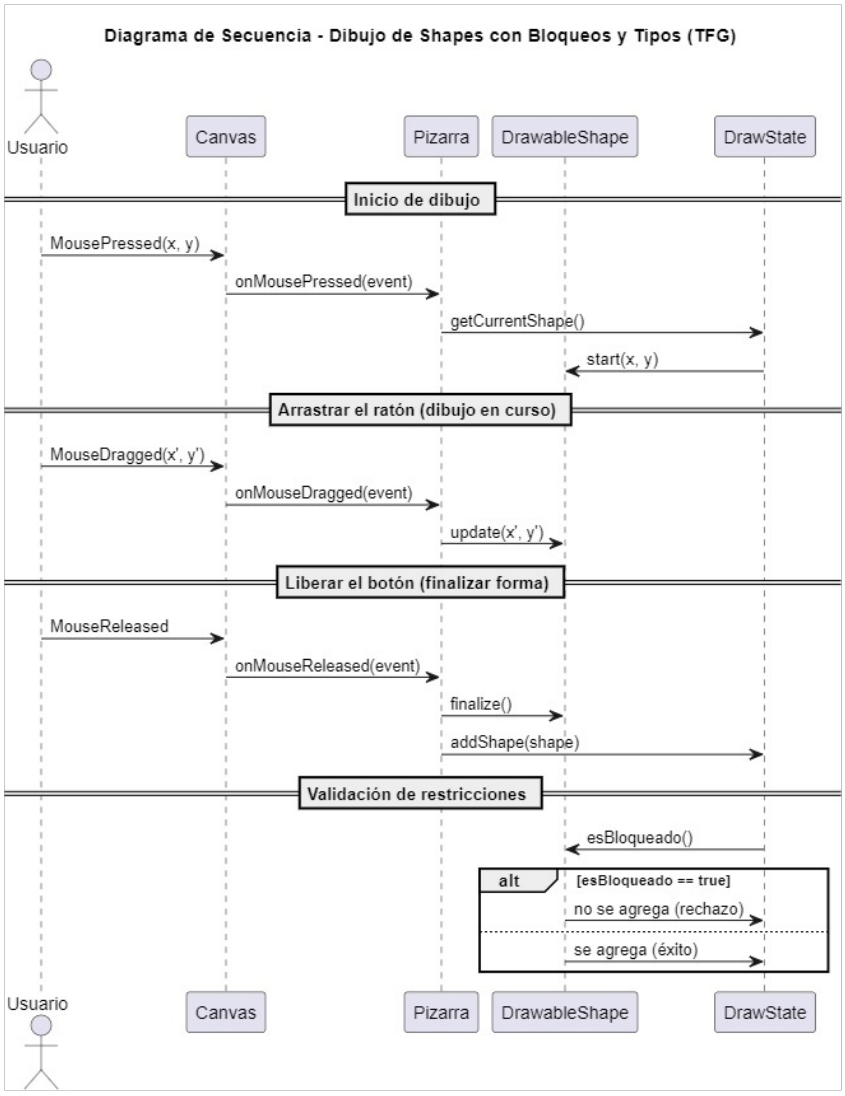
5.4.2. Dibujo de elementos tácticos

Cuando el usuario emplea las herramientas de dibujo, el flujo se activa desde la interfaz de Canvas, que detecta los eventos MousePressed, MouseDragged y MouseReleased. Estos son gestionados por Pizarra, que solicita al modelo gráfico (DrawableShape) que genere o actualice la figura correspondiente.

Al finalizar el trazo, se comprueba si la forma está sujeta a restricciones lógicas, como en el caso de una línea de bloqueo (con perpendicular). En caso afirmativo, se ejecuta una validación (esBloqueado()) que puede aceptar o rechazar el objeto antes de incorporarlo al historial mediante un DrawState.

Este análisis evidencia la separación clara entre lógica de entrada, procesamiento de forma y almacenamiento, característica fundamental para garantizar un entorno gráfico flexible y extensible.

Figura 10: Diagrama de Secuencia-Dibujo de Shapes con Bloqueos y Tipos.



5.4.3. Guardado de jugada (Serialización a JSON)

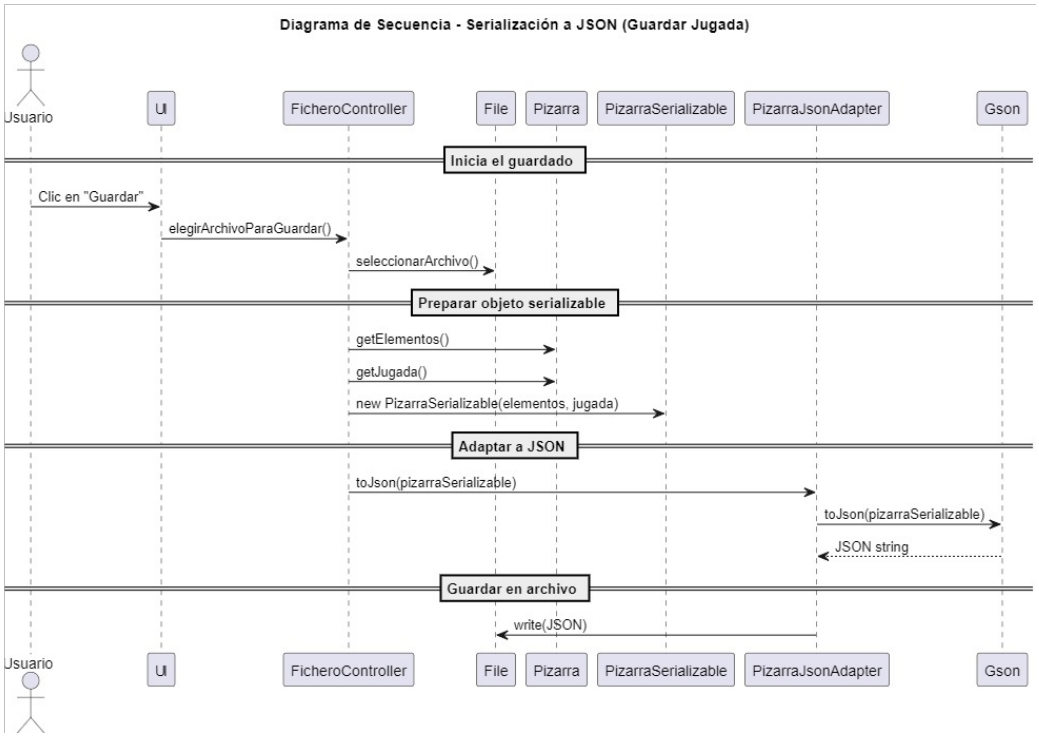
Este diagrama describe detalladamente el proceso de persistencia de una jugada activa. El flujo se inicia cuando el usuario selecciona la opción de guardar desde la interfaz gráfica.

Flujo detallado:

1. El FicheroController solicita al sistema un archivo de destino utilizando un diálogo nativo (elegirArchivoParaGuardar()).
2. Una vez seleccionado el archivo, se accede al modelo (Pizarra) para obtener los elementos gráficos y la jugada activa (getElementos() y getJugada()).
3. Se crea una instancia de PizarraSerializable, una clase intermedia que contiene solo los datos necesarios para exportar.
4. El objeto PizarraSerializable se transforma a texto JSON mediante PizarraJsonAdapter.toJson(...), usando internamente una librería como Gson o Jackson.
5. Finalmente, se escribe el resultado en disco mediante write (JSON).

Este diseño asegura separación entre lógica de negocio y formato de persistencia, lo que permite adaptar fácilmente el sistema a otros formatos (XML, bases de datos, etc.) si se desea en el futuro.

Figura 11: Diagrama de Secuencia-Serialización a JSON.



5.4.4. Grabación de jugadas dinámicas

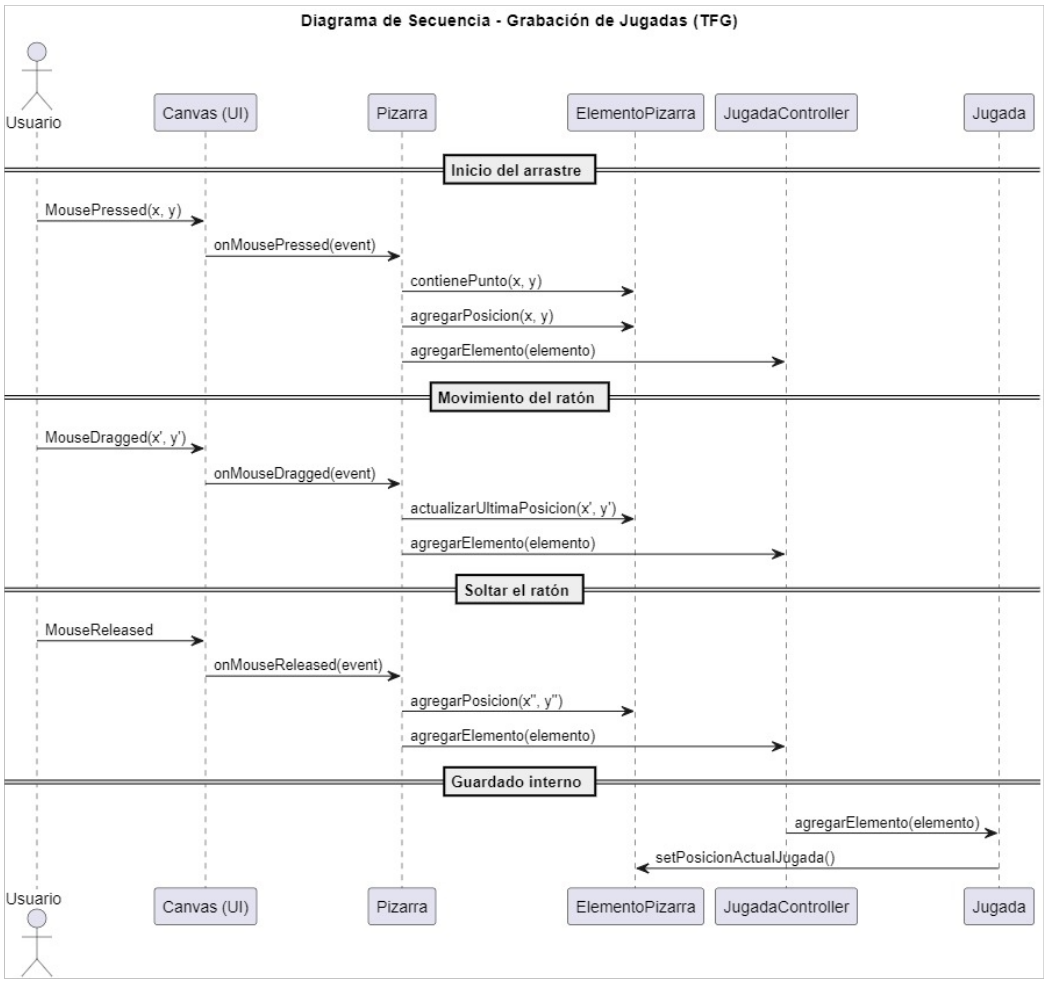
Este diagrama representa el proceso completo de registro de jugadas durante la interacción directa del usuario con la pizarra. Es uno de los flujos más complejos y críticos del sistema, ya que afecta tanto a la experiencia visual como a la estructura de datos.

Fases del flujo:

1. **Inicio del arrastre (MousePressed):**
    - El Canvas captura el evento y lo remite al controlador principal (onMousePressed(event)).
    - Se detecta si se ha presionado sobre un ElementoPizarra mediante contienePunto(x, y).
    - Si es así, se inicializa la jugada almacenando la primera posición (agregarPosicion(x, y)) y el elemento afectado.
2. **Movimiento del ratón (MouseDragged):**
    - Durante el desplazamiento, se actualiza la posición en tiempo real (actualizarUltimaPosicion(x', y')) y se almacena en la lista interna del ElementoPizarra.
3. **Liberación (MouseReleased):**
    - Al soltar el ratón, se registra el último punto y se actualiza la jugada activa mediante el método setPosicionActualJugada().
4. **Guardado interno:**
    - Todo el movimiento se encapsula como un nuevo Movimiento y se añade al historial de la jugada (Jugada.agregarElemento(...)).

Este flujo garantiza que la jugada capturada es fiel a la interacción del usuario, permitiendo su posterior reproducción exacta y edición.

Figura 12: Diagrama de Secuencia-Grabación de Jugadas.

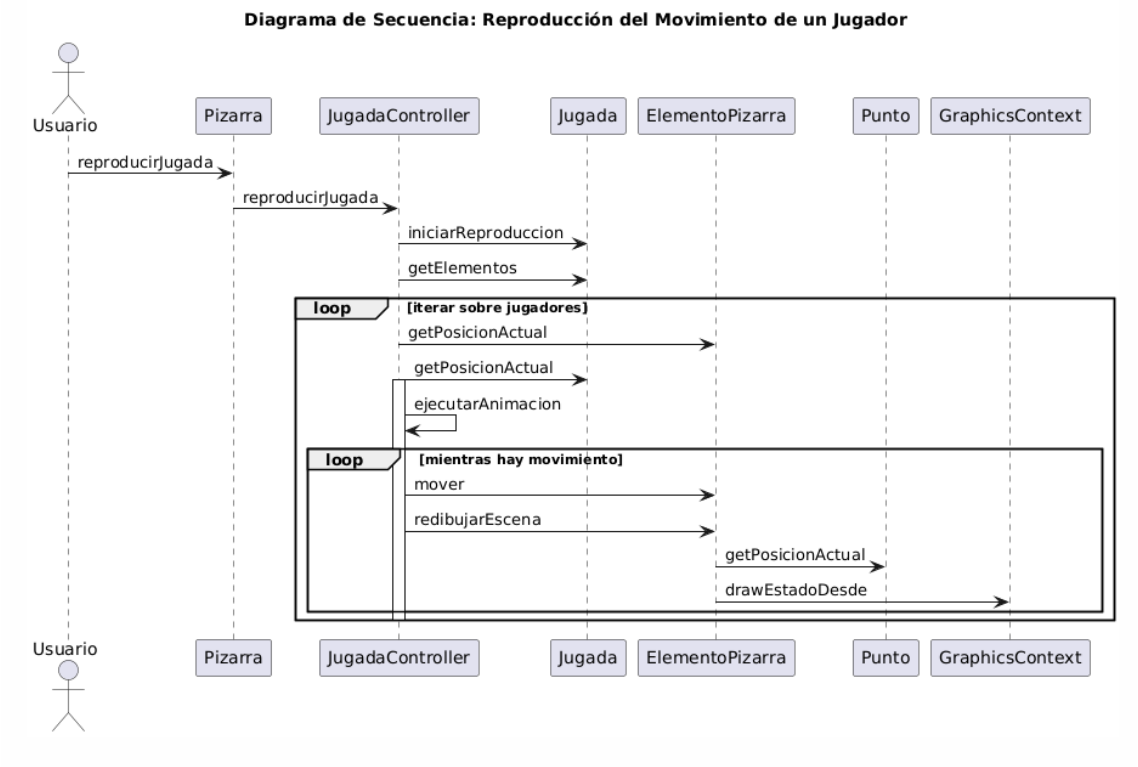


5.4.5. Reproducción de una jugada

Este flujo comienza cuando el usuario solicita la reproducción de una jugada previamente registrada. El componente Pizarra delega el control a JugadaController, que se encarga de acceder a los elementos involucrados (ElementoPizarra) y reiniciar su posición en la jugada (setPosicionActual 0)). A continuación, se ejecuta una animación temporal donde, en cada iteración, los elementos son movidos a través del método mover(...) que calcula su posición actual interpolada y solicita a la clase GraphicsContext que actualice su representación gráfica.

El sistema se encuentra preparado para permitir la configuración de velocidad y para interrumpir o reiniciar la reproducción desde cualquier punto de la secuencia, garantizando flexibilidad de uso tanto en sesiones formativas como en presentaciones tácticas.

Figura 13: Diagrama de Secuencia-Reproducción del Movimiento de un jugador.



5.5. Modelo de datos

El modelo de datos de *MiPizarra* define las estructuras clave necesarias para representar de forma precisa la información visual y lógica de una jugada táctica. Este modelo está orientado a objetos y organizado en torno a clases que encapsulan tanto la representación de los elementos sobre el lienzo como su evolución temporal.

A continuación, se describen las entidades más relevantes:

Jugada

La clase Jugada es el núcleo del modelo. Contiene una lista ordenada de pasos que representan los movimientos de los jugadores u objetos sobre el campo. También incluye información sobre el estado actual de la reproducción y los elementos disponibles en la jugada.

Atributos destacados:

- elementos: lista de elementos visuales (jugadores, balón...).
- posicionActual: índice que indica el punto de reproducción actual.
- reset(), avanzarPaso(), retrocederPaso(): métodos para controlar la jugada.

ElementoPizarra

Clase abstracta que representa cualquier elemento colocable en el campo. Sirve de base para Jugador, Balon, ZonaCircular, etc. Define la posición, tipo y recursos gráficos asociados al objeto.

Movimiento

Almacena la información de un cambio de posición o acción dentro de una jugada. Incluye:

- Identificador del elemento implicado.
- Coordenadas de origen y destino.
- Paso dentro de la secuencia.

Esto permite reconstruir la jugada paso a paso durante la reproducción.

DrawState

Clase que encapsula el estado del dibujo libre (líneas, flechas, zonas) en un momento determinado. Se utiliza para implementar las funcionalidades de deshacer y rehacer.

Atributos:

- List<DrawableShape> shapes: conjunto de formas trazadas en el lienzo.
- Métodos de clonación y restauración.

PizarraSerializable

Clase adaptadora utilizada para convertir el contenido de una jugada a formato JSON utilizando la librería Jackson. Permite separar la lógica de visualización del formato de persistencia, facilitando el guardado y carga de datos.

Este modelo de datos ha sido diseñado para ser flexible, extensible y fácilmente integrable con el sistema gráfico de JavaFX, permitiendo representar jugadas dinámicas con fidelidad visual y estructural.

---

6. Implementación

La implementación de *MiPizarra* ha seguido un enfoque modular, orientado a objetos y centrado en la claridad del código y la reutilización de componentes. A lo largo del desarrollo se han creado múltiples clases especializadas para gestionar tanto la lógica del sistema como su representación visual, manteniendo una separación entre el modelo de datos, la interfaz gráfica y el control de eventos.

En esta sección se describen las principales funcionalidades implementadas, detallando cómo se resolvieron los aspectos técnicos clave que dan soporte al dibujo táctico, la gestión de jugadas, la navegación por movimientos y el guardado en formato JSON. Además, se comentan las decisiones técnicas adoptadas y se ilustra cómo se integran las distintas partes del sistema para ofrecer una experiencia fluida y robusta.

6.1. Dibujar línea táctica

Una de las funcionalidades fundamentales del sistema es la posibilidad de representar visualmente una línea sobre la pista de juego. Esta acción permite al usuario reflejar desplazamientos de jugadores o instrucciones tácticas, ya sea mediante **líneas normales** (con flecha final) o **líneas de bloqueo** (con remate en perpendicular).

Funcionamiento general

El usuario selecciona el modo “línea” o “bloqueo” mediante los botones situados en la barra de herramientas. Posteriormente, traza un segmento con el ratón desde el punto inicial hasta el punto final sobre la superficie de la pizarra. Según el modo activo, el sistema dibuja:

- Una **flecha** que indica dirección de movimiento.
- Una **perpendicular** si se trata de un bloqueo.

Lógica interna

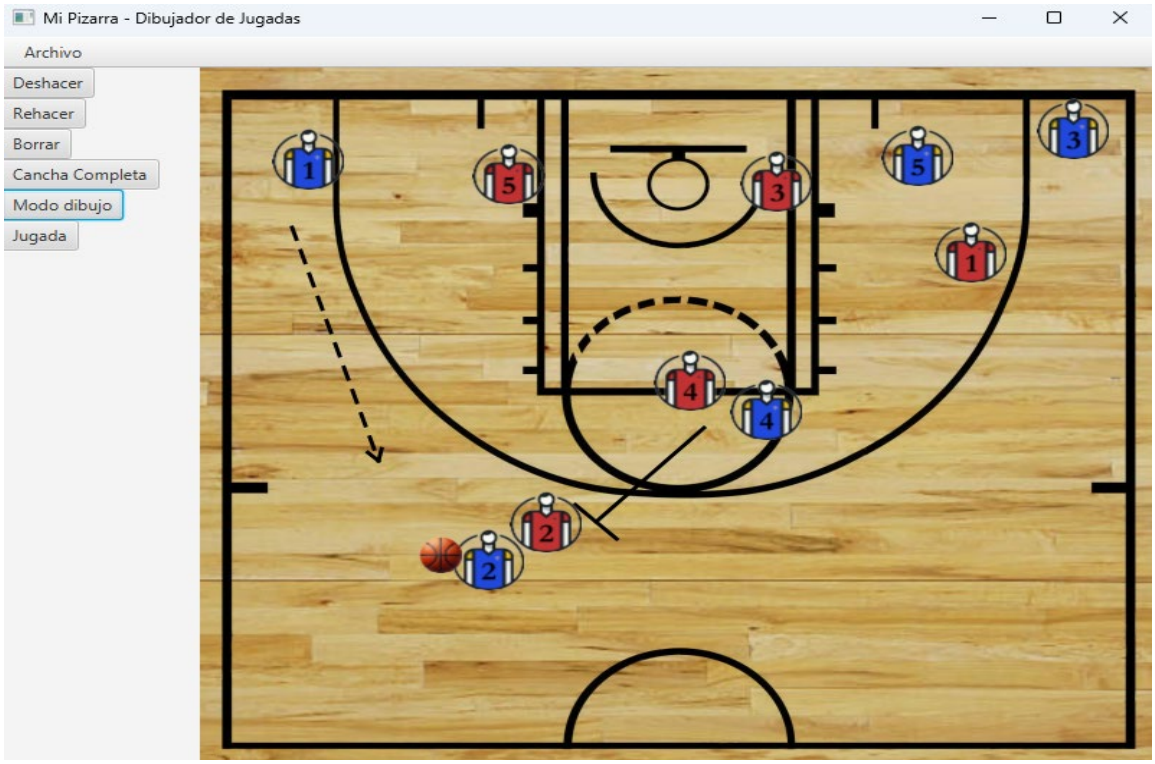
La funcionalidad se implementa en la clase LineShape, que hereda de DrawableShape. Este objeto almacena las coordenadas originales, el color, el grosor del trazo y un booleano que indica si se trata de un bloqueo:

```
public LineShape(double x1, double y1, double x2, double y2, Color color, double strokeWidth, boolean esBloqueo) {
    super(color, strokeWidth);
```

En el método draw(GraphicsContext gc, ...), se utiliza gc.strokeLine(...) para trazar el segmento y, en función del tipo de línea, se llama a drawArrowHead(...) o drawPerpendicular(...):

```
if (esBloqueo) {
    // Si es un bloqueo, dibujar una perpendicular en lugar de la flecha
    drawPerpendicular(gc, originalX2 * scaleX + offsetX, originalY2 * scaleY + offsetY,
                      originalX1 * scaleX + offsetX, originalY1 * scaleY + offsetY, scale);
} else {
    // Si no es un bloqueo, dibujar la flecha
    drawArrowHead(gc, originalX2 * scaleX + offsetX, originalY2 * scaleY + offsetY,
                  originalX1 * scaleX + offsetX, originalY1 * scaleY + offsetY, scale);
}
```

Figura 14: La siguiente imagen muestra dos líneas trazadas: una de movimiento con flecha, y otra de bloqueo con perpendicular.



Problemas encontrados

- Escalado incorrecto al cambiar el tamaño del lienzo, solucionado usando Math.min(scaleX, scaleY).
- Flechas desproporcionadas en pantallas pequeñas: se ajustó el tamaño relativo en drawArrowHead(...).

- Clics rápidos que generaban líneas incompletas: se filtraron correctamente los eventos de ratón en onMouseReleased.

Evaluación

Esta funcionalidad cumple plenamente su propósito: permite al usuario representar elementos tácticos con claridad visual e inmediata. Su diseño modular basado en objetos facilita su mantenimiento y ampliación, pudiendo incorporar en el futuro nuevas variantes como líneas discontinuas, curvas o zonas sombreadas.

6.2. Reescalado automático de elementos en la pizarra

Una de las mejoras técnicas claves implementadas en *MiPizarra* es el reescalado dinámico de todos los elementos gráficos cuando se modifica el tamaño de la ventana o del lienzo principal. Esta funcionalidad garantiza una correcta visualización de jugadores, trayectorias y objetos sobre la pista, independientemente de la resolución de pantalla o del tamaño de la aplicación.

Funcionamiento general

Al modificar el tamaño de la ventana, se recalculan automáticamente las proporciones del lienzo. Posteriormente, cada elemento almacenado en la estructura de datos interna (jugadores, líneas, movimientos, etc.) se redibuja con los nuevos factores de escala. Esto permite que todos los objetos mantengan su posición relativa sobre la cancha, respetando proporciones, alineaciones y relaciones espaciales entre sí.

Lógica interna

El sistema calcula los factores de escalado horizontal y vertical (scaleX, scaleY) comparando el tamaño actual del lienzo con las dimensiones base sobre las que se diseñó la jugada:

```
// Obtener nuevas dimensiones
double newWidth = drawingCanvas.getWidth();
double newHeight = drawingCanvas.getHeight();

// Calcular escala relativa basada en el tamaño inicial `tamX` y `tamY`
double scaleX = newWidth / tamX;
double scaleY = newHeight / tamY;
```

Cada objeto gráfico se dibuja multiplicando sus coordenadas por estos factores. Por ejemplo, en LineShape.java:

```
double scale = Math.min(scaleX, scaleY);
gc.setStroke(color);
gc.setLineWidth(strokeWidth * scale);

// Dibujar la línea
gc.strokeLine(originalX1 * scaleX + offsetX, originalY1 * scaleY + offsetY,
              originalX2 * scaleX + offsetX, originalY2 * scaleY + offsetY);
```

Este enfoque se aplica también a jugadores, trayectorias libres y círculos, a través de la jerarquía de DrawableShape.

Resultado visual

A continuación, se muestran dos capturas del mismo diseño táctico con diferentes tamaños de ventana. Puede observarse que **todos los elementos se reescalan proporcionalmente**, manteniendo la claridad y alineación en ambas resoluciones:

Figura 14: Resolución original (ventana más estrecha).

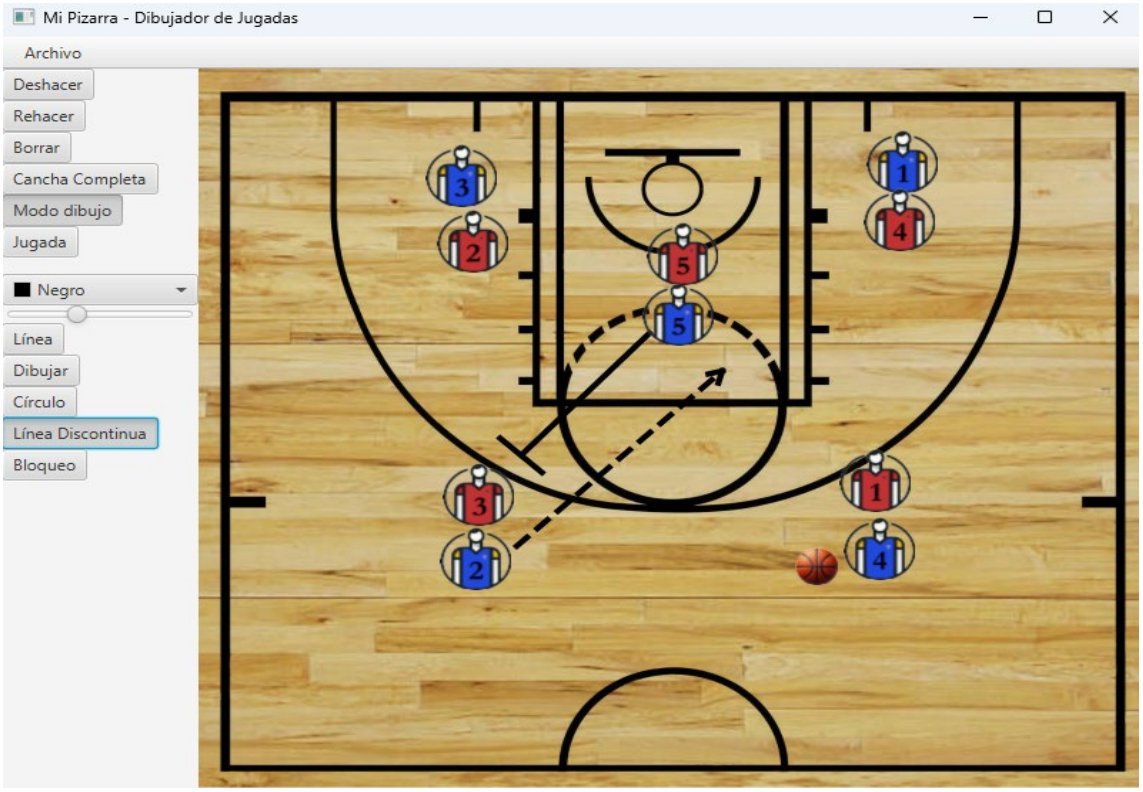
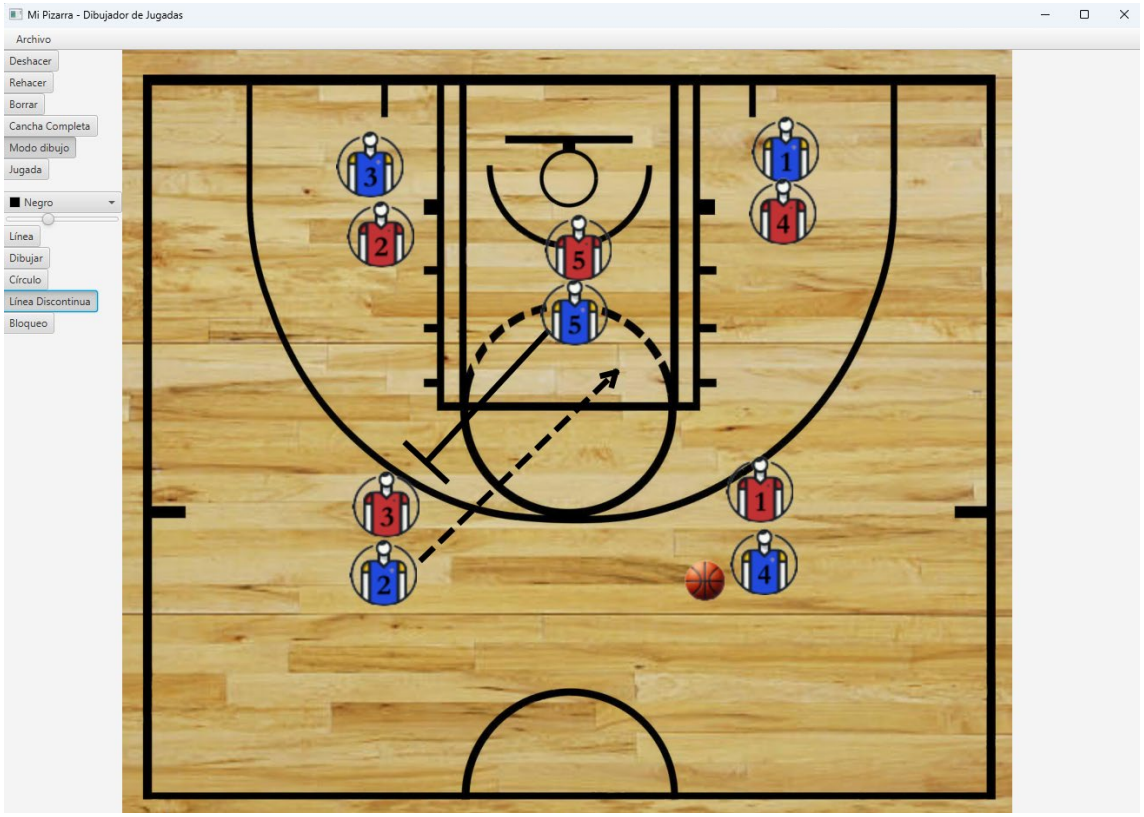


Figura 15: Resolución expandida (ventana más ancha):.



Problemas encontrados

Durante el desarrollo inicial se detectaron los siguientes inconvenientes:

- **Elementos con coordenadas absolutas** no respondían al cambio de tamaño, quedando desalineados.
- Los objetos perdían su relación de proporción respecto a la pista.
- Las flechas y grosores de línea eran demasiado grandes o pequeños en algunas resoluciones.

Todos estos aspectos fueron solucionados:

- Generalizando el uso de scaleX y scaleY.
- Corrigiendo strokeWidth para que también se escale.
- Reestructurando el método draw() de cada clase heredada de DrawableShape.

Evaluación

El sistema de reescalado automático aporta robustez y adaptabilidad a la aplicación. Resulta especialmente útil si se utiliza en entornos variados (pantallas

pequeñas, proyectores, pizarras digitales). Además, permite mantener una experiencia visual uniforme y profesional sin que el usuario deba ajustar manualmente ningún elemento.

6.3. Gestión de jugadas y reproducción secuencial de movimientos

Una de las funcionalidades más destacadas de *MiPizarra* es el sistema de gestión de jugadas dinámicas, que permite registrar y reproducir secuencias de movimientos de los elementos sobre la pista. Esta característica convierte a la aplicación en una herramienta no solo útil para el diseño de disposiciones tácticas estáticas, sino también para la **simulación progresiva y didáctica** de jugadas deportivas paso a paso.

Gracias a este sistema, los usuarios pueden almacenar los desplazamientos realizados por jugadores u otros elementos, recorrer manualmente cada paso en ambas direcciones, reiniciar la secuencia desde el principio y, adicionalmente, **activar una reproducción automática a través de un sistema basado en Timeline de JavaFX**, con control de velocidad ajustable mediante un slider.

Descripción general

Cada jugada se construye como una **secuencia ordenada de "fotogramas"** o estados intermedios. En cada fotograma, la posición de los elementos (jugadores, balón, etc.) sobre la pista queda registrada. Al mover un objeto durante el modo "jugada", se captura ese estado y se añade cronológicamente a la estructura de la jugada.

Esta lógica se basa en una lista de movimientos encapsulada en la clase Jugada.java, que permite navegar la secuencia mediante botones de avance y retroceso. Esto facilita el análisis visual de la evolución táctica en entornos educativos, deportivos o expositivos.

Implementación técnica

La clase central de esta funcionalidad es Jugada.java, que encapsula toda la lógica de almacenamiento, navegación y control de secuencia. Internamente, mantiene una lista de objetos de tipo Movible, que representan elementos capaces de cambiar de posición sobre la pista.

```
private List<Movible> elementos;
private int posicionActual;
```

Cuando un usuario mueve un elemento, dicho movimiento se registra mediante:

```
public void agregarElemento(Movible obj) {
    elementos.add(obj);
    ++posicionActual;
    mostrarMovimientosJugada();
}
```

Esto añade el nuevo estado a la lista de la jugada y actualiza el índice de posición actual (posicionActual), que indica el punto en la secuencia desde donde se está reproduciendo.

La navegación entre pasos se gestiona con los siguientes métodos:

```
public Movible obtenerSiguiente() {
    if (posicionActual < elementos.size()) {
        return elementos.get(posicionActual++);
    }
    return null;
}

public Movible obtenerAnterior() {
    if (posicionActual > 0) {
        return elementos.get(posicionActual--);
    }
    return null;
}
```

El primero devuelve el siguiente estado si existe, y el segundo permite retroceder por la secuencia. Ambos actualizan automáticamente la posición interna.

La jugada puede reiniciarse desde el principio mediante:

```
public void reset() {
    //
    posicionActual = 0;
}
```

Además, el método mostrarMovimientosJugada() permite imprimir por consola el contenido completo de la secuencia, con información detallada de las posiciones registradas por cada objeto.

Control de reproducción y escalabilidad

Este sistema está preparado para integrarse con temporizadores como Timeline (de JavaFX), lo que permitiría añadir una reproducción automática de la jugada a velocidad configurable. El método getElementosDesde(int posInicial) permite acceder a una sublista de movimientos desde un punto intermedio, lo que abre la puerta a:

- Reproducción parcial de una jugada.
- Avance automático desde cualquier punto.
- Posibilidad futura de pausar, repetir o acelerar jugadas.

La arquitectura diseñada también permite asociar metadatos (por ejemplo, nombre de la jugada, etiquetas, autor) y extenderse con funcionalidades como exportación o edición.

Estado anterior vs. funcionalidad actual

En versiones anteriores del proyecto, los movimientos de los elementos se realizaban de forma manual, pero no se conservaban ni existía ninguna estructura para representarlos como parte de una jugada. El usuario no podía volver atrás ni avanzar secuencialmente por los desplazamientos realizados,además que se printeara cada movimiento hasta el punto final

La situación actual introduce un modelo de almacenamiento ordenado, navegación completa y una base sólida para reproducir y analizar jugadas paso a paso.

Figura 16. Estado anterior.

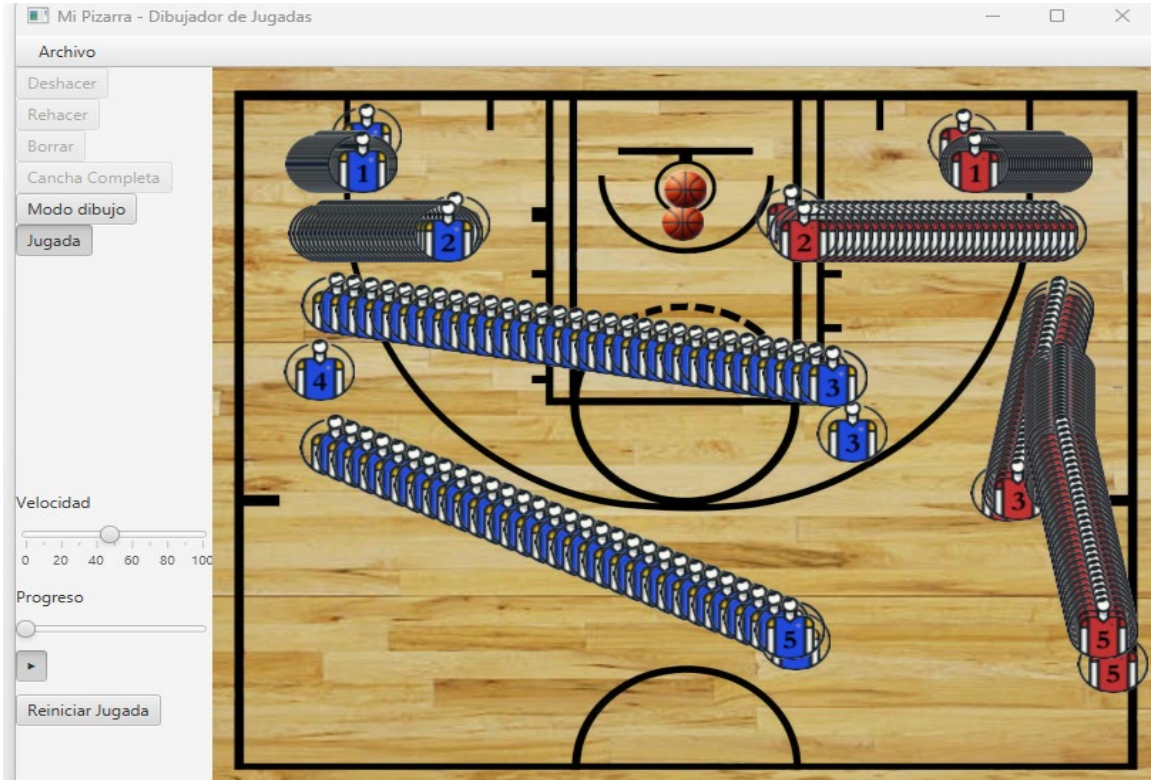
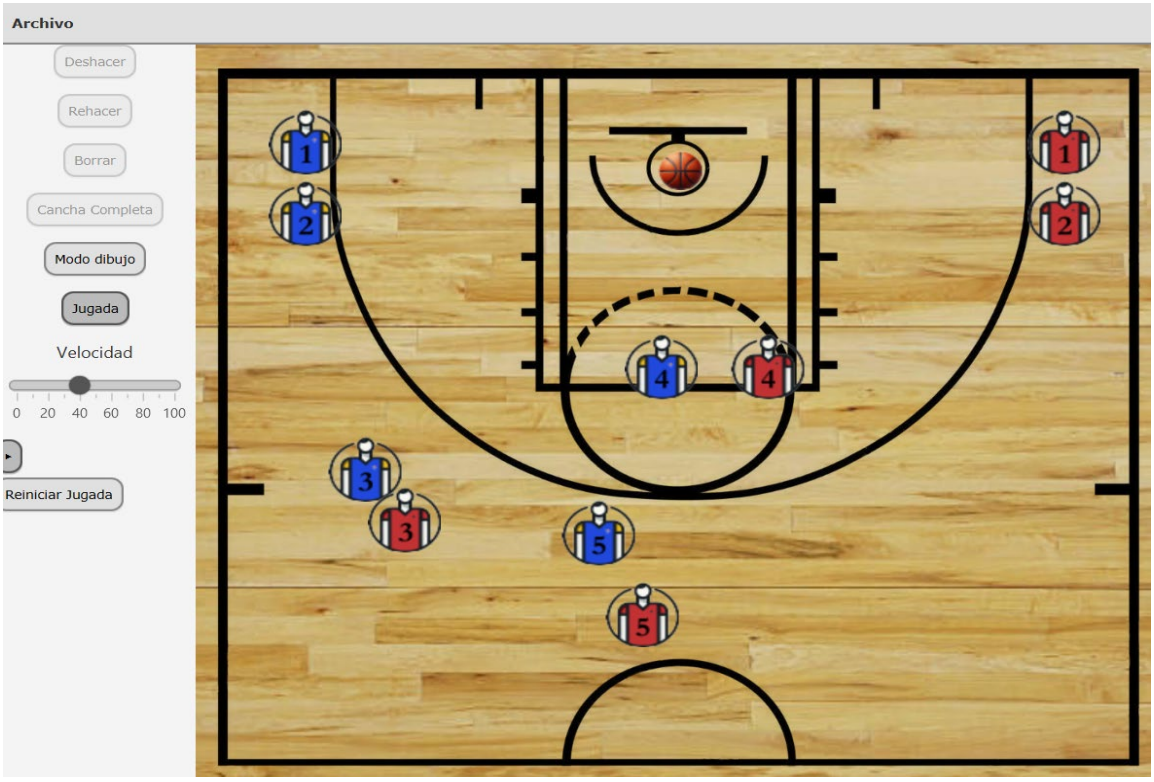


Figura 17. Estado actual.



Evaluación y posibilidades futuras

La gestión de jugadas representa una de las funcionalidades más destacadas del proyecto. Aporta valor pedagógico, realismo táctico y versatilidad de uso en contextos formativos, deportivos o recreativos. Su diseño modular y su clara separación entre lógica de control y visualización permiten futuras ampliaciones como:

- Asociación de comentarios por paso.
- Edición, inserción o eliminación de estados intermedios.

Esta funcionalidad convierte a *MiPizarra* en una herramienta más completa, flexible y alineada con las necesidades reales de los entrenadores, docentes y jugadores.

6.4. Guardar, cargar y borrar jugadas

Una vez desarrollada la funcionalidad de gestión de jugadas como secuencia de movimientos dinámicos, el siguiente paso lógico en la evolución de la aplicación *MiPizarra* ha sido implementar un sistema que permita **guardar dichas jugadas**, **cargarlas en sesiones posteriores** y **eliminarlas cuando ya no son necesarias**.

Este módulo de persistencia proporciona al usuario una herramienta esencial para trabajar con jugadas predefinidas, reutilizar estrategias en distintos entrenamientos, compartir contenido y mantener un archivo organizado de sesiones tácticas.

Objetivo y planteamiento funcional

El sistema permite:

- **Guardar** una jugada construida por el usuario, asignándole un nombre y generando un archivo en disco.
- **Cargar** una jugada previamente guardada desde un listado de ficheros disponibles.
- **Borrar** una jugada seleccionada, eliminando su archivo físico del almacenamiento.

Estas operaciones están diseñadas para ser intuitivas y rápidas, y trabajan directamente con los archivos locales del sistema, utilizando el formato **JSON** como estándar de serialización.

Implementación técnica

La lógica principal está encapsulada en la clase FicheroController.java, que actúa como intermediario entre el modelo de jugadas (Jugada.java) y el sistema de archivos.

La serialización y deserialización de objetos Jugada se realiza utilizando la librería **Jackson**, ampliamente utilizada para trabajar con JSON en Java. Gracias a la anotación @JsonIgnoreProperties(ignoreUnknown = true) en la clase Jugada, se asegura compatibilidad con versiones anteriores y se ignoran atributos irrelevantes en la carga.

Guardar jugada

La funcionalidad para guardar una jugada recoge la instancia actual de la clase Jugada, la convierte a JSON y la guarda como un archivo .json con el nombre elegido por el usuario:

```
private static final ObjectMapper mapper = new ObjectMapper();
```

Esta operación genera una representación textual completa de la jugada, incluyendo los elementos, sus posiciones, rutas de imagen y otras propiedades relevantes.

Cargar jugada

La carga se realiza mediante la selección de un archivo .json desde un menú o ventana de exploración. El archivo se interpreta y reconstruye automáticamente el objeto Jugada en memoria:

```
public static PizarraSerializable cargarPizarra(File archivo) throws IOException {
    return mapper.readValue(archivo, PizarraSerializable.class);
}
```

Una vez cargada, los elementos se dibujan nuevamente sobre la pizarra y la secuencia de movimientos queda activa para su navegación paso a paso.

Borrar jugada (reiniciar jugada activa)

A diferencia de un borrado físico de archivo, la acción de "borrar jugada" en *MiPizarra* no elimina ficheros del sistema, sino que reinicia la jugada actual cargada en memoria. Esta operación vacía la lista de movimientos y posiciones previamente registrados, devolviendo el estado interno del sistema a un punto limpio desde el cual se puede comenzar a diseñar una nueva jugada.

La operación se implementa en la clase Jugada mediante el método:

```
public void reiniciarJugada(){
    reset();
    elementos.removeAll(elementos);
}
```

Esto garantiza que tanto la lista de elementos registrados como el índice de reproducción (posicionActual) se reinicien por completo, asegurando un entorno vacío y funcionalmente idéntico al estado inicial de la aplicación tras su apertura.

Esta opción es útil cuando el usuario desea descartar una jugada que está construyendo y comenzar otra desde cero, sin necesidad de cerrar ni recargar el programa.

Organización de archivos

Las jugadas guardadas se almacenan en una carpeta específica del proyecto, accesible al usuario. Cada archivo se nombra de acuerdo con el título introducido en el guardado y contiene toda la información de forma autosuficiente.

Ejemplo de ruta:

Ventajas del enfoque JSON

- **Ligero y legible:** puede abrirse y editarse con cualquier editor de texto.
- **Independiente del sistema:** compatible con otros programas si se desea exportar.
- **Flexible:** permite futuras ampliaciones del modelo sin romper la compatibilidad.

Comparativa antes/después

Antes: no era posible conservar jugadas entre sesiones. Cada ejecución partía de cero.
Ahora: el usuario puede almacenar su trabajo, cargar jugadas tácticas previamente guardadas y gestionar su biblioteca de estrategias desde la propia interfaz.

Figura 18: Parte de la interfaz para manejar las jugadas.

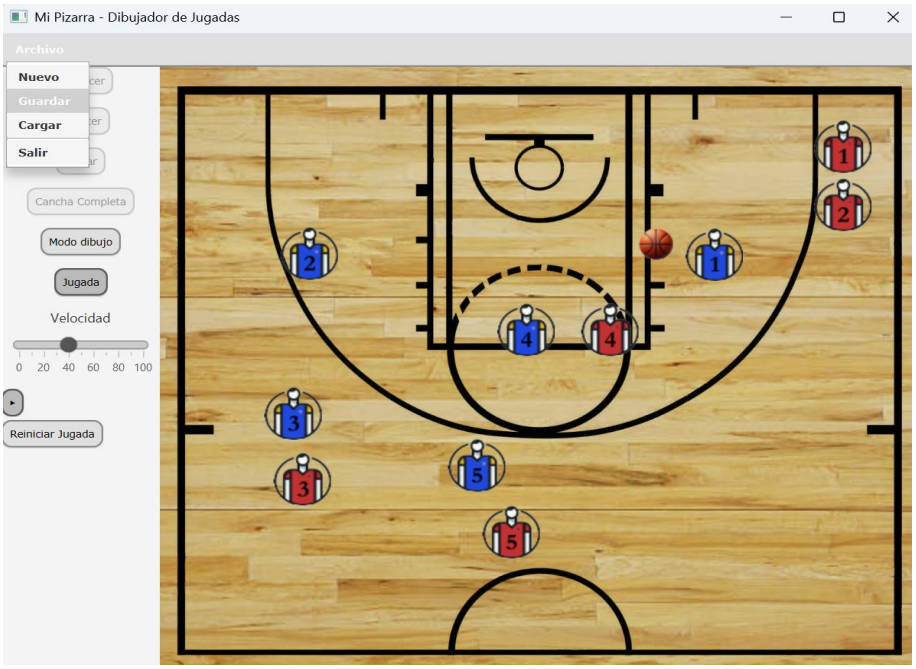
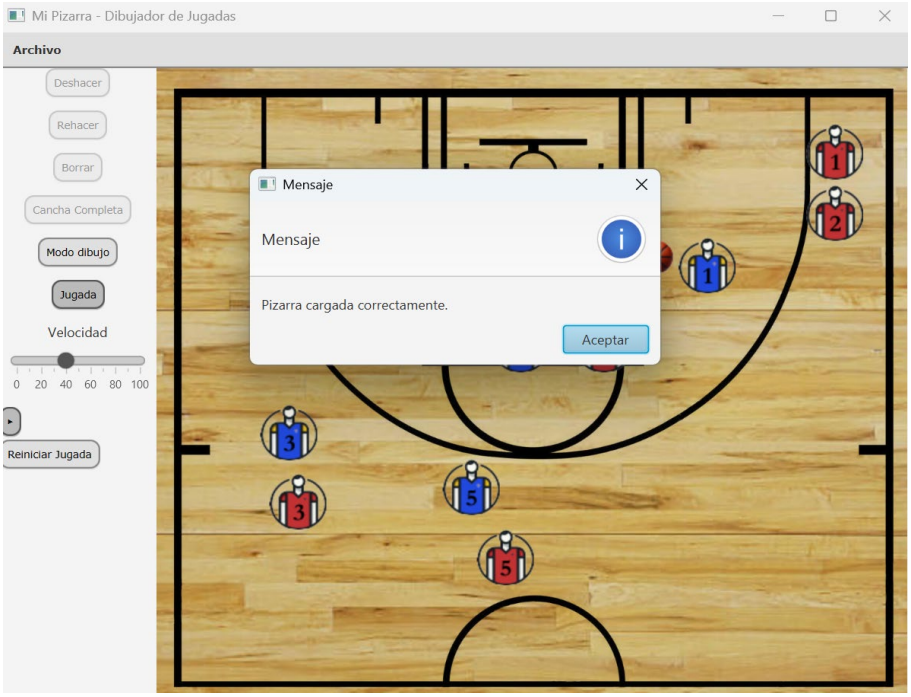


Figura 19: Mensaje de carga de jugada realizada.



Evaluación y posibles mejoras

La incorporación de esta funcionalidad representa un paso fundamental en la madurez del proyecto. Ofrece continuidad entre sesiones, fomenta la reutilización del trabajo realizado y permite compartir jugadas con terceros.

Entre las mejoras previstas se incluyen:

- Añadir etiquetas o metadatos a cada jugada (nivel, tipo de defensa...).
- Permitir previsualizar una jugada antes de cargarla.
- Sincronizar jugadas con almacenamiento en la nube o bases de datos externas.

Esta funcionalidad no solo amplía las capacidades del sistema, sino que lo aproxima al estándar de herramientas profesionales utilizadas por cuerpos técnicos en distintos deportes colectivos.

---

7. Pruebas y validación

El presente capítulo recoge el conjunto de pruebas realizadas para garantizar la correcta funcionalidad, estabilidad y robustez de la aplicación *MiPizarra*. Estas pruebas han sido diseñadas para verificar que cada componente del sistema responde conforme a los requisitos definidos, y que el comportamiento general de la aplicación es predecible, consistente y libre de errores críticos.

El proceso de validación se ha estructurado en distintos bloques, atendiendo a las principales funcionalidades implementadas: edición gráfica, gestión de jugadas, control de historial y persistencia de datos. En cada bloque se han definido casos de prueba representativos, que han sido ejecutados manualmente y en diferentes entornos.

Además, se han evaluado aspectos como el rendimiento ante operaciones repetidas, la consistencia del estado del sistema tras acciones encadenadas, y la fiabilidad de la estructura de datos exportada en los archivos generados. Las pruebas se han complementado con inspección visual, análisis de código y verificación directa de archivos y trazas internas.

7.1. Metodología de pruebas

Las pruebas se han realizado de forma manual durante todo el proceso de desarrollo, ejecutando la aplicación desde el entorno NetBeans y comprobando su comportamiento frente a distintos escenarios de uso. Se han probado casos de uso completos, incluyendo secuencias de dibujo, reescalado dinámico, manipulación de elementos y gestión de jugadas. Para cada funcionalidad, se han contemplado tanto casos de éxito como de error, con el objetivo de verificar su robustez y comportamiento esperado.

7.1.1. Pruebas relacionadas con el dibujo táctico

Funcionalidad Dibujar línea táctica (flecha) – Prueba de Éxito

Precondiciones El usuario ha seleccionado el modo "Línea" en la barra de herramientas.
Ejecución El usuario hace clic y arrastra desde un punto A hasta un punto B en el lienzo.
Resultado Se dibuja una línea con una flecha al final que indica la dirección del desplazamiento.
Evaluación Completada exitosamente.

Funcionalidad Dibujar línea de bloqueo (perpendicular) – Prueba de Éxito

Precondiciones El usuario ha seleccionado el modo "Bloqueo".
Ejecución El usuario hace clic y arrastra sobre la pista.
Resultado Se representa una línea con remate perpendicular al final.
Evaluación Completada exitosamente.

Funcionalidad Dibujar línea discontinua – Prueba de Éxito

Precondiciones El usuario ha activado el modo "Línea discontinua".
Ejecución El usuario traza una línea discontinua entre dos puntos.
Resultado La línea aparece correctamente fragmentada.
Evaluación Completada exitosamente.

Funcionalidad Deshacer línea dibujada – Prueba de Éxito

Precondiciones El usuario ha trazado una o varias líneas sobre la pista.
Ejecución El usuario pulsa el botón "Deshacer".
Resultado Se elimina el último trazo realizado del historial gráfico.
Evaluación Completada exitosamente.

Funcionalidad Rehacer línea deshecha – Prueba de Éxito

Precondiciones El usuario ha deshecho una línea previamente dibujada.
Ejecución El usuario pulsa el botón "Rehacer".
Resultado La línea eliminada vuelve a mostrarse en la pizarra.
Evaluación Completada exitosamente.

7.1.2. Pruebas sobre la gestión de jugadas (grabación y navegación)

Funcionalidad Registro de movimiento de jugador – Prueba de Éxito

Precondiciones El botón "Jugada" está activado.
Ejecución El usuario mueve un jugador desde una posición inicial a una nueva posición.
Resultado El movimiento queda almacenado como parte de una secuencia.
Evaluación Completada exitosamente.

Funcionalidad Reproducción de jugada paso a paso – Prueba de Éxito

Precondiciones Hay una jugada con varios movimientos registrados.
Ejecución El usuario pulsa "Siguiente paso" varias veces.
Resultado Se avanza correctamente por la secuencia de posiciones.
Evaluación Completada exitosamente.

Funcionalidad Reiniciar jugada – Prueba de Éxito

Precondiciones Una jugada ha sido cargada o registrada parcialmente.
Ejecución El usuario pulsa el botón "Reiniciar jugada".
Resultado Todos los movimientos son eliminados de la jugada activa.
Evaluación Completada exitosamente.

Funcionalidad No se graban movimientos si el modo jugada no está activado – Prueba de Éxito

Precondiciones El botón "Jugada" está desactivado.
Ejecución El usuario mueve varios jugadores.
Resultado Ninguno de los movimientos queda registrado como parte de una jugada.
Evaluación Comportamiento correcto validado.

7.1.3. Pruebas sobre guardado, carga y borrado de jugadas

Funcionalidad Guardar jugada – Prueba de Éxito

Precondiciones Hay una jugada registrada y activa.
Ejecución El usuario pulsa “Guardar”, introduce un nombre y confirma.
Resultado Se crea un archivo JSON con la información de la jugada.
Evaluación Completada exitosamente.

Funcionalidad Cargar jugada válida – Prueba de Éxito

Precondiciones Existe un archivo .json con una jugada previamente guardada.
Ejecución El usuario selecciona “Cargar jugada” y escoge el archivo.
Resultado La jugada se carga correctamente y se visualiza en el lienzo.
Evaluación Completada exitosamente.

Funcionalidad Cargar jugada inválida (formato erróneo) – Prueba de Error

Precondiciones El usuario selecciona un archivo JSON corrupto o mal formado.
Ejecución Se intenta cargar la jugada desde la interfaz.
Resultado Se muestra un mensaje de error, sin cerrar ni afectar la aplicación.
Evaluación Completada exitosamente.

7.1.4. Pruebas de movimiento y reescalado de elementos

Funcionalidad Reescalado automático al cambiar tamaño de ventana – Prueba de Éxito

Precondiciones Hay varios elementos en la pista.
Ejecución El usuario redimensiona la ventana principal.
Resultado Todos los elementos (jugadores, líneas, zonas) se reescalan proporcionalmente.
Evaluación Completada exitosamente.

Funcionalidad Pérdida de proporciones en versión inicial – Prueba de Error corregido

Precondiciones El proyecto no aplicaba correctamente scaleX y scaleY.
Ejecución Al reescalar la ventana, las líneas se deforman o se desplazan mal.
Resultado Error identificado y corregido ajustando todos los elementos a las proporciones relativas.
Evaluación Completada exitosamente.

7.1.5. Pruebas sobre persistencia (JSON)

Funcionalidad
Guardar una jugada con elementos y movimientos – Prueba de Éxito
Precondiciones
Existen jugadores colocados en la pista y se ha realizado al menos un movimiento.
Ejecución
El usuario pulsa el botón "Guardar jugada" y selecciona una ubicación.
Resultado
Se genera un archivo .json bien formado que contiene todos los elementos y la secuencia de movimientos.
Evaluación
Completada exitosamente.

Funcionalidad
Cargar una jugada previamente guardada – Prueba de Éxito
Precondiciones
Existe un archivo.json válido guardado por la aplicación.
Ejecución
El usuario pulsa "Cargar jugada" y selecciona el archivo correspondiente.
Resultado
Los elementos y movimientos se restauran fielmente en el mismo estado en el que se guardaron.
Evaluación
Completada exitosamente.

Funcionalidad
Sobrescritura de una jugada – Prueba de Éxito
Precondiciones
Se ha cargado una jugada existente y se han hecho modificaciones.
Ejecución
El usuario vuelve a guardar la jugada en el mismo archivo.
Resultado
El archivo se actualiza correctamente sin errores, y conserva las modificaciones.
Evaluación
Completada exitosamente.

Funcionalidad
Cargar archivo JSON malformado – Prueba de Error controlado
Precondiciones
El archivo seleccionado no contiene una estructura JSON válida.
Ejecución
El usuario intenta cargar el archivo mediante el botón correspondiente.
Resultado
El sistema detecta el error y lanza un aviso, sin bloquear la interfaz ni provocar cierres inesperados.
Evaluación
Completada exitosamente.

Funcionalidad
Compatibilidad con múltiples tipos de elementos – Prueba de Éxito
Precondiciones
La jugada incluye jugadores, balón, zonas y movimientos.
Ejecución
Se guarda y carga el archivo .json generado.
Resultado
Todos los tipos de elementos y sus propiedades son correctamente serializados y deserializados.
Evaluación
Completada exitosamente.

7.2. Casos prácticos

Para ilustrar el funcionamiento y la aplicabilidad de *MiPizarra*, se presentan a continuación tres casos prácticos que reflejan situaciones reales de uso. Estos ejemplos muestran cómo entrenadores, docentes o estudiantes pueden

beneficiarse de las funcionalidades del sistema para diseñar, reproducir y analizar jugadas tácticas.

7.2.1. Diseño de una jugada estática con múltiples elementos

Contexto:
Un entrenador de baloncesto desea representar la disposición inicial de su equipo para una jugada de salida de presión. Para ello, utiliza la aplicación para dibujar la cancha y posicionar cinco jugadores.

Acciones realizadas:

- Coloca los jugadores en posiciones específicas sobre la pista.
- Usa líneas con flecha para indicar trayectorias de desmarque.
- Añade líneas de bloqueo y zonas circulares para señalar espacios clave.
- Ajusta los colores y grosores para diferenciar funciones (bloqueo, pase, desplazamiento).

Resultado esperado: La jugada queda representada de forma clara y visual, lista para proyectarse en una clase o sesión de entrenamiento. El archivo puede guardarse como .json para futuras modificaciones.

7.2.2. Simulación dinámica de una jugada ofensiva paso a paso

Contexto:
Durante una sesión teórica, el docente quiere mostrar cómo evoluciona una jugada ofensiva de balonmano en tiempo real, paso a paso.

Acciones realizadas:

- Activa el modo “jugada”.
- Coloca a los jugadores y simula los movimientos a lo largo de varios pasos, desplazándolos manualmente.
- Utiliza el sistema de navegación para avanzar y retroceder por la secuencia.
- Controla la velocidad de reproducción con el slider correspondiente.

Resultado esperado:
Se obtiene una simulación dinámica que permite visualizar los cambios de

posición en cada fase. Esto facilita la comprensión táctica por parte del alumnado y permite detenerse en momentos clave del desarrollo.

7.2.3. Carga de una jugada guardada y análisis retrospectivo

Contexto:
Un usuario ha diseñado previamente una jugada defensiva durante una práctica anterior. Ahora desea revisarla para introducir mejoras.

Acciones realizadas:

- Abre la aplicación e importa una jugada almacenada en formato .json.
- Recorre cada paso con los botones de avance para analizar la secuencia.
- Utiliza las herramientas de dibujo para remarcar zonas débiles.
- Limpia el lienzo, modifica posiciones y guarda una nueva versión con otro nombre.

Resultad esperado:

El usuario puede trabajar sobre una jugada ya existente, evaluarla y modificarla sin necesidad de repetir todo el proceso desde cero. Esto favorece la iteración y la mejora progresiva de las estrategias diseñadas.

---

8. Manual de usuario

Este apartado proporciona una guía detallada para el uso correcto de la aplicación *MiPizarra*. Se describe el comportamiento del sistema desde el punto de vista del usuario final, abordando el flujo general de trabajo, las funcionalidades principales, los controles disponibles en la interfaz y los procedimientos habituales para la creación, edición y gestión de jugadas tácticas. La aplicación está diseñada para ser utilizada por perfiles no técnicos, como entrenadores, docentes o estudiantes, sin necesidad de conocimientos previos en programación.

8.1. Requisitos mínimos del sistema

Para asegurar el funcionamiento correcto de la aplicación *MiPizarra*, se recomienda cumplir con los siguientes requisitos mínimos tanto a nivel de hardware como de software:

Requisitos de hardware:

- Procesador: Intel i3 (o equivalente) a 2 GHz o superior.
- Memoria RAM: 4 GB mínimo (8 GB recomendados).
- Espacio en disco: 200 MB libres.
- Resolución de pantalla: 1280 x 720 píxeles (mínimo).

Requisitos de software:

- Sistema operativo: Windows 10 / 11, macOS 10.14+ o una distribución Linux actualizada.
- Java Development Kit (JDK): versión 20 o superior.
- JavaFX SDK: versión compatible con JDK 20 (si no está embebido en el JDK).
- NetBeans IDE: versión 18 o superior (u otro IDE compatible con Maven y JavaFX).
- Navegador de archivos (para seleccionar archivos .json al guardar/cargar jugadas).

8.2. Manual de instalación

Para ejecutar y modificar el proyecto *MiPizarra* en un entorno local, se deben seguir los siguientes pasos:

1. Instalar Java y JavaFX

- Descargar e instalar el JDK 20 desde https://jdk.java.net.
- Asegurarse de que el PATH del sistema incluya java y javac.
- Si la distribución del JDK no incluye JavaFX, descargar el SDK desde https://gluonhq.com/products/javafx/.

2. Instalar NetBeans (recomendado)

- Descargar NetBeans desde https://netbeans.apache.org/download/index.html.
- Durante la instalación, asegurarse de seleccionar soporte para Java y Maven.
- Configurar JavaFX como biblioteca global si es necesario (en *Tools* > *Libraries*).

3. Clonar o importar el proyecto

- Clonar el repositorio desde GitHub o descomprimir el ZIP con el proyecto.
- Abrir NetBeans y seleccionar **File > Open Project**, luego localizar la carpeta raíz del proyecto.

4. Compilar y ejecutar

- NetBeans detectará automáticamente el archivo pom.xml de Maven.
- Ejecutar el proyecto con **Run > Run Project** o presionando F6.
- Asegurarse de que las rutas de los recursos (/images/...) sean accesibles desde el sistema de archivos.

5. Verificación final

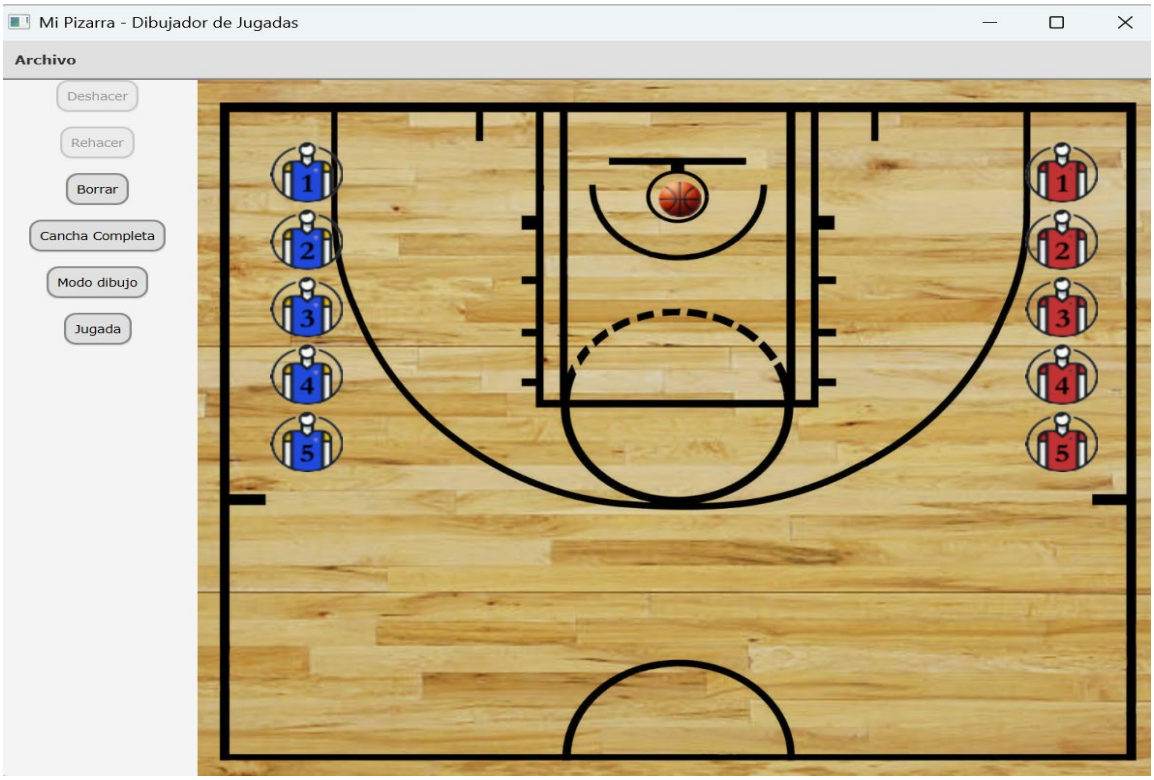
- Comprobar que se abra la ventana principal con la pista de juego.
- Probar añadir jugadores y dibujar líneas

8.3. Interfaz inicial

Al iniciar la aplicación, el usuario accede a una ventana principal donde se muestra el **campo de juego** (cancha) en el centro de la interfaz. En la parte superior se encuentra una **barra de herramientas** con múltiples iconos que permiten cambiar entre modos de edición, modificar el comportamiento del entorno y gestionar jugadas.

A la derecha (o en la parte inferior, según la resolución), se disponen controles complementarios que permiten avanzar o retroceder en la jugada, guardar y cargar movimientos, y reiniciar la escena.

Figura 20: Imagen al iniciar la aplicación.



8.4. Modos de dibujo

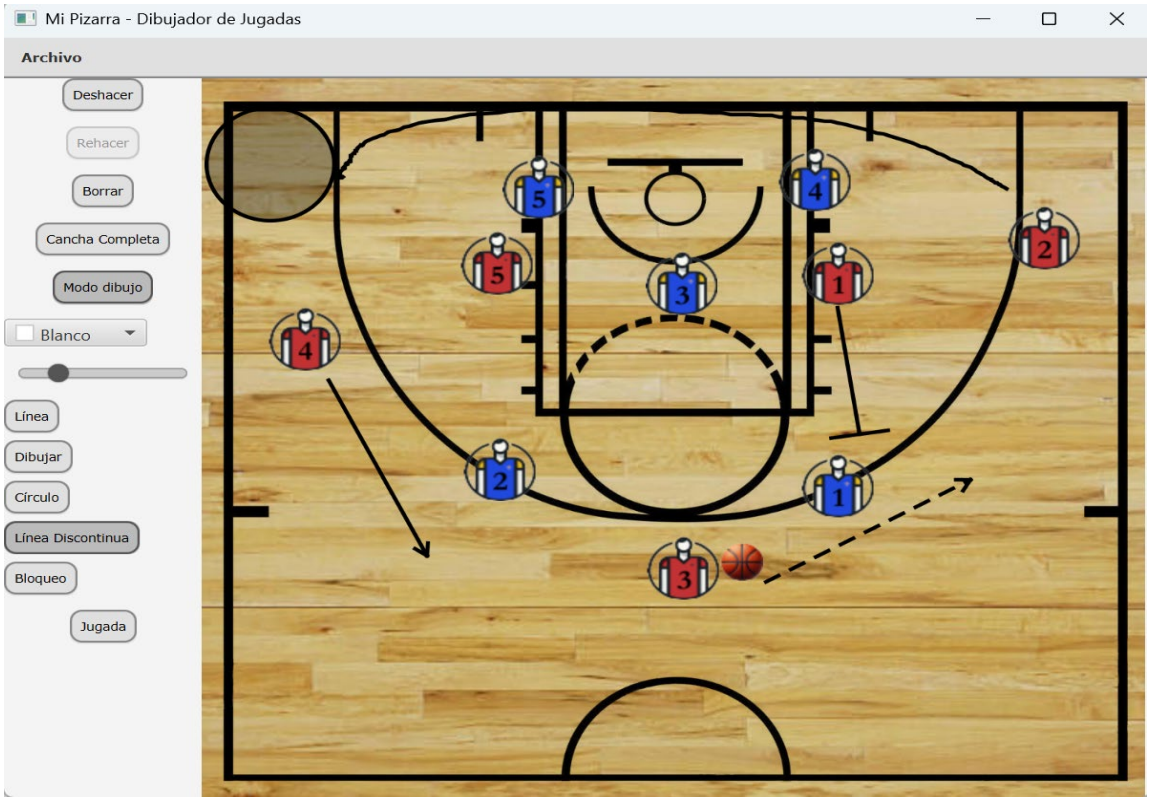
La aplicación permite **dibujar trayectorias o interacciones tácticas** utilizando varios tipos de línea:

- **Línea de movimiento (flecha)**: indica una dirección de desplazamiento.
- **Línea de bloqueo (perpendicular)**: representa una acción de pantalla o bloqueo.

- **Línea discontinua**: para mostrar movimientos secundarios o indirectos.
- **Círculos o zonas**: para marcar áreas estratégicas.
- **Dibujo libre**: para expresar otro tipo de movimientos como un aclarado.

Para dibujar, el usuario selecciona el tipo deseado, y luego hace clic y arrastra sobre el lienzo. La aplicación también permite cambiar el color de los dibujos (pudiendo elegir entre los predeterminados o personalizados) o cambiar el grosor de las líneas mediante el slider. Las líneas se ajustan automáticamente a la escala del campo y se renderizan con proporciones correctas.

Figura 21: Imagen despues de dibujar.



8.4.1. Gestión de acciones: borrar, deshacer y rehacer

La aplicación *MiPizarra* incorpora un sistema de gestión de acciones que permite **controlar, revertir o rehacer cambios sobre los elementos de dibujo libre**, es decir: líneas, flechas, zonas circulares, líneas de bloqueo y trayectorias discontinuas realizadas sobre el lienzo.

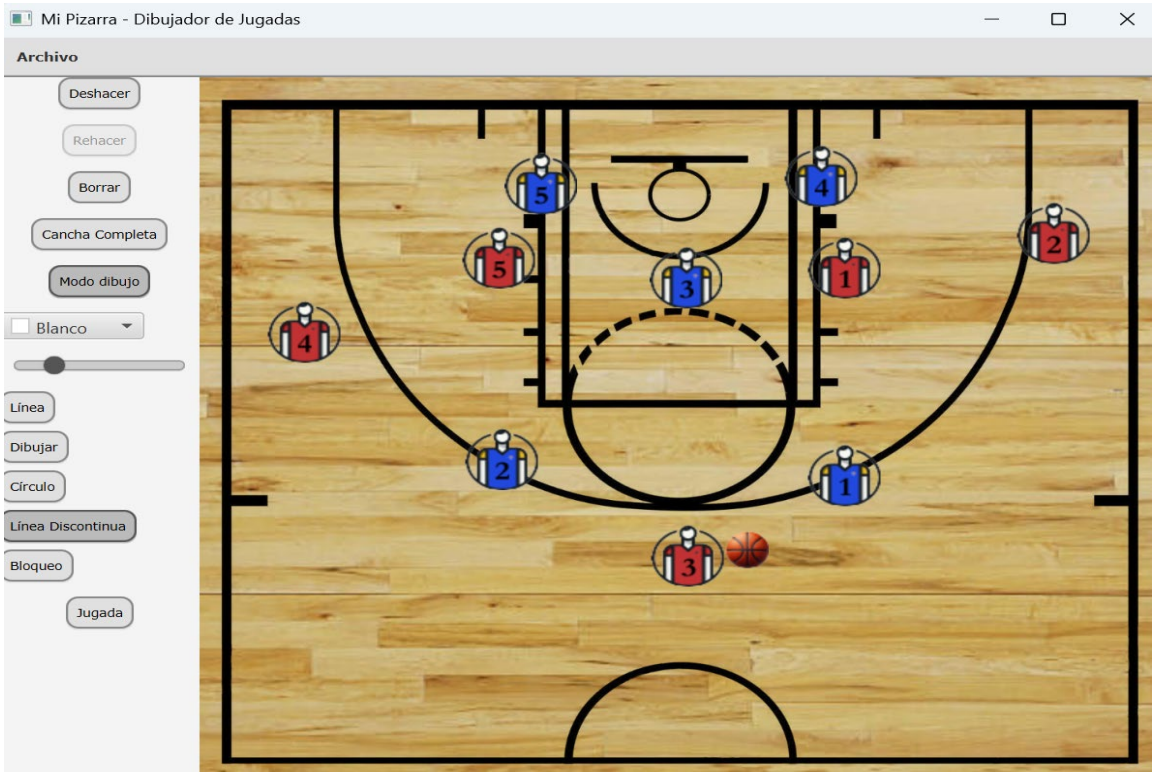
Este conjunto de herramientas mejora la experiencia de uso durante la edición, permitiendo al usuario experimentar, corregir errores o limpiar la pizarra sin afectar a los jugadores, el balón u otros elementos fijos.

Borrar todo el dibujo

El botón de “borrar” o “limpiar pizarra” ejecuta una limpieza completa del lienzo de dibujo, eliminando todas las formas generadas mediante herramientas gráficas (líneas, círculos, trayectorias, etc.). Esta acción no elimina jugadores ni objetos colocados sobre el campo, únicamente borra los **trazos manuales** hechos con las herramientas de dibujo.

Al ejecutarse, el sistema también guarda automáticamente este estado en el historial, permitiendo que el usuario pueda deshacer el borrado si fue accidental

Figura 22: Aplicación despues de borrar el dibujo.



Deshacer una acción

La función de **deshacer** permite revertir la última modificación realizada en el área de dibujo. Cada vez que el usuario traza una nueva línea, añade una zona, o limpia el lienzo, el estado visual es almacenado en una estructura de historial.

Al presionar el botón de deshacer:

- Se recupera el estado anterior del lienzo.
- Se descarta el último trazo realizado.
- Se mantiene intacta la posición de los jugadores u otros elementos.

El historial está limitado a una profundidad máxima, evitando la acumulación innecesaria de estados.

Rehacer una acción

Si el usuario ha deshecho una acción por error, puede utilizar el botón de **rehacer** para volver a aplicar el cambio anterior. Esta función recupera el estado eliminado por la acción de deshacer y lo vuelve a aplicar sobre el lienzo.

Ámbito de aplicación

Es importante tener en cuenta que **estas funciones solo afectan al sistema de dibujo libre**. Es decir:

- No modifican ni eliminan jugadores, balón o sus posiciones.
- No afectan a la jugada registrada (movimientos paso a paso).
- No interactúan con los archivos guardados o cargados.

Esta separación garantiza que el usuario pueda experimentar gráficamente sin interferir con la construcción estructurada de una jugada.

8.5. Movimiento de elementos y registro de jugada

Los elementos añadidos al campo pueden moverse libremente. Cada desplazamiento queda registrado internamente en una estructura de jugada, permitiendo la creación de **secuencias dinámicas**, pero únicamente cuando el botón **jugada está activo**, si este botón no está activado podremos hacer los movimientos que deseemos, pero estos no se guardaran para ejecutar la jugada. Estas secuencias forman una jugada que puede ser navegada paso a paso o reproducida más adelante.

La jugada se construye automáticamente a medida que el usuario interactúa con los elementos. Se registra:

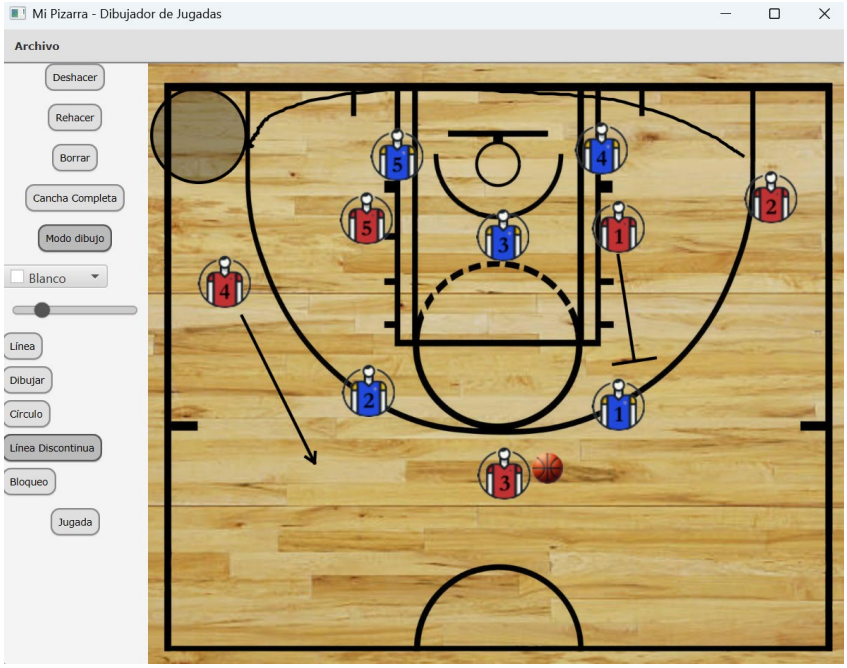
- La posición inicial y final del movimiento.
- La identidad del elemento.
- El orden cronológico del desplazamiento.

8.6. Gestión de jugadas

Una de las funcionalidades principales es la posibilidad de **gestionar jugadas completas**. Desde el panel superior o desde menús contextuales, el usuario puede:

- **Guardar jugada**: abre un cuadro de diálogo donde se solicita un nombre, y se guarda la secuencia en un archivo .json.
- **Cargar jugada**: permite seleccionar un archivo previamente guardado y restaurar el estado completo de la pista.
- **Reiniciar jugada**: borra todos los movimientos y elementos del campo, devolviendo la pizarra a su estado inicial.
- **Subir/bajar velocidad de la jugada**: permite incrementar o decrementar la velocidad de la jugada a través del slider.

Figura 23: Gestionando la jugada.



8.7. Guardado y formato JSON

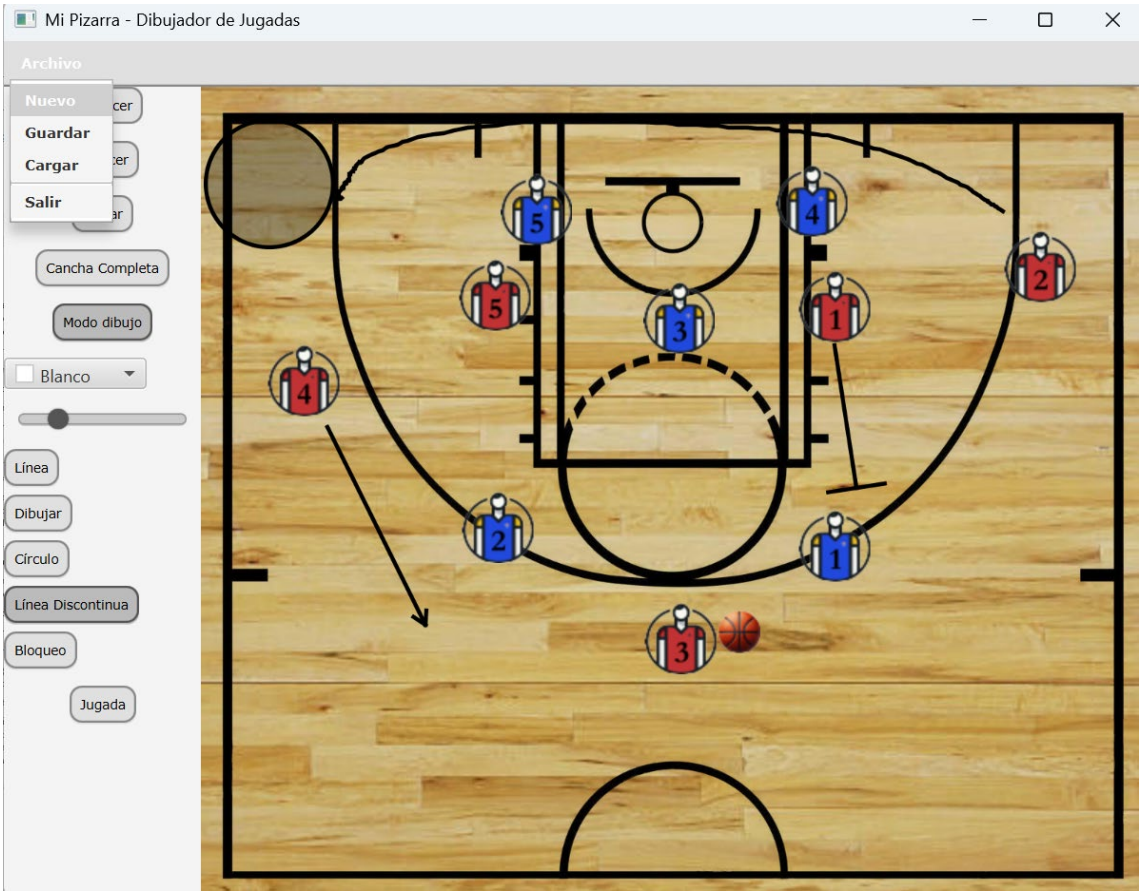
Las jugadas guardadas se almacenan en formato .json, un estándar abierto y legible. Cada archivo contiene:

- Lista de elementos movibles.
- Posiciones paso a paso.
- Identificadores y rutas de imagen.
- Posición actual de reproducción.

Este enfoque permite:

- Compartir jugadas entre usuarios.
- Editar jugadas externamente si se desea.
- Cargar secuencias en futuras versiones de la aplicación.

Figura 24: Guardar Jugada.



8.8. Flujo típico de uso

Un ejemplo de uso habitual puede seguir el siguiente flujo:

1. Abrir la aplicación y seleccionar una pista de juego.
2. Colocar jugadores y balón según el esquema deseado.
3. Dibujar y borrar líneas de movimiento y zonas clave.
4. Mover elementos para simular la jugada paso a paso.
5. Guardar la jugada con un nombre representativo.
6. Reproducirla más adelante o compartirla con otros usuarios.

---

9. Bibliografía

Adobe. (s.f.). *Photoshop - Official user guide*. Adobe Inc. Recuperado de https://helpx.adobe.com/photoshop/

Apache NetBeans. (s.f.). *NetBeans IDE*. The Apache Software Foundation. Recuperado de https://netbeans.apache.org/

Apache Software Foundation. (s.f.). *Apache Maven Project*. Recuperado de https://maven.apache.org/

FasterXML. (s.f.). *Jackson JSON Processor*. GitHub. Recuperado de https://github.com/FasterXML/jackson

Gluon. (s.f.). *JavaFX y Scene Builder*. GluonHQ. Recuperado de https://gluonhq.com/products/javafx/

JSON.org. (s.f.). *Introducing JSON*. Recuperado de https://www.json.org/

Oracle. (s.f.). *Java Platform, Standard Edition Documentation*. Recuperado de https://docs.oracle.com/javase/

PlantUML. (s.f.). *Herramienta de diagramas UML basada en texto*. Recuperado de https://plantuml.com/

PlantUML. (2020). *PlantUML QEditor v1.0: Aplicación para edición de diagramas UML desde texto*. Recuperado de https://plantuml.com/download

GanttProject. (s.f.). *Gestión de proyectos y diagramas de Gantt*. Recuperado de https://www.ganttproject.biz/

Baradwan, A. (2022). *GanttProject v3.2: Herramienta de planificación libre para proyectos educativos* [Software]. Recuperado de https://www.ganttproject.biz/

FastModel Sports. (s.f.). *FastDraw – Basketball play diagramming software*. Recuperado de https://www.fastmodelsports.com/

TacticalPad. (s.f.). *Herramienta profesional para planificación táctica*. Recuperado de https://www.tacticalpad.com/

YouCoach. (s.f.). *Recursos y software para entrenadores deportivos*. Recuperado de https://www.youcoach.com/

10. Anexo

Link del repositorio del proyecto: https://github.com/Kaesaril/Aplicaci-n-de-Pizarra-de-Jugadas-para-Baloncesto.git