Trabajo Especial:
Diseño de un Juego de Fútbol Autónomo en Pygame

Alumno/s:

Bonelli del Hoyo Santiago Nicolás
Guerrero Tomás
Librandi Gonzalo
Rautto Braian
Rodriguez Tomás Agustín
Romero Mateo

Profesor/es:

Analía Amandi

Introducción

El presente informe tiene como objetivo ofrecer una visión general del diseño del juego 2D de fútbol desarrollado en Pygame. Pygame es un conjunto de módulos en Python diseñados para facilitar el desarrollo de videojuegos en 2D. Este framework proporciona herramientas y funcionalidades esenciales para la creación de juegos, incluyendo gráficos, manejo de eventos, sonidos y colisiones. Este juego simula el enfrentamiento entre dos equipos conformados por jugadores autónomos, cuya autonomía se logra gracias a las características y funcionalidades proporcionadas por Pygame.

Cuando hablamos de autonomía nos referimos a la capacidad del juego para operar de manera independiente y realizar acciones sin intervención constante del usuario. En este caso, la autonomía se manifiesta a través de diversas características y comportamientos autónomos de los elementos del juego, como jugadores controlados por la inteligencia artificial (IA) (evitando la interacción directa a través de comandos introducidos por parte de los usuarios) y la simulación de situaciones realistas en un juego de fútbol.

Arquitectura General del Diseño del Sistema

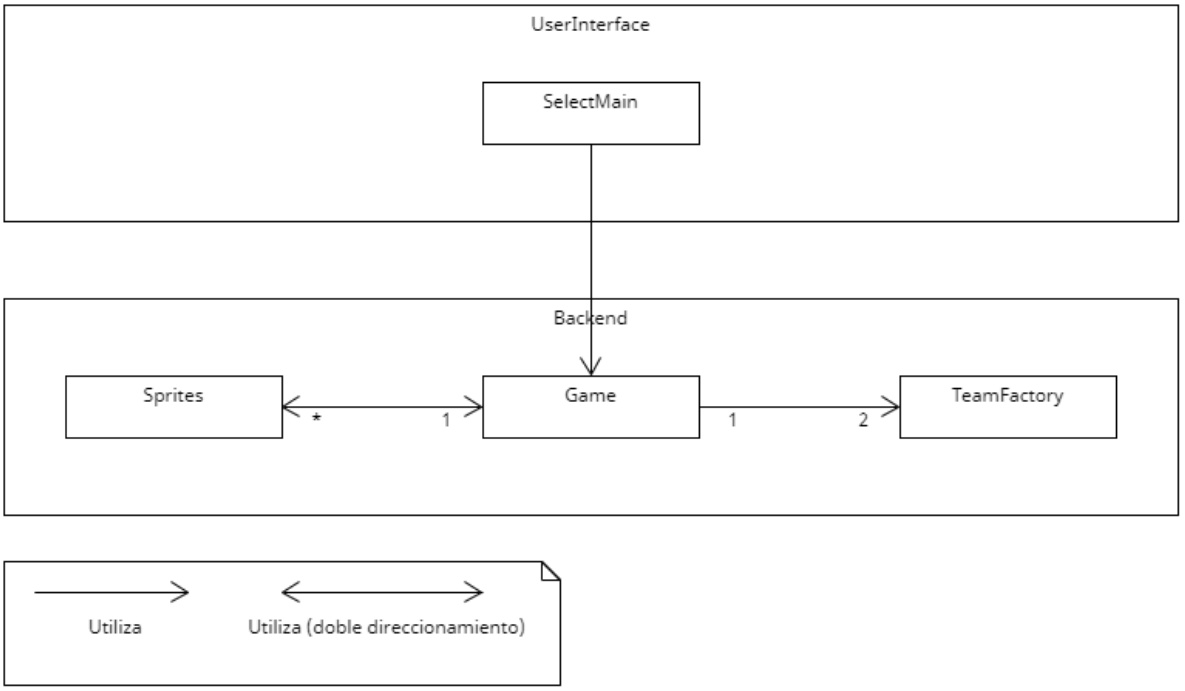

Figura 1 Arquitectura general de capas

El diseño general del sistema refleja una arquitectura de capas. Con esta arquitectura, se logra una separación clara entre la interfaz del juego y la lógica interna. La interfaz se encarga de la interacción con el usuario y la creación del juego, mientras que el backend maneja la lógica del juego y la gestión de equipos. Esta estructura facilita la extensión y mantenimiento del código, ya que los cambios en la interfaz o la lógica del juego se pueden realizar de manera independiente.
Existen dos capas:

UserInterface:

Esta capa es la encargada de gestionar la interfaz del juego, especialmente el menú principal.
SelectMain es la función principal que se ocupa de ello. Utiliza la biblioteca Pygame para crear la ventana del juego y manejar eventos como clics del mouse y teclas. Aquí, el jugador puede seleccionar dos equipos pulsando sobre el botón que los representa y comenzar un nuevo juego, creando una instancia del mismo (Game) el cual inicia a través del llamado al método play(), realizando la transición entre el menú de selección y el juego en sí. Si desea retirarse, el usuario puede pulsar ESC para salir.

Backend:

En esta capa se encuentra toda la lógica debajo de la interfaz, es decir, la implementación del juego una vez que los equipos a jugar fueron seleccionados por el usuario.

Game contiene la lógica principal del juego, como el seguimiento del estado del juego, la detección de colisiones, la gestión de eventos y la actualización de la posición de los jugadores y la pelota, incluyendo las interacciones entre estos dos últimos.
La creación de los equipos por parte de *Game* se lleva a cabo empleando el *Factory Method,* un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán. Es útil que *Game* utilice un componente llamado ***TeamFactory*****,** una clase abstracta que actúa como una interfaz para la creación de equipos en el juego, ya que nos concede dinamismo a la hora de instanciar a los jugadores. El objetivo principal de este componente es definir un método común denominado *createTeam*, el cual cada implementación de *TeamFactory* debe implementar. En el contexto del juego, cada fábrica crea un equipo diferente proporcionando una implementación única del método *createTeam*. Esta función se encarga de instanciar y configurar los jugadores de cada equipo específico con una estrategia particular que determina el comportamiento de los mismos y con una skin identificatoria. Este enfoque permite una flexibilidad significativa al agregar nuevos equipos al juego sin necesidad de modificar el código principal.

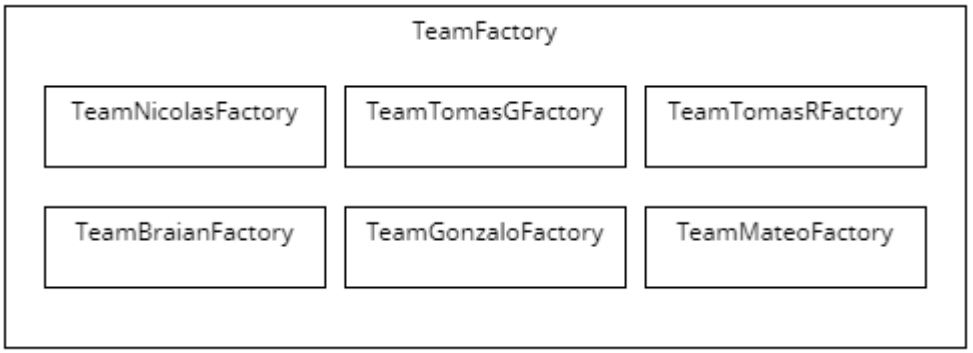

Figura 2 Vista sobre el apartado de Factories

Cabe recalcar que además de crear los equipos, *Game* configura las condiciones iniciales del juego (como el contador de goles en 0 y el posicionamiento de equipos y la pelota en la cancha).

Sprites

En esta capa (Backend) se gestionan los objetos visuales, conocidos como **Sprites** en Pygame. Existen sprites fijos como la cancha, y sprites que se actualizan y dibujan en cada fotograma del juego, concretamente los jugadores y la pelota en este caso.

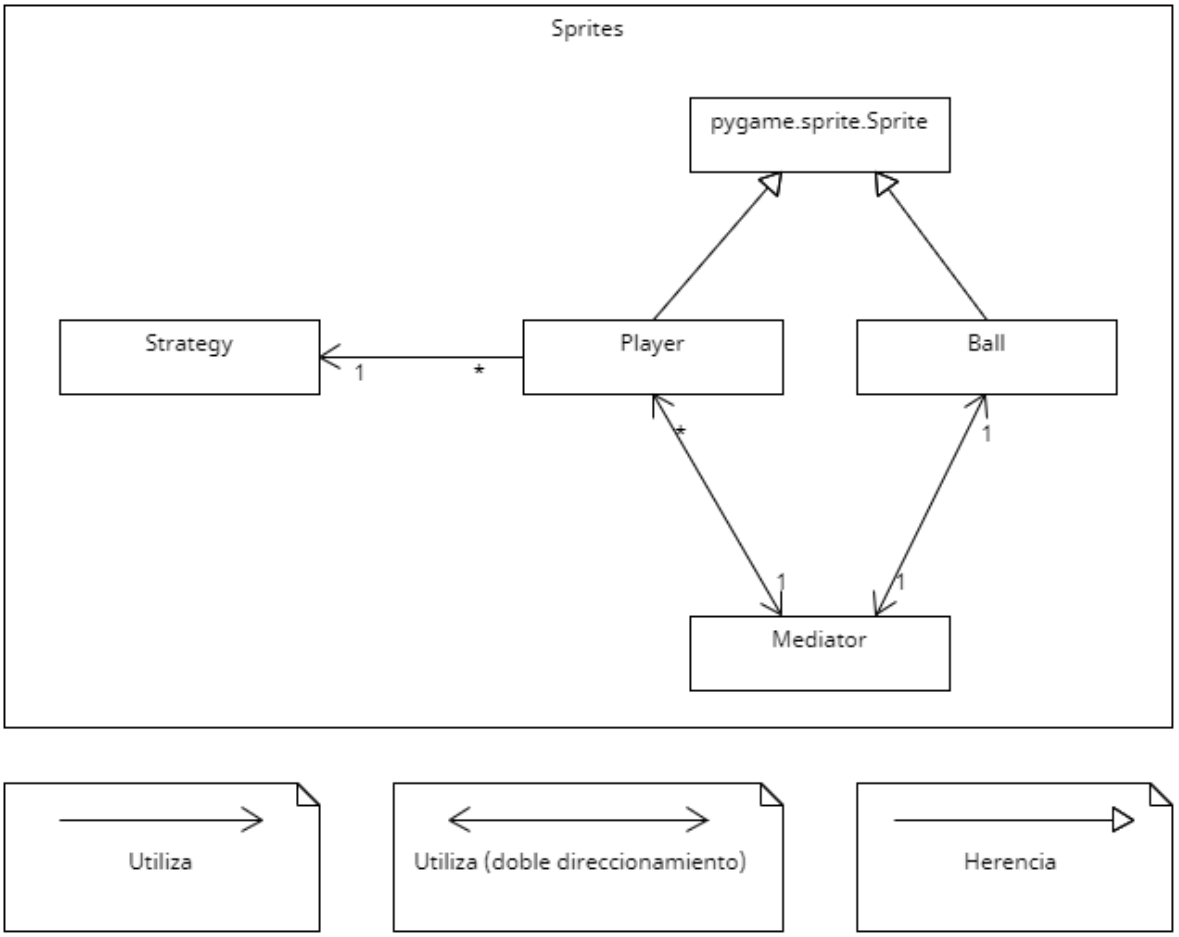

Figura 3 Vista sobre el apartado de Sprites

La actualización de cada fotograma se lleva a cabo gracias al método *play* alojado en *Game*, el cual es llamado desde la capa de interfaz y sirve como el punto de entrada principal del juego. Para ejecutar el juego, éste método tiene un bucle continuo (‘while True’) donde se ejecutan diferentes métodos que permiten correr los eventos y la lógica del juego, y mostrar los sprites por pantalla. Estos métodos son:

- *process_events:* verifica los eventos de entrada (teclas presionadas, clics del mouse, etc.) durante la ejecución del juego y realiza acciones específicas en respuesta a esos eventos, como pueden ser cerrar la ventana al presionar “X” en la esquina superior derecha de la pantalla, pausar el juego al pulsar SPACE, reiniciar el juego al presionar CTRL+R o volver al menú de selección de equipos pulsando ESC.
- *display_frame:* renderiza y muestra el contenido visual del juego en cada fotograma, incluyendo el dibujo de los sprites, contador de tiempo, marcador de goles y mensajes relevantes (mensaje de gol, de presentación de ganador o empate, de pausa, etc.).

- *run_logic:* actualiza la lógica del juego en cada fotograma, incluyendo el tiempo transcurrido, la posible pausa de entretiempo o finalización del juego y la actualización de la lógica de los sprites, la cual se logra al invocar al método *update* de los mismos, método que proviene de la herencia por parte de los sprites, concretamente las clases *Ball* y *Player,* de la clase *pygame.sprite.Sprite.*

La clase **pygame.sprite.Sprite** proporciona un mecanismo simple pero efectivo para gestionar la animación y el movimiento de sprites a través de la implementación del método *update* que determina cómo se comportan los sprites en cada fotograma.
La lógica de threads en Pygame con sprites se basa en el hecho de que cada sprite puede ejecutar su propio hilo de actualización de forma independiente. Esto permite que múltiples sprites se actualicen simultáneamente de manera concurrente. En este escenario, los jugadores y la pelota tienen cada uno su propio hilo de ejecución que gestiona sus movimientos, colisiones y comportamiento específico.

Autonomía de la pelota y los jugadores

Las clases *PlayerField* y *GoalKeeper* heredan de la clase base *Player*, y ambas implementan el método abstracto *update* que contiene la lógica de movimiento del jugador, el pase y el disparo de la pelota.

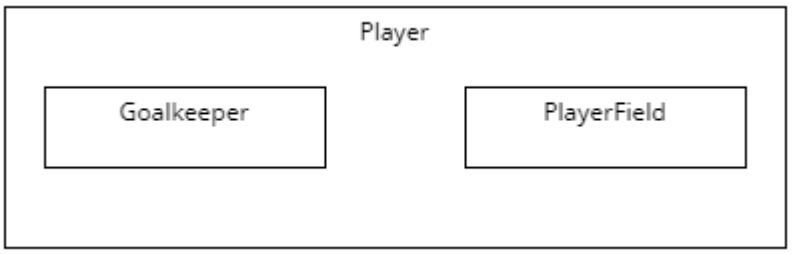

Figura 4 Vista sobre el apartado de Player

La elección y ejecución de estas acciones se delega a la implementación de la estrategia de cada equipo. La separación de las estrategias se logra a partir de la aplicación del patrón ***Strategy***, un patrón de tipo comportamental (behavioral pattern) cuyo propósito es mantener un conjunto de algoritmos (estrategias) de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades. Las instancias de *Player* de cada equipo poseen una estrategia específica establecida por la fábrica del equipo en concreto (cada implementación de estrategia hereda de la clase *Strategy*) que determina cómo accionan tales instancias ante la posesión de la pelota, como patear al arco, pasarla a un compañero (implica que elijan a quién pasársela) o moverse a lo largo de la cancha (precisa especificar el destino al que desplazarse).

Figura 5 Vista sobre el apartado Strategy

Cada estrategia debe implementar tres métodos:

- *with_ball:* retorna un entero que indica al jugador que tiene la pelota qué acción realizar. Las acciones posibles son patear al arco (SHOT), moverse (MOVE) o pasar la pelota a un compañero (PASS).
- *getProxPos:* retorna las coordenadas (x, y) a las que el jugador se debe mover.
- *where_to_pass:* retorna las coordenadas (x, y) a las que el jugador debe pasar la pelota.

```
def update(self):
        self.mediator.check_collision_with_ball(self)
        if self.hasBall:
            action = self.strategy.with_ball(self)
            # patear al arco
            if action == SHOT:
                self.mediator.shot_ball(self.team)
                self.hasBall = False
            # pasar pelota
            elif action == PASS:
                x, y = self.strategy.where_to_pass(self)
                self.mediator.pass_ball(x, y)
                self.hasBall = False
            # moverse
            elif action == MOVE:
                self.move()

        else:
            self.move()
```

Fragmento de código perteneciente al método update() común en PlayerField y GoalKeeper

Las circunstancias en las que estos métodos son invocados radican en la posesión o no posesión de la pelota por parte del jugador. Tal condición es representada a través de un booleano perteneciente a la clase *Player (has_ball)*. El método *check_collision_with_ball* se

ocupa de chequear si el jugador ha colisionado con la pelota, ajustando el valor del booleano en consecuencia. Si su valor es False, se envía un mensaje al método *move* el cual realiza la animación de desplazamiento desde donde se halla el jugador hasta la nueva posición indicada por la estrategia (invocación a *getProxPos*), siempre y cuando el movimiento sea válido, es decir, que el jugador no sobrepase los límites de la cancha. En función del tipo de la instancia de jugador (*PlayerField-Goalkeeper*), las coordenadas válidas varían, ya que el arquero solo puede moverse a lo largo del área grande de su equipo mientras que el jugador de campo está habilitado a recorrer toda la cancha. No obstante, si el valor del booleano es True, se invoca al método *with_ball* para definir la siguiente acción a llevar a cabo por el jugador con la pelota. Considerando la opción de pase, el método *where_to_pass* es invocado, si el jugador desea moverse se efectúa el procedimiento antes mencionado (cuando el jugador no tiene posesión de la pelota) y si la opción es patear al arco se envía un mensaje al método *shot_ball* común a todas las instancias de *Player.*
La autonomía de los jugadores se evidencia en cómo cada clase de jugador decide sus acciones sin depender directamente de instrucciones externas en tiempo real, ya que no reciben comandos desde el exterior. A su vez se manifiesta también en el hecho de que la lógica de movimiento y decisiones está integrada en cada clase de jugador, permitiéndoles actuar de manera independiente en respuesta a las condiciones del juego.

```
def animation_of_move(self):
        if self.move_speed != 0 and self.distance != 0:
            self.rect.x += self.dx / self.distance * self.move_speed
            self.rect.y += self.dy / self.distance * self.move_speed
            self.move_speed -= 1
        if self.move_speed <= 0 or
self.mediator.check_collision_with_players():
            self.move_speed = 35
            self.is_moving = False
```

Fragmento de código perteneciente al método animation_of_move() en Ball

Por otra parte, la autonomía de movimiento presente en la clase Ball se rige en función de su interacción con los jugadores. Al colisionar con un jugador, este último se hace de su posesión y puede decidir dirigirla a un compañero, al arco o simplemente moverse con ella como se mencionó anteriormente. En su método *update*, además de chequear los límites de la cancha, ya sea para identificar un gol o saques de arco o laterales que ocasionan el reposicionamiento de los jugadores en la cancha, realiza el llamado al método *animation_of_move* que implementa la animación de desplazamiento desde donde se haya la pelota hasta donde le indique el jugador que tiene su posesión.
Existe un caso especial que es cuando la pelota colisiona con dos o más jugadores. Para evitar que los mismos peleen por la posesión de la pelota por un tiempo indefinido, lo que ocurre es que el método *fighting_for_ball* que se ejecuta en el *update* de la clase *Ball* hace que la pelota salga disparada hacia una dirección aleatoria.

Arquitectura de eventos en la comunicación de sprites

Para establecer la comunicación e interacción entre los sprites se aplica el patrón de diseño ***Mediator***, el cual actúa como un punto central que promueve el desacoplamiento entre componentes como *PlayerField*, *Goalkeeper* y *Ball*. El *Mediator* gestiona la pelota y dos grupos de sprites que representan los equipos 1 y 2, compuestos por instancias de *PlayerField* y una única instancia de *Goalkeeper* cada uno. Todas estas clases poseen el Mediator, logrando que ninguna se conozca directamente pero que puedan interactuar y comunicarse entre sí. Este enfoque ofrece diversas ventajas, como la centralización de la lógica del juego, simplificación del mantenimiento y la expansión del código, facilitación de nuevas interacciones y establecimiento de una comunicación clara y estructurada. Asimismo, el *Mediator* permite que tanto jugadores como la pelota generen y consuman eventos esenciales para el juego, canalizando esta comunicación a través de un intermediario compartido al hacer uso del patrón *Observer* para manejar eventos, patrón que permite que objetos llamados observadores se suscriban y reciban notificaciones sobre cambios o eventos ocurridos en otro objeto, llamado sujeto u observado, facilitando la actualización automática de los observadores en respuesta a cambios en el sujeto sin la necesidad de acoplar directamente ambos componentes.

Este enfoque configura una **arquitectura de eventos**, un patrón de diseño de software creado para registrar, comunicar y procesar eventos entre servicios desacoplados, manteniendo la asincronía de los sistemas y permitiendo el intercambio de información y la realización de tareas. La arquitectura incluye elementos productores (publicadores) y consumidores (suscriptores) de eventos. Los productores detectan eventos, los representan como mensajes y los transmiten asincrónicamente a los consumidores a través de canales específicos. Los consumidores reciben notificaciones de los eventos, procesándolos o simplemente siendo afectados por ellos. La plataforma ejecuta la respuesta adecuada al evento y envía la actividad resultante al consumidor correspondiente, permitiendo así el flujo asincrónico de la información en el sistema.

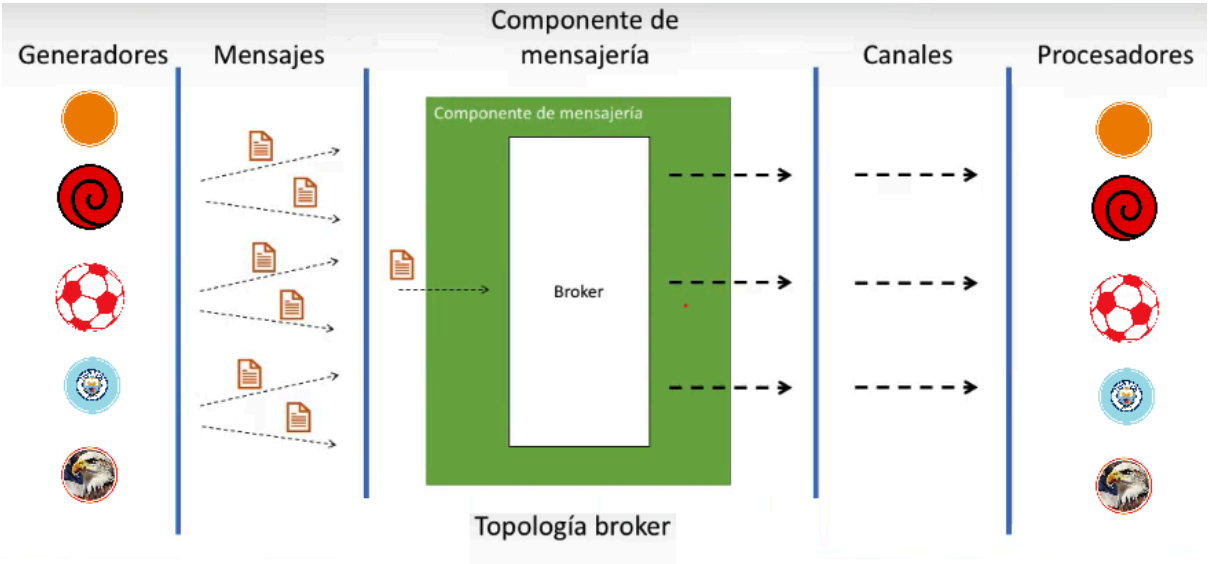

Figura 6 Esquema conceptual Arquitectura de Eventos (topología Broker)

La forma de organizar el componente de mensajería se conoce como topología. En este contexto se utiliza una *topología Broker*, el cual recibe los eventos de los generadores y los

transmite al canal apropiado para que los procesadores interesados reciban el mensaje. Mientras que el *Mediator* ejerce de broker, *PlayerField, Goalkeeper* y *Ball* desempeñan roles duales de consumidores y productores.
Entre los eventos principales que se desarrollan durante el juego tenemos:

- Pase de la Pelota:
    - Productores: *PlayerField* y *Goalkeeper* generan eventos al decidir pasar la pelota mediante la invocación del método *pass_ball* ubicado en el *Mediator,* el cual desencadena un desplazamiento de la pelota en dirección a un compañero del jugador que efectúa el pase.
    - Consumidor: La instancia de *Ball* consume el evento de pase para actualizar su posición utilizando el método *set_prox_pos* que recibe las coordenadas brindadas por *pass ball.*
- Disparo de la Pelota:
    - Productores: *PlayerField* y *Goalkeeper* generan eventos al decidir disparar la pelota por medio del llamado al método *shot ball* del *Mediator,* el cual ocasiona un desplazamiento de la pelota en dirección al arco rival dentro de un rango aleatorio que puede terminar en gol o en saque del arco rival.
    - Consumidor: La instancia de *Ball* recibe el evento de disparo para actualizar su posición usando el método *set_prox_pos* que recibe las coordenadas dadas por *shot_ball.*

- Detección de Gol/Lateral/Saque de Fondo:
    - Productor: *Ball* genera eventos al detectar situaciones como goles, laterales y saques de fondo, los cuales invocan a los métodos *restart_positions*, *restart_positcions_to_lateral* y *restart_positions_to_goal_kick* correspondientemente en base a los límites de la cancha que sean superados.
    - Consumidores: *PlayerField* y *Goalkeeper* consumen estos eventos para reiniciar sus posiciones según el evento desencadenante (lateral, gol o saque de arco).

Arquitectura de sistemas distribuidos y la ejecución concurrente de los componentes del sistema

Un **sistema distribuido** es un conjunto de programas informáticos que utilizan recursos computacionales en varios nodos de cálculo distintos para lograr un objetivo compartido común. Este tipo de sistemas usan nodos distintos para comunicarse y sincronizarse a través de una red común. Estos nodos suelen representar dispositivos de hardware físicos diferentes, pero también pueden representar procesos de software diferentes u otros sistemas encapsulados recursivos. La finalidad de los sistemas distribuidos es eliminar los cuellos de botella o los puntos de error centrales de un sistema.

En el marco de nuestra implementación, *PlayerField, Goalkeeper* y *Ball* emergen como nodos fundamentales distribuidos en diversos threads que operan de manera concurrente y aportan significativamente a la flexibilidad, escalabilidad y tolerancia a fallos del sistema.

Estas características son emblemáticas de una arquitectura de sistemas distribuidos, donde la descentralización de los componentes juega un papel clave.

La distribución de jugadores y la pelota en nodos independientes es esencial para materializar estas cualidades. La comunicación entre ellos se establece a través de un mediador (red común) que facilita la interacción sin que la falla de un nodo afecte el flujo de ejecución de los demás, obteniendo una considerable tolerancia a fallos.

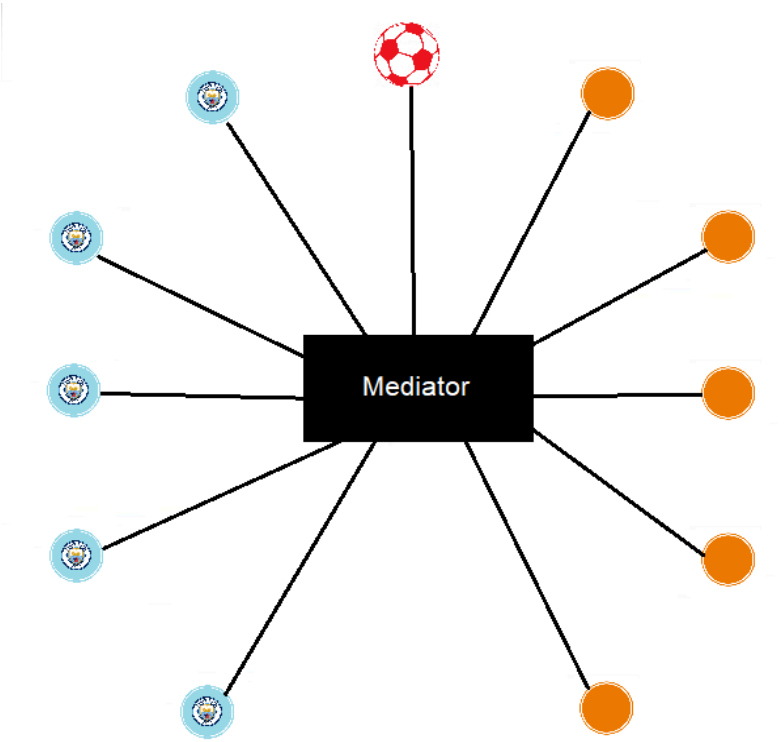

Figura 7 Esquema conceptual de la red de nodos referida a la arquitectura de sistemas distribuidos

Sumado a ello, esta arquitectura favorece la adición de nuevos nodos sin comprometer la estabilidad del sistema, permitiendo la incorporación de componentes adicionales, como ocurrió por ejemplo en el pasado cuando en las primeras etapas de diseño se implementó al arquero y a los jugadores de campo como una única clase (*Player*) y luego se optó por separar una parte de la implementación de ambos (concretamente la limitación de sus movimientos en la cancha), obteniendo como resultado el añadido de un nuevo nodo (clase *Goalkeeper*) (escalamiento horizontal). De igual manera, el agregado de funciones específicas se logra de manera eficiente (escalamiento vertical). Este enfoque resalta la capacidad inherente de una arquitectura de sistemas distribuidos para adaptarse y escalar de manera dinámica.

Desestimación de otras arquitecturas

- **Microservicios/Servicios**: Descartamos la arquitectura de microservicios debido a preocupaciones de rendimiento. La comunicación constante entre microservicios

mediante el envío de mensajes podría generar una carga significativa, impactando negativamente en el rendimiento del juego en tiempo real.

- **Monolítica**: La arquitectura monolítica fue descartada debido a la ejecución concurrente del juego en diferentes threads. Al tener nodos independientes ejecutándose simultáneamente, la arquitectura monolítica no se adapta a la naturaleza concurrente del sistema.
- **MVC (Modelo-Vista-Controlador)**: La aplicación del patrón MVC se descartó, ya que no contamos con un controlador interactivo. Dado que los jugadores son manipulados mediante inteligencia artificial y no hay un componente que reciba órdenes directas del cliente, la presencia de un controlador resultaría innecesaria en este contexto.
- **Cliente-Servidor**: La arquitectura cliente-servidor no es viable para nuestro sistema, ya que no hay una interacción directa del cliente con el juego. La participación del cliente se limita a la interfaz, donde selecciona las inteligencias artificiales que jugarán de manera autónoma sin intervención directa.
- **Master-Slave**: Descartamos la arquitectura master-slave debido a la falta de jerarquía entre los nodos del sistema. Los nodos se ejecutan de manera independiente, sin ser dirigidos por un nodo central, eliminando la necesidad y la aplicabilidad de una relación de maestro-esclavo en este contexto.

Bibliografía

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
2. Bass, L., Clements, P., & Kazman, R. (2012). Software Architecture in Practice (3rd ed.). Addison-Wesley.
3. Reactive Programming. (2023). Factory method. Reactive Programming https://reactiveprogramming.io/blog/es/patrones-de-diseno/factory-method
4. Medium. (2023). Patrón estrategia (Strategy Pattern). Medium. https://medium.com/all-you-need-is-clean-code/patr%C3%B3n-estrategia-strategy-pattern-654c6e9d2abe
5. Red Hat. (2023). What is event-driven architecture? Red Hat https://www.redhat.com/es/topics/integration/what-is-event-driven-architecture
6. TIBCO. (2023). What is event-driven architecture? TIBCO https://www.tibco.com/es/reference-center/what-is-event-driven-architecture
7. Atlassian. (2023). Distributed architecture. Atlassian https://www.atlassian.com/es/microservices/microservices-architecture/distributed-architecture
8. TecLab. (2023). Tipos de arquitecturas de software: ¿cuáles hay y en qué se diferencian? TecLab. https://www.teclab.edu.ar/tipos-de-arquitecturas-de-software-cuales-hay-y-en-que-se-diferencian
9. OpenWebinars. (2023). Arquitectura de software: qué es y qué tipos existen. OpenWebinars. https://openwebinars.net/blog/arquitectura-de-software-que-es-y-que-tipos-existen/