



Escuela de Ingeniería en Computadores

Documentación Tarea #4 DunCE Kong-Jr

Paradigmas de Programación Gr#2

Integrantes:

Jose Pablo Guerrero Duarte

Génesis Gabriela Conejo Montero

Luis Alejandro Vargas Benavides

Docente:

Marco Rivera Meneses

Fecha:

28 de noviembre del 2025

<b>Documentación Técnica</b>	<b>3</b>
<b>Descripción de la utilización de las estructuras de datos desarrolladas</b>	<b>3</b>
<b>Descripción detallada de los algoritmos desarrollados</b>	<b>4</b>
Problemas sin solución	5
Problemas encontrados	5
Plan de Actividades realizadas por estudiante	5
Conclusiones	6
Recomendaciones	6
Bibliografía	7
<b>Bitácora</b>	<b>8</b>

# Documentación Técnica

## Descripción de la utilización de las estructuras de datos desarrolladas

### De Factory:

Estructuras de Entidad:

**Clases Base (Enemy, Fruit):** Sirven como tipos genéricos (Enemy y Fruit) para que el sistema pueda manejar a todos los cocodrilos y frutas de manera uniforme, respectivamente.

**Implementaciones Concretas (RedCroc, BlueCroc, SimpleFruit):** Son las estructuras de datos que se instancian realmente. Almacenan la posición (x, y) y atributos específicos (como el valor en puntos de la SimpleFruit o el comportamiento del cocodrilo).

Estructura de Creación (Factory):

**Clase DefaultFactory:** Su función es encapsular la lógica de instanciación. Decide qué clase concreta de enemigo (RedCroc o BlueCroc) o fruta (SimpleFruit) debe crear según los parámetros, desacoplando el código que usa estos objetos de sus detalles de construcción.

### De UI:

**Server java:** La clase Server utiliza principalmente estructuras concurrentes y patrones de diseño (Singleton y Observador) para gestionar el estado del juego, las conexiones, el input de los clientes y la simulación.

**ConcurrentHashMap<Integer, Player> players:** Estructura clave para el estado del juego. Almacena el estado actual y los atributos (posición, puntaje) de cada jugador activo, indexado por su playerId.

**ConcurrentLinkedQueue<InputEvent> inputQueue:** Una cola concurrente que funciona como *buffer* de eventos de movimiento (InputEvent) enviados por los clientes. Esto permite que el hilo principal (tick()) procese las entradas de forma segura en un orden secuencial.

**ConcurrentHashMap<Integer, CopyOnWriteArrayList<ClientHandler>> spectatorsByPlayer:** Estructura que implementa el Patrón Observador. Mapea un jugador (sujeto) a una lista concurrente de espectadores (observadores) que reciben sus actualizaciones de estado en cada ciclo de simulación.

**char[][] MAP (Matriz):** Una matriz estática que define la geografía inmutable del nivel (paredes, plataformas, agua). Es consultada por las funciones de colisión y movimiento.

**ConcurrentHashMap<Integer, GameSession> sessions:** Mapea el playerId a su instancia de sesión de juego (GameSession), la cual contiene los objetos variables del juego (enemigos y frutas) específicos de esa partida.

**Clase Server (Singleton):** Garantiza que solo exista una instancia única para gestionar centralizadamente todo el estado global y las conexiones del servidor.

**Clase InputEvent:** Una estructura de datos inmutable para encapsular el *input* del cliente (ID, secuencia, dx, dy) antes de su procesamiento.

## Descripción detallada de los algoritmos desarrollados

### **Algoritmo de Movimiento y Comportamiento del Cocodrilo Azul (BlueCroc)**

El algoritmo, implementado en el método tick() del Servidor Java, gestiona el movimiento vertical ascendente y la desaparición del enemigo. Primero, ajusta la frecuencia de movimiento (stepTicks= 6-nivel) para incrementar la velocidad según el nivel. El avance solo ocurre si el tickCounter es divisible por stepTicks. Cuando se mueve, el cocodrilo sube una unidad vertical  $y=y-1$ . El mecanismo de autodestrucción se activa si la posición vertical alcanza o supera el límite superior del mapa.

### **Algoritmo de Movimiento y Comportamiento del Cocodrilo Rojo (RedCroc)**

El algoritmo del cocodrilo rojo, implementado en la clase RedCroc del servidor Java, gestiona un patrullaje vertical oscilatorio estrictamente confinado a una liana. En la inicialización, el constructor realiza una detección dinámica de límites escaneando la coordenada X del mapa para definir las fronteras verticales exactas de la liana (minLianaY, maxLianaY). La velocidad de movimiento es controlada por la frecuencia de ticks ajustada al nivel de dificultad.

### **Algoritmo de la Fábrica de Elementos del Juego (DefaultFactory)**

El algoritmo implementado en la clase DefaultFactory es una aplicación directa del Patrón de Diseño Factory, cuyo objetivo es encapsular la lógica de instanciación de las entidades del juego. Este patrón permite que el código principal del Servidor invoque la creación de objetos sin depender de sus clases concretas. El método createCrocodile gestiona la creación condicional de enemigos: verifica el parámetro de tipo string y retorna una instancia de RedCroc, por defecto, retorna una instancia de BlueCroc. De manera similar, el método createFruit simplemente retorna una nueva instancia de SimpleFruit con los parámetros de posición y puntos suministrados, garantizando que todos los elementos se creen de forma controlada y centralizada.

### **Algoritmo de Sincronización Global del Juego (Game Loop)**

El algoritmo del Game Loop, implementado en el método tick() del Servidor Java y ejecutado a una tasa fija (cada 125ms), es el corazón de la simulación. Se divide en tres fases críticas. Primero, procesa las entradas resolviendo conflictos para aplicar el único "mejor" movimiento por jugador (priorizando saltos) y actualiza la posición de Donkey Kong Jr. (verificando límites y colisiones con paredes). Segundo, aplica la Gravedad y Simulación: se aplica la gravedad a los jugadores que no están sobre una plataforma o liana, se llama al método tick() de todas las entidades para que se muevan, y se ejecuta la detección de colisiones (enemigos, agua y recolección de frutas), actualizando vidas y puntuaciones. Finalmente, en la fase de Notificación de Estado, el Servidor emite un mensaje STATE con la información actualizada (posición, vidas, score) a todos los clientes (jugadores y espectadores). Para optimizar el ancho de banda, la lista completa de enemigos solo se envía cada dos ticks.

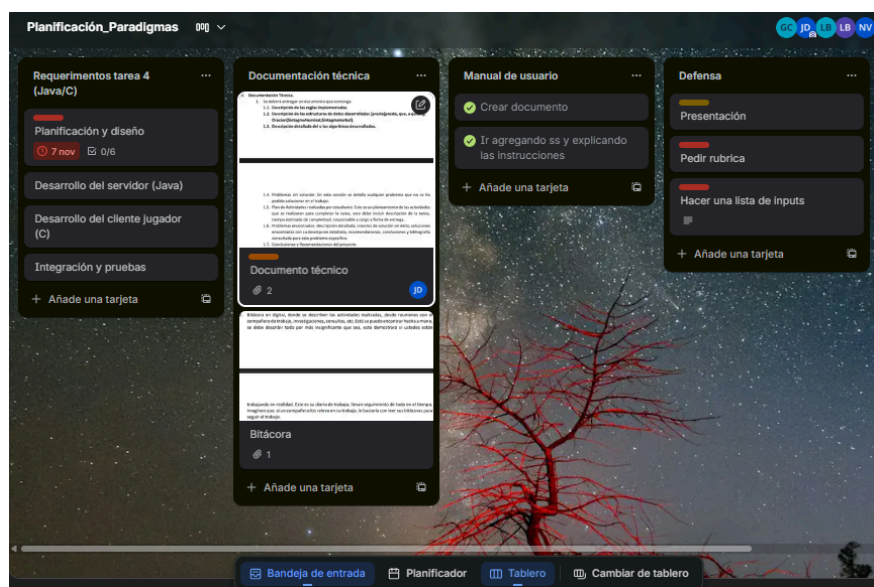
## Problemas sin solución

- Durante la fase de desarrollo del servidor en Java, se identificó un problema significativo al intentar utilizar la librería JavaFX para la interfaz de administración del juego. La dificultad principal no residió en la implementación del código en sí, sino en la correcta integración del JavaFX en los entornos de desarrollo de los integrantes del grupo.
- 

## Problemas encontrados

- Un reto importante en la gestión del tiempo es el requerimiento de mantener la Bitácora. El requerimiento de registrar todas las actividades con una minuciosidad tal que un compañero pueda hacer el trabajo solo consultando la bitácora está en competencia directa con el enfoque necesario para desarrollar el código. Administrar el tiempo entre la documentación detallada y la resolución de problemas técnicos críticos, como el funcionamiento de los sockets o la lógica del juego, es complicado; por lo tanto, a pesar de ser crucial este aspecto, se puede posponer con facilidad.

## Plan de Actividades realizadas por estudiante



Link de trello:

<https://trello.com/invite/b/68a80807f36a700f334f9c04/ATTI2a958cb542cc95ab402775430293ae8cEC5FD360/planificacionparadigmas>

## Conclusiones

El desarrollo del sistema distribuido "DonCEy Kong Jr." se concluye como un éxito en la integración de paradigmas y la arquitectura Cliente-Servidor. Se logró la separación efectiva de responsabilidades, cumpliendo con la implementación de la lógica del juego en Java (Orientado a Objetos) en el servidor y la interfaz gráfica y control en C (Imperativo) para los Clientes. El servidor Java demostró la capacidad de manejar la concurrencia al soportar la conexión simultánea de múltiples clientes, jugadores y espectadores. Este manejo de hilos garantiza que la lógica del juego se mantenga coherente para todos los participantes, lo que fue fundamental para el desarrollo del juego. Además, se estableció un canal de comunicación estable y funcional entre el cliente (C) y el servidor (Java) utilizando Sockets. En el servidor, se cumplieron los requisitos de POO implementando clases, paquetes y dos patrones de diseño, mientras que en el cliente C se utilizaron satisfactoriamente structs y listas ligadas para la manipulación de las estructuras de datos, lo que valida la comprensión y aplicación de ambos paradigmas en un sistema. El sistema logra su objetivo principal al ofrecer una plataforma funcional que simula la mecánica de juego de Donkey Kong Jr. y la aplicación de al menos dos patrones de diseño (excluyendo Singleton), asegurando un código modular y de alta calidad. La necesidad de soportar múltiples clientes (hasta 2 jugadores y espectadores) implicó el manejo de la concurrencia en el Servidor Java, garantizando que la lógica del juego se mantenga coherente para todos los participantes.

## Recomendaciones

Tras la implementación y validación de la lógica Cliente-Servidor, se recomienda enfocar los esfuerzos finales en optimizar la robustez de la comunicación y la mantenibilidad del código base para la entrega. Es fundamental integrar un mecanismo de heartbeat o keep-alive dentro del protocolo de Sockets; este sistema permitirá una detección proactiva de la caída de clientes por parte del Servidor Java, lo cual es vital para liberar recursos de hilo de manera oportuna y garantizar la continuidad estable de la simulación del juego. En el ámbito del Cliente C, se debe completar la estandarización de configuraciones asegurando que todas las constantes críticas (coordenadas de conexión, límites de movimiento y parámetros del game loop) residan en un archivo de cabecera independiente para facilitar el mantenimiento futuro y cualquier ajuste necesario. Finalmente, para demostrar la solidez del manejo de errores y dar cumplimiento a los requerimientos, se recomienda reforzar el logging, esto implica registrar explícitamente cualquier excepción que surja en el Servidor Java durante la ejecución, asegurando la trazabilidad y la corrección de fallos que pudieran surgir en la defensa.

## Bibliografía

Meza González, J. D. (2018). *Sockets en Java: Un sistema cliente-servidor con sockets*. Programarya.  
<https://www.programarya.com/Cursos-Avanzados/Java/Sockets>

RetroGames.cz. (s. f.). *Donkey Kong Jr. (NES) - online game*.  
[https://www.retrogames.cz/play\\_002-NES.php](https://www.retrogames.cz/play_002-NES.php)

# Bitácora

Fecha	Participantes	Objetivos	Actividades Realizadas	Acuerdos y Tareas Pendientes
24/10	Jose Pablo Génesis Conejo Luis Alejandro	Comprender qué se pedía realizar.	Los integrantes leen el documento para que en la próxima reunión estén claros.	Anotar cualquier duda y próxima reunión.
28/10	Jose Pablo Génesis Conejo Luis Alejandro	Comprender las instrucciones del trabajo de manera detallada.	Lectura del documento como equipo y anotaciones de dudas que surgieron.	Aclarar las dudas con el Profesor después de la clase.
10/11	Jose Pablo Génesis Conejo Luis Alejandro	Comprender el código que había hasta el momento.	Breve explicación del código que se tenía, se aclaró qué faltaba para continuar y organizar el tiempo.	Manejar el tiempo de forma adecuada y ciertas tareas en específico.
15/11	Jose Pablo Génesis Conejo Luis Alejandro	Detallar lista de prioridades por tiempo.	Compresión de que había para priorizar tareas.	Se dejó una lista de tareas para ir cumpliendo, tareas pequeñas para lapsos cortos.
23/11	Jose Pablo Génesis Conejo Luis Alejandro	Revisar detalles finales.	Verificación de que cosas estaban listas y cuales estaban faltantes.	La documentación quedó pendiente.
25/11	Jose Pablo Génesis Conejo Luis Alejandro	Revisión final.	Se corrió el juego.	Se revisó que estuviese todo preparado para la entrega.
28/11	Jose Pablo Génesis Conejo Luis Alejandro	Entrega de la Tarea	Se corrigieron cosas en la documentación de la tarea.	Se entregó el trabajo.