



UNSA

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

Memoria descriptiva

**MicroPatosMania - Videojuego
Multijugador**

I. INFORMACIÓN GENERAL	4
A. NOMBRE DEL PROYECTO	4
B. INTEGRANTES DEL EQUIPO	4
C. INSTITUCIÓN Y CURSO}	4
D. OBJETIVO GENERAL	4
II. DESCRIPCIÓN DEL PROYECTO	4
A. PROPÓSITO	4
B. CARACTERÍSTICAS PRINCIPALES	4
III. ARQUITECTURA TÉCNICA	5
A. TECNOLOGÍAS UTILIZADAS	5
B. ESTRUCTURA DEL PROYECTO	5
C. COMPONENTES CLAVE	6
1. MÓDULO DE RED	6
2. MÓDULO DE MINIJUEGOS	6
3. MÓDULO DE PANTALLAS	7
4. SISTEMA UI (ui/)	7
IV. FLUJO DE FUNCIONAMIENTO	7
A. FLUJO DEL JUGADOR	7
B. ARQUITECTURA DE RED	7
A. DEPENDENCIAS TÉCNICAS	7
B. REQUISITOS DEL SISTEMA	7
C. HERRAMIENTAS OPCIONALES	7
VI. COMANDO DE EJECUCIÓN	8
A. CON GRADLE	8
B. CON JUST (ALTERNATIVA)	8
VII. PATRONES DE DISEÑO UTILIZADOS	8
A. FACTORY PATTERN: MINIGAMEFACTORY	8
1. DESCRIPCIÓN	8
2. PROBLEMA	8
3. IMPLEMENTACIÓN	9
4. VENTAJAS	10
B. OBSERVER PATTERN: SISTEMA DE EVENTOS DE RED	10
1. DESCRIPCIÓN	10
2. PROBLEMA	10
3. IMPLEMENTACIÓN	11
4. VENTAJAS	13
C. MVC PATTERN: SEPARACIÓN DE LÓGICA, VISTA Y PRESENTACIÓN	13
1. DESCRIPCIÓN	13
2. PROBLEMA	13
3. IMPLEMENTACIÓN	14
4. VENTAJAS	16
D. SINGLETON PATTERN: NETWORKMANAGER, MAIN	16

1. DESCRIPCIÓN	16
2. PROBLEMA	16
3. IMPLEMENTACIÓN	16
4. VENTAJAS	18
E. STRATEGY PATTERN: DIFERENTES MINIJUEGOS	18
1. DESCRIPCIÓN	18
2. PROBLEMA	18
3. IMPLEMENTACIÓN	19
4. VENTAJAS	21
VIII. DESAFÍOS Y SOLUCIONES	21
A. DESAFÍO 01: SINCRONIZACIÓN EN TIEMPO REAL	21
1. PROBLEMÁTICA	21
a) LATENCIA DE RED VARIABLE	21
b) PÉRDIDA DE PAQUETES	22
c) CONGESTIÓN DE RED	22
d) DESFASE TEMPORAL	22
2. SOLUCIÓN: KRYONET Y SERIALIZACIÓN EFICIENTE	22
3. VENTAJAS	24
B. DESAFÍO 02: GESTIÓN DE MÚLTIPLES MINIJUEGOS	24
1. PROBLEMÁTICA	24
a) LÓGICA DISTINTA	24
b) REQUISITOS DE RED DIFERENTES	24
c) INTERFACES GRÁFICAS PERSONALIZADAS	24
d) ESTADOS COMPLEJOS	24
2. SOLUCIÓN: FACTORY + STRATEGY + OBSERVER PATTERNS	25
3. VENTAJAS	26
C. DESAFÍO 03: INTERFAZ GRÁFICA COMPLEJA	27
1. PROBLEMÁTICA	27
a) MÚLTIPLES PANTALLAS	27
b) COMPONENTES PERSONALIZADOS NECESARIOS	27
c) TRANSICIONES SUAVES	27
d) RESPONSIVIDAD	27
2. SOLUCIÓN: SCENE2D + COMPONENTES PERSONALIZADOS	27
3. VENTAJAS	31
IX. ESTADO ACTUAL Y PRÓXIMOS PASOS	31
A. ESTADO	31
B. MEJORAS FUTURAS	32
X. REFERENCIAS Y RECURSOS	32

I. INFORMACIÓN GENERAL

A. NOMBRE DEL PROYECTO

MicroPatosMania: Videojuego multijugador en tiempo real con múltiples minijuegos competitivos.

B. INTEGRANTES DEL EQUIPO

- CACERES RUIZ, Johann Andre
- GUTIERREZ CCAMA, Juan Diego
- JARA MAMANI, Mariel Alisson
- MESTAS ZEGARRA, Christian Raul
- NOA CAMINO, Yenaro Joel
- VALDIVIA SEGOVIA, Ryan Fabian

C. INSTITUCIÓN Y CURSO}

- Universidad: UNSA (Universidad Nacional de San Agustín de Arequipa)
- Carrera: Ingeniería de Sistemas
- Curso: Tecnología de Objetos (Sexto Semestre)

D. OBJETIVO GENERAL

Desarrollar un videojuego multijugador en red que integre múltiples minijuegos, permitiendo que los jugadores compitan en tiempo real a través de una arquitectura cliente-servidor.

II. DESCRIPCIÓN DEL PROYECTO

A. PROPÓSITO

MicroPatosMania es un juego de múltiples minijuegos competitivos que permite a los jugadores:

- Conectarse a través de una red local o internet
- Crear y unirse a salas de juego
- Competir en diferentes tipos de minijuegos
- Visualizar rankings y puntuaciones en tiempo real

B. CARACTERÍSTICAS PRINCIPALES

- **Arquitectura Cliente-Servidor:** Comunicación en tiempo real usando Kryonet
- **Múltiples Minijuegos:**
 - Ball Movement: Control de movimiento de bola
 - Catch Them All: Juego de captura con patos

- Sumo: Juego de enfrentamiento tipo sumo
- The Finale: Minijuego final
- **Gestión de Salas:** Crear, buscar y unirse a salas de juego
- **Sistema de Puntuaciones:** Tracking de puntos y rankings
- **Menú Principal y Pantallas de Transición:** Interfaz gráfica completa
- **Modo Espectador:** Visualización de partidas en progreso

III. ARQUITECTURA TÉCNICA

A. TECNOLOGÍAS UTILIZADAS

- **Lenguaje:** Java (JDK 21+)
- **Framework Gráfico:** libGDX
- **Networking:** Kryonet (serialización y comunicación de objetos)
- **Sistema de Build:** Gradle
- **Plataforma Gráfica:** LWJGL3 (Lightweight Java Game Library 3)
- **IDE:** IntelliJ IDEA / Eclipse

B. ESTRUCTURA DEL PROYECTO

```
None
mpm/
└── core/ # Módulo principal (lógica del juego)
    ├── src/main/java/to/mpm/
    │   ├── Main.java # Clase principal del juego
    │   ├── minigames/ # Lógica de minijuegos
    │   │   ├── Minigame.java
    │   │   ├── MinigameFactory.java
    │   │   ├── MinigameType.java
    │   │   └── ballmovement/
    │   ├── catchThemAll/
    │   ├── sumo/
    │   └── theFinale/
    ├── network/ # Sistema de red
    │   ├── NetworkManager.java
    │   ├── NetworkClient.java
    │   ├── NetworkServer.java
    │   ├── KryoClassRegistrar.java
    │   ├── Packets.java
    │   └── handlers/
    ├── screens/ # Pantallas de interfaz
    │   ├── MainMenuScreen.java
    │   ├── CreateRoomScreen.java
    │   ├── JoinLobbyScreen.java
    │   ├── GameScreen.java
    │   └── ...
    └── ui/ # Componentes UI
        ├── UIStyles.java
```

```
| | |   └── components/
| | |   └── transitions/
| | |   └── utils/ # Utilidades
| | |   ├── PlayerData.java
| | |   ├── DebugKeybinds.java
| | |   └── FirewallHelper.java
| | └── build.gradle
|
└── lwjgl3/ # Módulo de lanzamiento LWJGL3
    ├── src/main/java/to/mpm/lwjgl3/
    |   ├── Lwjgl3Launcher.java
    |   └── StartupHelper.java
    └── build.gradle
|
└── assets/ # Recursos del juego
    └── ui/
        ├── font.fnt
        ├── uiskin.json
        └── uiskin.atlas
|
└── scripts/ # Scripts de utilidad
    ├── run-many.ps1
    └── run-many.sh
|
└── build.gradle # Configuración de build principal
└── settings.gradle # Configuración de submódulos
└── gradle.properties # Propiedades de Gradle
└── Justfile # Automatización de tareas
└── Doxyfile # Configuración de documentación
└── README.md # Documentación principal
```

C. COMPONENTES CLAVE

1. MÓDULO DE RED

- **NetworkManager:** Gestor central de comunicación
- **NetworkServer:** Servidor de juego
- **NetworkClient:** Cliente de juego
- **KryoClassRegistrar:** Registro de clases serializables
- **Handlers:** Procesamiento de paquetes de red

2. MÓDULO DE MINIJUEGOS

- **MinigameFactory:** Patrón Factory para crear minijuegos
- **MinigameType:** Enumeración de tipos de minijuegos

- Cada minijuego con sus propias lógicas de physics, rendering e input

3. MÓDULO DE PANTALLAS

- Navegación entre diferentes pantallas del juego
- Gestión de estado de aplicación

4. SISTEMA UI (ui/)

- Componentes personalizados
- Transiciones de pantalla
- Sistema de estilos

IV. FLUJO DE FUNCIONAMIENTO

A. FLUJO DEL JUGADOR

- 1. Inicio:** Pantalla principal (MainMenuScreen)
- 2. Crear/Unirse a Sala:** CreateRoomScreen o JoinLobbyScreen
- 3. Lobby:** Espera de jugadores (LobbyScreen)
- 4. Selección de Minijuego:** Selección automática o manual
- 5. Gameplay:** Ejecución del minijuego (GameScreen)
- 6. Resultados:** Visualización de puntuaciones (ResultsScreen)
- 7. Repetición o Fin:** Vuelta al menú o siguiente ronda

B. ARQUITECTURA DE RED

- **Modelo:** Cliente-Servidor con Kryonet
- **Comunicación:** Serialización de objetos de paquetes
- **Sincronización:** Sincronización de estado en tiempo real

V. DEPENDENCIAS

A. DEPENDENCIAS TÉCNICAS

- libGDX: Framework gráfico
- Kryonet: Comunicación en red
- Gradle Wrapper: Gestión de build

B. REQUISITOS DEL SISTEMA

- Java JDK 21 o superior
- Gradle (incluido wrapper)
- Conexión de red para multijugador

C. HERRAMIENTAS OPCIONALES

- Doxygen 1.15+ (documentación)
- Graphviz (diagramas)

- Python 3+ (servir documentación)
- LaTeX (documentación PDF)
- Just (automatización de tareas)

VI. COMANDO DE EJECUCIÓN

A. CON GRADLE

Shell

```
# Construir el proyecto  
./gradlew build  
  
# Ejecutar la aplicación  
./gradlew lwjgl3:run  
  
# Generar documentación  
./gradlew docs # (si está configurado)  
  
# Limpiar archivos de build  
./gradlew clean
```

B. CON JUST (ALTERNATIVA)

Shell

```
just build  
just run  
just docs  
just clean
```

VII. PATRONES DE DISEÑO UTILIZADOS

A. FACTORY PATTERN: MINIGAMEFACTORY

1. DESCRIPCIÓN

El patrón Factory es un patrón de diseño creacional que proporciona una interfaz para crear objetos sin especificar sus clases exactas. En MicroPatosMania, este patrón se utiliza para crear diferentes tipos de minijuegos de forma centralizada y flexible.

2. PROBLEMA

Sin el Factory Pattern, la creación de minijuegos estaría dispersa en múltiples lugares del código, haciendo difícil:

- Mantener el código cuando se agregan nuevos minijuegos
- Cambiar la lógica de inicialización de minijuegos
- Garantizar que todos los minijuegos se crean correctamente

3. IMPLEMENTACIÓN

Java

```

Enumeración de tipos:
```java
public enum MinigameType {
 BALL_MOVEMENT,
 CATCH_THEM_ALL,
 SUMO,
 THE_FINAL
}
```

Interfaz base:
```java
public interface Minigame {
 void initialize();
 void update(float delta);
 void render(SpriteBatch batch, ShapeRendererer
shapeRenderer);
 void handleInput(float delta);
 boolean isFinished();
 Map<Integer, Integer> getScores();
 int getWinnerId();
 void dispose();
 void resize(int width, int height);
}
```

Factory:
```java
public class MinigameFactory {
 public static Minigame createMinigame(MinigameType type,
int localPlayerId) {
 switch (type) {
 case BALL_MOVEMENT:
 return new BallMovementMinigame(localPlayerId);
 case CATCH_THEM_ALL:
 return new CatchThemAllMinigame(localPlayerId);
 case SUMO:
 return new SumoMinigame(localPlayerId);
 case THE_FINAL:
 return new TheFinaleMinigame(localPlayerId);
 }
 }
}
```

```

```

        default:
            Gdx.app.error("MinigameFactory", "Tipo de
minijuego desconocido: " + type);
            return null;
        }
    }
...
}

Uso en GameScreen:
```java
public GameScreen(Main game, MinigameType minigameType, int
currentRound, int totalRounds) {
 this.game = game;
 this.minigameType = minigameType;
 this.currentRound = currentRound;
 this.totalRounds = totalRounds;
}

// En el método show()
currentMinigame = MinigameFactory.createMinigame(minigameType,
localPlayerId);
currentMinigame.initialize();
```

```

4. VENTAJAS

- Centraliza la creación de minijuegos
- Facilita agregar nuevos minijuegos sin modificar código existente
- Desacopla la GameScreen de las implementaciones específicas
- Garantiza que todos los minijuegos se inicialicen correctamente

B. OBSERVER PATTERN: SISTEMA DE EVENTOS DE RED

1. DESCRIPCIÓN

El patrón Observer permite que múltiples objetos (observadores) se suscriban a eventos de un objeto central (sujeto). Cuando ocurre un evento, el sujeto notifica automáticamente a todos los observadores.

2. PROBLEMA

En una aplicación multijugador con red, múltiples componentes necesitan reaccionar a eventos de red (paquetes recibidos) sin que el NetworkManager necesite conocer todos los detalles de estos componentes. Sin el Observer Pattern:

- La lógica de red sería monolítica y difícil de mantener

- Sería complejo agregar nuevos manejadores de paquetes
- Habría acoplamiento fuerte entre componentes

3. IMPLEMENTACIÓN

Java

```

```java
public interface NetworkPacketHandler<T extends PacketContext>
{
 void handle(T context, NetworkPacket packet);
}

public interface ClientPacketHandler extends
NetworkPacketHandler<ClientPacketContext> {
}

public interface ServerPacketHandler extends
NetworkPacketHandler<ServerPacketContext> {
}
```

Contexto de paquete:
```java
public class ClientPacketContext {
 public NetworkClient client;

 public ClientPacketContext(NetworkClient client) {
 this.client = client;
 }
}
```

NetworkManager (Sujeto):
```java
public class NetworkManager {
 private static NetworkManager instance;
 private List<ClientPacketHandler> clientHandlers = new
ArrayList<>();

 public void registerClientHandler(ClientPacketHandler
handler) {
 if (client != null) {
 clientHandlers.add(handler);
 client.registerHandler(handler);
 }
 }
}
```

```

```

        public void unregisterClientHandler(ClientPacketHandler
handler) {
    if (client != null) {
        clientHandlers.remove(handler);
        client.unregisterHandler(handler);
    }
}

public static NetworkManager getInstance() {
    if (instance == null) {
        instance = new NetworkManager();
    }
    return instance;
}
```
```

```

Implementación de Observer (Ejemplo en CatchThemAll):

```

```java
public class CatchThemAllClientHandler implements
ClientPacketHandler {
 private CatchThemAllMinigame minigame;

 public CatchThemAllClientHandler(CatchThemAllMinigame
minigame) {
 this.minigame = minigame;
 }

 @Override
 public void handle(ClientPacketContext context,
NetworkPacket packet) {
 if (packet instanceof CatchThemAllPackets.PlayerUpdate)
{
 CatchThemAllPackets.PlayerUpdate update =
(CatchThemAllPackets.PlayerUpdate) packet;
 minigame.updatePlayerPosition(update.playerId,
update.x, update.y);
 }
 }
}
```
```

```

Registro de observadores en minijuegos:

```

```java
@Override
public void initialize() {
    NetworkManager nm = NetworkManager.getInstance();

```

```
// Crear y registrar observador
clientHandler = new CatchThemAllClientHandler(this);
nm.registerClientHandler(clientHandler);

if (nm.isHost()) {
    serverRelay = new CatchThemAllServerRelay(this);
    nm.registerServerHandler(serverRelay);
}
...
```
Limpieza de observadores:
```java
@Override
public void dispose() {
    NetworkManager nm = NetworkManager.getInstance();
    nm.unregisterClientHandler(clientHandler);
    nm.unregisterServerHandler(serverRelay);
}
```
...
```

#### 4. VENTAJAS

- Cada minijuego puede registrar sus propios manejadores sin afectar otros
- Facilita pruebas unitarias
- Los manejadores pueden ser agregados/removidos dinámicamente
- Separación clara de responsabilidades entre componentes

### C. MVC PATTERN: SEPARACIÓN DE LÓGICA, VISTA Y PRESENTACIÓN

#### 1. DESCRIPCIÓN

El patrón MVC (Model-View-Controller) separa la aplicación en tres capas:

- Model: Lógica de negocio y estado
- View: Presentación visual
- Controller: Lógica de interacción y control

#### 2. PROBLEMA

Sin MVC, la lógica de juego, la renderización y la entrada del usuario estarían mezcladas en un único archivo, causando:

- Código difícil de mantener y probar
- Cambios visuales que afectan la lógica
- Dificultad para reutilizar componentes

### 3. IMPLEMENTACIÓN

Java

```

```java
// SumoMinigame.java - Contiene la lógica del juego
public class SumoMinigame implements Minigame {
    private final IntMap<SumoPlayer> players = new IntMap<>();
    private final Map<Integer, Integer> scores = new
HashMap<>();
    private boolean finished = false;

    @Override
    public void initialize() {
        // Inicializar lógica de juego
        NetworkManager nm = NetworkManager.getInstance();
        nm.registerAdditionalClasses(
            SumoPackets.PlayerKnockback.class,
            SumoPackets.PlayerFell.class
        );
        scores.put(localPlayerId, 0);
    }

    @Override
    public void update(float delta) {
        // Actualizar lógica
        for (IntMap.Entry<SumoPlayer> entry : players) {
            entry.value.update(delta);
        }
        if (NetworkManager.getInstance().isHost() && !finished)
    {
        checkCollisions();
        checkFallout();
    }
}

    @Override
    public Map<Integer, Integer> getScores() {
        return scores;
    }
}
```
```
**View (Rendering):**
```
java
// GameScreen.java - Contiene la presentación
public class GameScreen implements Screen {
 private SpriteBatch batch;
 private ShapeRenderer shapeRenderer;

```

```

 private Stage uiStage;
 private Label scoreLabel;
 private Label timerLabel;

 @Override
 public void render(float delta) {
 // Limpiar pantalla
 Gdx.gl.glClearColor(0.1f, 0.1f, 0.1f, 1);
 Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

 // Renderizar minijuego
 batch.begin();
 currentMinigame.render(batch, shapeRenderer);
 batch.end();

 // Actualizar y renderizar UI
 updateUI();
 uiStage.draw();
 }

 private void updateUI() {
 Map<Integer, Integer> scores =
 currentMinigame.getScores();
 int currentScore = scores.getOrDefault(localPlayerId,
 0);
 scoreLabel.setText("Score: " + currentScore);
 timerLabel.setText("Time: " + (int)gameTimer);
 }
 ...
}

Controller (Input & Interaction):
```
java
// En cada minijuego - Manejo de entrada
@Override
public void handleInput(float delta) {
    if (Gdx.input.isKeyPressed(Input.Keys.LEFT)) {
        localPlayer.moveLeft(delta);
    }
    if (Gdx.input.isKeyPressed(Input.Keys.RIGHT)) {
        localPlayer.moveRight(delta);
    }
}

// En GameScreen - Orquestación
@Override
public void render(float delta) {
    // Procesar entrada
}
```

```

```

 currentMinigame.handleInput(delta);

 // Actualizar modelo
 currentMinigame.update(delta);

 // Renderizar vista
 batch.begin();
 currentMinigame.render(batch, shapeRenderer);
 batch.end();
 }
...

```

#### 4. VENTAJAS

- La lógica de juego es independiente de la renderización
- Facilita cambiar visualización sin afectar la lógica
- Permite pruebas independientes de cada componente
- Cada capa tiene una responsabilidad única

### D. SINGLETON PATTERN: NETWORKMANAGER, MAIN

#### 1. DESCRIPCIÓN

El patrón Singleton garantiza que una clase tenga una única instancia en toda la aplicación y proporciona un punto de acceso global a esa instancia.

#### 2. PROBLEMA

Para sistemas centralizados como network management y el juego principal, es crítico tener una única instancia para evitar:

- Múltiples servidores escuchando en el mismo puerto
- Inconsistencia de estado entre instancias
- Consumo innecesario de recursos

#### 3. IMPLEMENTACIÓN

Java

```

```java
public class NetworkManager {
    private static NetworkManager instance; // Variable
    estática privada
    private NetworkServer server;
    private NetworkClient client;
}

```

```

private boolean isHost;

// Constructor privado para evitar instanciación externa
private NetworkManager() {
}

// Método para obtener la instancia única
public static NetworkManager getInstance() {
    if (instance == null) {
        instance = new NetworkManager();
    }
    return instance;
}

public void hostGame(String hostPlayerName, int port)
throws IOException {
    if (server != null) {
        Gdx.app.log("NetworkManager", "Server is already
running");
        return;
    }
    isHost = true;
    server = new NetworkServer();
    server.start(port);
    client = new NetworkClient();
    client.connect("127.0.0.1", port, hostPlayerName);
}

public void sendPacket(NetworkPacket packet) {
    if (client != null && client.isConnected()) {
        client.send(packet);
    }
}
```
...
```
**Main (Clase del Juego):**
```
java
public class Main extends Game {
 public SpriteBatch batch;
 public DebugKeybinds debugKeybinds;
 private SettingsOverlayManager settingsOverlayManager;

 @Override
 public void create() {
 batch = new SpriteBatch();
 debugKeybinds = new DebugKeybinds(this);
 settingsOverlayManager = new SettingsOverlayManager();
 }
}

```

```
 setScreen(new MainMenuScreen(this));
 DebugKeybinds.printHelp();
 }

 public void toggleSettings() {
 settingsOverlayManager.toggle();
 }
}

```
**Uso en toda la aplicación:**  

```java
// En cualquier clase del proyecto
NetworkManager nm = NetworkManager.getInstance();
nm.sendPacket(packet);
nm.hostGame("Player", 8080);
nm.joinGame("192.168.1.100", 8080, "Player");
```
```

```

#### 4. VENTAJAS

- Garantiza una única conexión de red
- Punto de acceso centralizado desde cualquier punto del código
- Facilita la gestión de recursos compartidos
- Evita conflictos de sincronización

### E. STRATEGY PATTERN: DIFERENTES MINIJUEGOS

#### 1. DESCRIPCIÓN

El patrón Strategy define una familia de algoritmos, encapsula cada uno en una clase, y los hace intercambiables. Permite que el algoritmo varíe independientemente del cliente que lo utiliza.

#### 2. PROBLEMA

Cada minijuego tiene lógica completamente diferente, pero todos deben integrarse de la misma manera con el sistema principal:

- Ball Movement: Control de trayectoria de una bola
- Catch Them All: Persecución de entidades
- Sumo: Combate competitivo
- The Finale: Lógica personalizada

Sin Strategy Pattern, GameScreen tendría que conocer todos los detalles de cada minijuego.

### 3. IMPLEMENTACIÓN

Java

```

```java
public interface Minigame {
    void initialize();
    void update(float delta);
    void render(SpriteBatch batch, ShapeRenderer
shapeRenderer);
    void handleInput(float delta);
    boolean isFinished();
    Map<Integer, Integer> getScores();
    int getWinnerId();
    void dispose();
    void resize(int width, int height);
}
```

Estrategia 1: CatchThemAll
```java
public class CatchThemAllMinigame implements Minigame {
    private final IntMap<Player> players = new IntMap<>();
    private final List<Duck> ducks = new ArrayList<>();
    private DuckSpawner duckSpawner;
    private final Map<Integer, Integer> scores = new
HashMap<>();

    @Override
    public void initialize() {
        duckSpawner = new DuckSpawner();
        // Inicializar patos y jugadores
    }

    @Override
    public void update(float delta) {
        duckSpawner.update(delta);
        for (Duck duck : ducks) {
            duck.update(delta);
        }
        // Lógica de captura
        detectCatches();
    }

    @Override
        public void render(SpriteBatch batch, ShapeRenderer
shapeRenderer) {
            for (Player player : players.values()) {
                player.render(batch);
            }
        }
}
```

```

```

 }
 for (Duck duck : ducks) {
 duck.render(batch);
 }
 }
}

```
**Estrategia 2: SumoMinigame**
```java
public class SumoMinigame implements Minigame {
 private final IntMap<SumoPlayer> players = new IntMap<>();
 private static final float MAP_RADIUS = 200f;
 private final Map<Integer, Integer> scores = new
 HashMap<>();

 @Override
 public void initialize() {
 for (int i = 0; i < playerCount; i++) {
 spawnPlayer(i);
 }
 }

 @Override
 public void update(float delta) {
 for (SumoPlayer player : players.values()) {
 player.update(delta);
 }
 checkCollisions();
 checkFallout();
 }

 @Override
 public void render(SpriteBatch batch, ShapeRenderer
 shapeRenderer) {
 shapeRenderer.begin(ShapeRenderer.ShapeType.Filled);
 shapeRenderer.circle(MAP_CENTER_X, MAP_CENTER_Y,
 MAP_RADIUS);
 shapeRenderer.end();

 for (SumoPlayer player : players.values()) {
 player.render(batch);
 }
 }
}

```
**GameScreen - Cliente (No necesita conocer detalles):**

```

```

```java
public class GameScreen implements Screen {
 private Minigame currentMinigame;

 @Override
 public void show() {
 currentMinigame = MinigameFactory.createMinigame(minigameType, localPlayerId);
 currentMinigame.initialize();
 }

 @Override
 public void render(float delta) {
 currentMinigame.handleInput(delta);
 currentMinigame.update(delta);

 batch.begin();
 currentMinigame.render(batch, shapeRenderer);
 batch.end();

 if (currentMinigame.isFinished()) {
 showResults();
 }
 }
}
```

```

4. VENTAJAS

- GameScreen no necesita cambiar cuando se agregan nuevos minijuegos
- Cada estrategia (minijuego) puede evolucionar independientemente
- Facilita pruebas unitarias de cada minijuego
- Permite intercambiar minijuegos sin afectar el flujo del juego

VIII. DESAFÍOS Y SOLUCIONES

A. DESAFÍO 01: SINCRONIZACIÓN EN TIEMPO REAL

1. PROBLEMÁTICA

En un juego multijugador en red, múltiples clientes deben estar sincronizados en tiempo real para que todos los jugadores vean el mismo estado del juego. Los principales problemas incluyen:

a) LATENCIA DE RED VARIABLE

- Diferentes jugadores experimentan diferentes latencias
- Un jugador puede percibir eventos en diferente orden

- Causa desincronización en el estado del juego

b) PÉRDIDA DE PAQUETES

- La red puede perder paquetes durante la transmisión
- Los cambios de estado no se transmiten completamente
- Causa inconsistencia en el estado de los jugadores

c) CONGESTIÓN DE RED

- Enviar toda la información de estado cada frame es ineficiente
- Bandwidth limitado en conexiones deficientes
- Requiere optimización del tráfico de red

d) DESFASE TEMPORAL

- Cada cliente tiene su propio reloj (puede no estar sincronizado)
- Eventos pueden procesarse en orden incorrecto
- Causa "desincronización" visual

2. SOLUCIÓN: KRYONET Y SERIALIZACIÓN EFICIENTE

Java

```
```java
public class NetworkManager {
 private static NetworkManager instance;
 private NetworkServer server;
 private NetworkClient client;

 public void sendPacket(NetworkPacket packet) {
 if (client != null && client.isConnected()) {
 client.send(packet); // Kryonet serializa y
transmite
 }
 }

 public void broadcastFromHost(NetworkPacket packet) {
 if (isHost && server != null && server.isRunning()) {
 server.broadcast(packet); // Broadcast eficiente a
todos
 }
 }
}

```

```

Registro de Clases Personalizadas:

```

```java
// KryoClassRegistrar.java
public class KryoClassRegistrar {
 public static void registerClasses(Kryo kryo) {
 // Registrar clases personalizadas para serialización
 // eficiente
 kryo.register(Packets.PlayerJoined.class);
 kryo.register(Packets.PlayerLeft.class);
 kryo.register(Packets.GameStateUpdate.class);
 kryo.register(CatchThemAllPackets.PlayerUpdate.class);
 kryo.register(SumoPackets.PlayerKnockback.class);
 }
}
```

**Paquetes Optimizados (Ejemplo CatchThemAll):**
```java
public class CatchThemAllPackets {
 public static class PlayerUpdate extends NetworkPacket {
 public int playerId;
 public float x;
 public float y;
 public float velocityX;
 public float velocityY;
 // Solo lo necesario para sincronización
 }

 public static class DuckCaught extends NetworkPacket {
 public int duckId;
 public int playerId;
 public int pointsEarned;
 }
}
```

**Sistema de Sincronización:**
```java
@Override
public void update(float delta) {
 // Actualizar estado local
 for (IntMap.Entry<Player> entry : players) {
 entry.value.update(delta);
 }

 // Enviar solo cambios significativos
 if (shouldBroadcastUpdate()) {
 CatchThemAllPackets.PlayerUpdate update =
 new CatchThemAllPackets.PlayerUpdate();
 }
}
```

```

```
        update.playerId = localPlayerId;
        update.x = localPlayer.getX();
        update.y = localPlayer.getY();
        update.velocityX = localPlayer.getVelocityX();
        update.velocityY = localPlayer.getVelocityY();

        NetworkManager.getInstance().sendPacket(update);
    }
}
```

3. VENTAJAS

- Serialización binaria eficiente (reduce ancho de banda)
- Latencia reducida mediante compresión
- Paquetes personalizados contienen solo información necesaria
- Arquitectura cliente-servidor garantiza consistencia

B. DESAFÍO 02: GESTIÓN DE MÚLTIPLES MINIJUEGOS

1. PROBLEMÁTICA

El sistema debe soportar cuatro minijuegos completamente diferentes, cada uno con:

a) LÓGICA DISTINTA

- Ball Movement: Física de proyectiles
- Catch Them All: IA de persecución, spawn de entidades
- Sumo: Física de colisiones, detección de caída
- The Finale: Lógica especial del evento final

b) REQUISITOS DE RED DIFERENTES

- Ball Movement: Actualizaciones de posición frecuentes
- Catch Them All: Eventos de captura, posición de patos
- Sumo: Eventos de golpe, caída de jugadores
- Cada uno requiere diferentes tipos de paquetes

c) INTERFACES GRÁFICAS PERSONALIZADAS

- Diferentes elementos en pantalla
- Diferentes métricas de puntuación
- Diferentes transiciones visuales

d) ESTADOS COMPLEJOS

- Control de turnos
- Gestión de puntuaciones

- Transiciones entre minijuegos

2. SOLUCIÓN: FACTORY + STRATEGY + OBSERVER PATTERNS

Java

```

```java
public class MinigameFactory {
 public static Minigame createMinigame(MinigameType type,
int localPlayerId) {
 switch (type) {
 case BALL_MOVEMENT:
 return new BallMovementMinigame(localPlayerId);
 case CATCH_THEM_ALL:
 return new CatchThemAllMinigame(localPlayerId);
 case SUMO:
 return new SumoMinigame(localPlayerId);
 case THE_FINAL:
 return new TheFinalMinigame(localPlayerId);
 default:
 return null;
 }
 }
}

```
**Interfaz Común para todos los minijuegos:**

```java
public interface Minigame {
 void initialize();
 void update(float delta);
 void render(SpriteBatch batch, ShapeRenderer
shapeRenderer);
 void handleInput(float delta);
 boolean isFinished();
 Map<Integer, Integer> getScores();
 int getWinnerId();
 void dispose();
 void resize(int width, int height);
}
```
**GameScreen - Agnóstica del tipo de minijuego:**

```java
public class GameScreen implements Screen {
 private Minigame currentMinigame;

 @Override
 public void show() {
```

```

```

        currentMinigame      =
MinigameFactory.createMinigame(minigameType, localPlayerId);
        currentMinigame.initialize();
    }

@Override
public void render(float delta) {
    // No necesita conocer qué minijuego está activo
    currentMinigame.handleInput(delta);
    currentMinigame.update(delta);
    currentMinigame.render(batch, shapeRenderer);

    if (currentMinigame.isFinished()) {
        transitionToResultsScreen();
    }
}
...
```
Manejadores de Red Específicos por Minijuego:

```java
// Cada minijuego registra sus propios manejadores
@Override
public void initialize() {
    NetworkManager nm = NetworkManager.getInstance();

    // Registrar clases personalizadas
    nm.registerAdditionalClasses(
        CatchThemAllPackets.PlayerUpdate.class,
        CatchThemAllPackets.DuckCaught.class
    );

    // Registrar manejadores
    clientHandler = new CatchThemAllClientHandler(this);
    nm.registerClientHandler(clientHandler);

    if (nm.isHost()) {
        serverRelay = new CatchThemAllServerRelay(this);
        nm.registerServerHandler(serverRelay);
    }
}
...
```

```

### 3. VENTAJAS

- Cada minijuego es independiente y autocontenido
- Agregar nuevos minijuegos no afecta código existente

- GameScreen permanece simple y reusable
- Fácil de probar cada minijuego en aislamiento

### C. DESAFÍO 03: INTERFAZ GRÁFICA COMPLEJA

#### 1. PROBLEMÁTICA

MicroPatosMania requiere múltiples pantallas con componentes UI complejos:

##### a) MÚLTIPLES PANTALLAS

- MainMenuScreen
- CreateRoomScreen
- JoinLobbyScreen
- LobbyScreen
- GameScreen
- ResultsScreen
- ScoreboardScreen
- SpectatorScreen

##### b) COMPONENTES PERSONALIZADOS NECESARIOS

- Botones estilizados
- Campos de entrada personalizados
- Listas de jugadores
- Indicadores de puntuación
- Controles de volumen

##### c) TRANSICIONES SUAVES

- Cambios de pantalla sin "parpadeos"
- Animaciones de entrada/salida
- Efectos visuales

##### d) RESPONSIVIDAD

- Soportar diferentes resoluciones
- Controles táctiles y teclado
- Escalado adaptativo

#### 2. SOLUCIÓN: SCENE2D + COMPONENTES PERSONALIZADOS

Java

```
Uso de Scene2D para UI:

```java  

public class GameScreen implements Screen {  

    private Stage uiStage; // Scene2D Stage para UI  

    private Skin skin;     // Skin para estilizar componentes  

    private Label scoreLabel;
```

```

private Label timerLabel;
private Table incrementsContainer;

@Override
public void show() {
    uiStage = new Stage(new ScreenViewport());
    skin = UISkinProvider.getInstance().getSkin();

    // Crear tabla principal
    Table mainTable = new Table();
    mainTable.setFillParent(true);
    mainTable.top();

    // Agregar etiquetas
    scoreLabel = new Label("Score: 0", skin);
    timerLabel = new Label("Time: 10", skin);

    mainTable.add(scoreLabel).pad(10).expandX().fillX();
    mainTable.row();
    mainTable.add(timerLabel).pad(10).expandX().fillX();

    uiStage.addActor(mainTable);

    Gdx.input.setInputProcessor(uiStage);
}

@Override
public void render(float delta) {
    currentMinigame.update(delta);

    // Actualizar UI
    Map<Integer, Integer> scores =
currentMinigame.getScores();
    int currentScore = scores.getOrDefault(localPlayerId,
0);
    scoreLabel.setText("Score: " + currentScore);

    gameTimer -= delta;
    timerLabel.setText("Time: " + (int)gameTimer);

    // Renderizar UI
    uiStage.act(delta);
    uiStage.draw();
}
```

```

```

```java
// StyledButton.java
public class StyledButton extends TextButton {
    public StyledButton(String text, Skin skin) {
        super(text, skin);
        setStyle(UIStyles.getButtonStyle(skin));
    }
}

// InputField.java
public class InputField extends TextField {
    public InputField(String placeholder, Skin skin) {
        super(placeholder, skin);
        setMessageText(placeholder);
    }
}

// PlayerListItem.java
public class PlayerListItem extends Table {
    private Label playerName;
    private Label playerStatus;

    public PlayerListItem(String name, String status, Skin skin) {
        this.playerName = new Label(name, skin);
        this.playerStatus = new Label(status, skin);

        add(playerName).expandX().fillX();
        add(playerStatus).padLeft(10);
    }
}
```
```
**Sistema de Estilos UI:**```
```java
public class UIStyles {
 public static TextButton.TextButtonStyle getButtonStyle(Skin skin) {
 TextButton.TextButtonStyle style = new TextButton.TextButtonStyle();
 style.font = skin.getFont("default-font");
 style.fontColor = Color.WHITE;
 style.up = skin.getDrawable("button-up");
 style.down = skin.getDrawable("button-down");
 return style;
 }

 public static Label.LabelStyle getLabelStyle(Skin skin) {
```

```

```

        Label.LabelStyle style = new Label.LabelStyle();
        style.font = skin.getFont("default-font");
        style.fontColor = Color.WHITE;
        return style;
    }
}

```
Transiciones de Pantalla:```
java
public class ScreenTransition {
 private float duration;
 private float elapsed = 0;
 private Screen fromScreen;
 private Screen toScreen;

 public void update(float delta) {
 elapsed += delta;
 float progress = elapsed / duration;

 if (progress >= 1) {
 // Transición completada
 }
 }

 public void render(SpriteBatch batch) {
 // Renderizar transición suave
 float opacity = elapsed / duration;
 batch.setColor(1, 1, 1, opacity);
 toScreen.render(0);
 batch.setColor(1, 1, 1, 1 - opacity);
 fromScreen.render(0);
 }
}
```
```
Manejo de Eventos de UI:```
java
public class CreateRoomScreen implements Screen {
 private TextField roomNameField;
 private TextButton createButton;

 @Override
 public void show() {
 Stage stage = new Stage(new ScreenViewport());
 Skin skin = UISkinProvider.getInstance().getSkin();

 roomNameField = new TextField("", skin);
 }
}
```

```

```
        createButton = new TextButton("Crear", skin);

        createButton.addListener(new ClickListener() {
            @Override
            public void clicked(InputEvent event, float x,
float y) {
                String roomName = roomNameField.getText();
                NetworkManager.getInstance().hostGame(roomName,
8080);
                game.setScreen(new LobbyScreen(game));
            }
        });

        Table table = new Table();
        table.setFillParent(true);
        table.center();
        table.add(new Label("Nombre de Sala:", skin));
        table.row();
        table.add(roomNameField).padBottom(20);
        table.row();
        table.add(createButton);

        stage.addActor(table);
        Gdx.input.setInputProcessor(stage);
    }
}
```
...
```

### 3. VENTAJAS

- Scene2D proporciona abstracción de bajo nivel
- Componentes reutilizables y personalizables
- Sistema de eventos robusto
- Responsive al redimensionamiento de ventana
- Fácil de mantener y extender con nuevas pantallas

## IX. ESTADO ACTUAL Y PRÓXIMOS PASOS

### A. ESTADO

- Arquitectura base implementada
- Múltiples minijuegos funcionales
- Sistema de red operacional

## B. MEJORAS FUTURAS

- Optimización de rendimiento
- Agregar más minijuegos
- Mejorar interfaz gráfica
- Implementar persistencia de datos

## X. REFERENCIAS Y RECURSOS

- [1] "libGDX - Cross-platform Java Game Development," libGDX Team, [Online]. Available: <https://libgdx.com/>. [Accessed: Dec. 3, 2025].
- [2] "Kryonet - Fast Client/Server Network Communication in Java," EsotericSoftware, [Online]. Available: <https://github.com/EsotericSoftware/kryonet>. [Accessed: Dec. 3, 2025].
- [3] "Java Platform, Standard Edition 21 Documentation," Oracle Corporation, [Online]. Available: <https://docs.oracle.com/en/java/javase/21/docs/>. [Accessed: Dec. 3, 2025].
- [4] "Gradle Build Tool Documentation," Gradle Inc., [Online]. Available: <https://gradle.org/docs/current/>. [Accessed: Dec. 3, 2025].
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley Professional, 1994.
- [6] G. Booch, "Object-Oriented Analysis and Design with Applications," 3rd ed. Addison-Wesley Professional, 2007.
- [7] "Scene2D GUI Framework for libGDX," libGDX Documentation, [Online]. Available: <https://libgdx.com/wiki/graphics/2d/scene2d>. [Accessed: Dec. 3, 2025].
- [8] "Kryo - Fast and Efficient Java Serialization and Cloning," EsotericSoftware, [Online]. Available: <https://github.com/EsotericSoftware/kryo>. [Accessed: Dec. 3, 2025].
- [9] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, "Java Concurrency in Practice," Addison-Wesley Professional, 2006.
- [10] R. C. Martin, "Clean Code: A Handbook of Agile Software Craftsmanship," Prentice Hall, 2008.
- [11] "Java Network Programming," Oracle Corporation, [Online]. Available: <https://docs.oracle.com/javase/tutorial/networking/>. [Accessed: Dec. 3, 2025].

[12] "libGDX Best Practices," libGDX Team, [Online]. Available: <https://libgdx.com/wiki/start/>. [Accessed: Dec. 3, 2025].