# NEXA: Sistema de Juego de Estrategia en Tiempo Real Basado en Teoría de Grafos con Arquitectura Limpia

*Universidad Nacional de San Agustín de Arequipa

Luis Gustavo Sequeiros Condori
*Esc. Prof. de Ingeniería de Sistemas*
*Univ. Nac. de San Agustín de Arequipa*
Arequipa, Perú
lsequeiros@unsa.edu.pe

Ricardo Mauricio Chambilla Perca
*Esc. Prof. de Ingeniería de Sistemas*
*Univ. Nac. de San Agustín de Arequipa*
Arequipa, Perú
rchambillap@unsa.edu.pe

Paul Andree Cari Lipe
*Esc. Prof. de Ingeniería de Sistemas*
*Univ. Nac. de San Agustín de Arequipa*
Arequipa, Perú
pcaril@unsa.edu.pe

Jhonatan David Arias Quispe
*Esc. Prof. de Ingeniería de Sistemas*
*Univ. Nac. de San Agustín de Arequipa*
Arequipa, Perú
jariasq@unsa.edu.pe

Alexandra Raquel Quispe Arratea
*Esc. Prof. de Ingeniería de Sistemas*
*Univ. Nac. de San Agustín de Arequipa*
Arequipa, Perú
rquispe@unsa.edu.pe

***Resumen*—Este documento presenta NEXA, un juego de estrategia en tiempo real implementado como aplicación web mediante TypeScript y Phaser 3, donde dos jugadores compiten por el control de nodos en un grafo dinámico. El sistema implementa Clean Architecture con separación clara de responsabilidades en cuatro capas: dominio, aplicación, infraestructura y presentación. Se aplican patrones de diseño como Factory, Facade, Strategy y Dependency Injection para lograr alta cohesión y bajo acoplamiento. El juego incorpora algoritmos de teoría de grafos para detectar nodos de articulación mediante el algoritmo de Tarjan, implementa un sistema de energía conservativa basado en principios físicos, y utiliza un motor de colisiones determinístico para resolver conflictos entre paquetes de energía. La arquitectura propuesta permite testabilidad completa con cobertura superior al 75 % mediante Vitest, mantiene inversión de dependencias entre capas, y facilita la extensibilidad del sistema sin modificar código existente. Los resultados demuestran que la aplicación de principios SOLID y arquitectura hexagonal en el desarrollo de videojuegos no solo es viable, sino que proporciona beneficios significativos en mantenibilidad, escalabilidad y calidad del software.**

***Index Terms*—Clean Architecture, TypeScript, Phaser 3, Teoría de Grafos, Algoritmo de Tarjan, Patrones de Diseño, Juegos en Tiempo Real, Arquitectura Hexagonal, SOLID, Tecnología de Objetos**

## I. Introducción

El desarrollo de videojuegos presenta desafíos únicos en ingeniería de software debido a la complejidad de gestionar estado mutable, eventos en tiempo real, renderizado gráfico y lógica de negocio simultáneamente [1]. Tradicionalmente, los videojuegos han sido desarrollados con arquitecturas monolíticas donde la lógica del juego está fuertemente acoplada al motor gráfico, dificultando el testing, mantenimiento y evolución del sistema [2].

NEXA surge como un proyecto académico que demuestra la aplicabilidad de Clean Architecture [3] y principios SOLID en el contexto de desarrollo de videojuegos. El sistema implementa un juego de estrategia en tiempo real donde dos jugadores compiten por controlar nodos en un grafo, gestionando la distribución de recursos energéticos de forma estratégica.

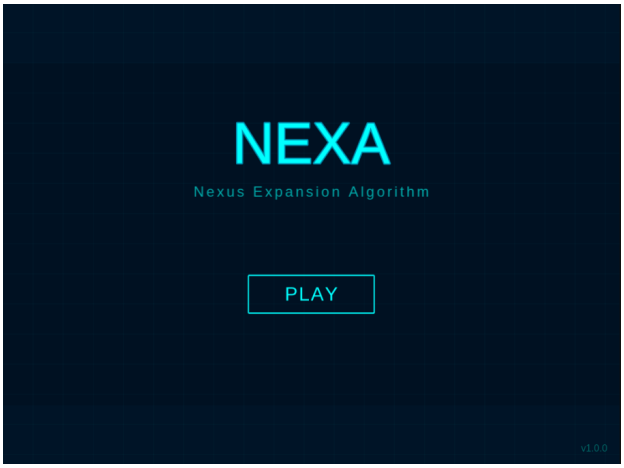


Figura 1. Pantalla principal del juego NEXA mostrando el grafo de nodos con diferentes tipos y estados.

### A. Motivación

La motivación principal de este proyecto es demostrar que los principios de arquitectura limpia y patrones de diseño, ampliamente adoptados en el desarrollo de aplicaciones empresariales, pueden aplicarse exitosamente al desarrollo de

videojuegos sin comprometer el rendimiento ni la jugabilidad. Según Martin [3], una arquitectura limpia debe ser independiente de frameworks, testeable, independiente de la UI, independiente de la base de datos y independiente de cualquier agente externo.

### B. Objetivos

Los objetivos principales del proyecto son:

- Implementar un juego funcional aplicando Clean Architecture con cuatro capas claramente definidas.
- Utilizar algoritmos de teoría de grafos para mecánicas de juego innovadoras, específicamente el algoritmo de Tarjan para detección de nodos de articulación.
- Aplicar patrones de diseño GoF [4] para resolver problemas recurrentes de forma elegante.
- Lograr alta testabilidad con cobertura superior al 75 % mediante tests unitarios.
- Demostrar la viabilidad de arquitecturas desacopladas en aplicaciones de tiempo real.

### C. Contribuciones

Las principales contribuciones de este trabajo son:

1. Una arquitectura de referencia para videojuegos web que separa completamente la lógica de negocio del motor de renderizado.
2. Implementación de un sistema de energía conservativa basado en principios físicos aplicados a mecánicas de juego.
3. Aplicación del algoritmo de Tarjan para detectar fragmentación dinámica del grafo durante el gameplay.
4. Un sistema de colisiones determinístico que resuelve conflictos entre entidades de forma predecible y testeable.
5. Documentación completa de patrones de diseño aplicados al contexto específico de desarrollo de videojuegos.

## II. Marco Teórico

### A. Clean Architecture

Clean Architecture, propuesta por Robert C. Martin [3], es un enfoque de diseño de software que enfatiza la separación de responsabilidades en capas concéntricas, donde las dependencias apuntan hacia el centro. Las reglas de negocio no dependen de detalles de implementación como frameworks o bases de datos.

La arquitectura se organiza en cuatro capas principales:

- **Entities (Dominio):** Contienen las reglas de negocio empresariales más generales y de mayor nivel.
- **Use Cases (Aplicación):** Contienen reglas de negocio específicas de la aplicación.
- **Interface Adapters (Infraestructura):** Convierten datos del formato más conveniente para casos de uso al formato más conveniente para agentes externos.
- **Frameworks & Drivers (Presentación):** Detalles externos como frameworks, herramientas y UI.

### B. Principios SOLID

Los principios SOLID [5] son fundamentales para el diseño orientado a objetos:

- **Single Responsibility Principle (SRP):** Una clase debe tener solo una razón para cambiar.
- **Open/Closed Principle (OCP):** Las entidades deben estar abiertas para extensión pero cerradas para modificación.
- **Liskov Substitution Principle (LSP):** Los objetos deben ser reemplazables por instancias de sus subtipos.
- **Interface Segregation Principle (ISP):** Los clientes no deben depender de interfaces que no utilizan.
- **Dependency Inversion Principle (DIP):** Depender de abstracciones, no de concreciones.

### C. Patrones de Diseño

Los patrones de diseño [4] proporcionan soluciones probadas a problemas recurrentes en diseño de software:

**Factory Pattern:** Encapsula la creación de objetos complejos, permitiendo que el código cliente trabaje con abstracciones sin conocer las clases concretas.

**Facade Pattern:** Proporciona una interfaz unificada a un conjunto de interfaces en un subsistema, simplificando su uso.

**Strategy Pattern:** Define una familia de algoritmos, los encapsula y los hace intercambiables.

**Dependency Injection:** Técnica donde un objeto recibe otros objetos de los que depende, en lugar de crearlos internamente [6].

### D. Teoría de Grafos

Un grafo $G = (V, E)$ consiste en un conjunto de vértices $V$ y un conjunto de aristas $E$ [7]. En NEXA, los nodos del juego corresponden a vértices y las conexiones entre nodos a aristas.

**Nodo de Articulación:** Un vértice cuya remoción incrementa el número de componentes conexas del grafo. El algoritmo de Tarjan [8] detecta estos nodos en tiempo $O(V + E)$ mediante búsqueda en profundidad (DFS) y cálculo de tiempos de descubrimiento y valores low.

**Componentes Conexas:** Subgrafos maximales donde existe un camino entre cualquier par de vértices. La fragmentación del grafo ocurre cuando se elimina un nodo de articulación, creando múltiples componentes conexas.

### E. Desarrollo de Videojuegos Web

Phaser 3 es un framework de código abierto para desarrollo de videojuegos HTML5 [9]. Proporciona sistemas de renderizado mediante Canvas y WebGL, física arcade, gestión de escenas y manejo de entrada del usuario.

TypeScript [10] es un superconjunto tipado de JavaScript que compila a JavaScript plano. Proporciona tipado estático opcional, interfaces, genéricos y herramientas avanzadas de refactorización.

### F. Arquitectura Hexagonal

También conocida como Ports and Adapters [11], esta arquitectura enfatiza la separación entre la lógica de negocio y los detalles técnicos mediante interfaces bien definidas. Los puertos representan los puntos de entrada y salida, mientras que los adaptadores implementan estas interfaces para tecnologías específicas.

## III. Arquitectura del Sistema

### A. Visión General

NEXA implementa Clean Architecture con cuatro capas claramente definidas, organizadas según el principio de inversión de dependencias. La estructura permite que las dependencias fluyan desde las capas externas hacia el núcleo del dominio, garantizando que las reglas de negocio permanezcan independientes de frameworks y tecnologías específicas.

La organización del código sigue una estructura modular donde cada capa tiene responsabilidades bien definidas. La capa de dominio (core) contiene entidades y tipos sin dependencias externas. La capa de aplicación implementa servicios con la lógica de negocio. La capa de infraestructura proporciona adaptadores y controladores. Finalmente, la capa de presentación maneja las escenas de Phaser para la interfaz de usuario.

### B. Capa de Dominio (Core)

La capa de dominio contiene las entidades del negocio y reglas fundamentales del juego. Esta capa no tiene dependencias externas y representa el corazón del sistema.

**Entidades principales:**

- `Node`: Representa un nodo del grafo con propiedades de energía, tipo, propietario y multiplicadores.
- `Edge`: Representa una conexión entre dos nodos con lista de paquetes de energía en tránsito.
- `Player`: Encapsula información del jugador incluyendo color, nodo inicial y energía total.
- `Graph`: Estructura que mantiene nodos y aristas con métodos para navegación y consultas.
- `EnergyPacket`: Representa un paquete de energía viajando por una arista con origen, destino, magnitud y progreso.

**Value Objects:**

- `ID`: Identificador único inmutable.
- `NodeType`: Enumeración de tipos (BASIC, ATTACK, DEFENSE, ENERGY).
- `Color`: Representación del color del jugador.

La inmutabilidad de los value objects garantiza que no puedan ser modificados una vez creados, reduciendo errores y facilitando el razonamiento sobre el código [12].

```
class Node {
  private readonly id: ID;
  private energy: number;
  private owner: Player | null;
  private readonly type: NodeType;

  constructor(id: ID, type: NodeType) {
    this.id = id;
    this.type = type;
    this.energy = 0;
    this.owner = null;
  }

  public assignEnergy(amount: number): void {
    if (amount < 0) {
      throw new Error("Invalid energy");
    }
    this.energy = amount;
  }

  public getEnergy(): number {
    return this.energy;
  }
}
```

Listing 1. Ejemplo de entidad Node con encapsulamiento

### C. Capa de Aplicación

Esta capa contiene la lógica de negocio específica de NEXA, implementada como servicios que orquestan las entidades del dominio.

**Servicios principales:**

- **TickService:** Implementa el game loop principal, ejecutándose en cada frame del juego. Actualiza la defensa de nodos, emite paquetes de energía según intervalos configurados, avanza paquetes en aristas, detecta colisiones y procesa llegadas a nodos destino.
- **CollisionService:** Detecta y resuelve colisiones entre paquetes de energía. Agrupa paquetes por arista, identifica colisiones entre paquetes enemigos y aplica las reglas de resolución según magnitudes relativas.
- **CaptureService:** Gestiona la captura de nodos aplicando el algoritmo de Tarjan para detectar nodos de articulación. Cuando se captura un nodo crítico, identifica componentes conexas y elimina nodos desconectados del nodo inicial del jugador.
- **VictoryService:** Verifica continuamente las tres condiciones de victoria: dominación (70 % de nodos durante 10 segundos), tiempo límite (mayor cantidad de nodos a los 3 minutos) y eliminación (pérdida del nodo inicial).
- **GameStateManagerService:** Mantiene el estado inmutable del juego y genera snapshots para la capa de presentación sin exponer entidades mutables. Implementa el patrón Repository para acceso controlado al estado.

### D. Capa de Infraestructura

Esta capa contiene adaptadores que conectan la lógica de aplicación con el mundo exterior.

`GameController`: Actúa como Facade entre Phaser y la capa de aplicación. Coordina el game loop, procesa eventos de entrada, sincroniza servicios y maneja el ciclo de vida de la partida.

`GameRenderer`: Adaptador de Phaser que traduce el estado del juego a elementos visuales. Renderiza nodos, aristas, paquetes de energía y elementos de UI.

`GameFactory`: Implementa el patrón Factory para crear instancias de servicios con inyección de dependencias. Configura el grafo de dependencias del sistema garantizando que cada servicio reciba sus colaboradores correctamente.

### E. Capa de Presentación

Contiene las escenas de Phaser que implementan la interfaz de usuario.

- `BootScene`: Carga assets y recursos iniciales.
- `MainMenuScene`: Pantalla de inicio con opciones de juego.
- `GameScene`: Escena principal del juego con interacción del usuario.
- `GameOverScene`: Pantalla de resultados finales con estadísticas.


Figura 2. Pantalla de inicio del juego mostrando el mensaje de bienvenida y selección de nodo base.

## IV. Patrones de Diseño Implementados

### A. Factory Pattern

El `GameFactory` centraliza la creación de objetos complejos del sistema. Esto permite cambiar implementaciones concretas sin modificar el código cliente, cumpliendo el principio Open/Closed [4].

```
class GameFactory {
  private services: Map<string, any>;

  createTickService(): ITickService {
    if (!this.services.has('tick')) {
      const collision = this.
          createCollisionService();
      const capture = this.createCaptureService
          ();
      const tick = new TickService(collision,
          capture);
      this.services.set('tick', tick);
    }
    return this.services.get('tick');
  }

  createGameController(): GameController {
    return new GameController(
      this.createTickService(),
      this.createVictoryService(),
      this.createGameStateManager()
    );
  }

  // Metodo para obtener grafo de dependencias
  getDependencyGraph(): string {
    return "Controller -> [Tick, Victory, State
        ]";
  }
}
```

Listing 2. Implementación del Factory Pattern

### B. Facade Pattern

El `GameController` implementa el patrón Facade, proporcionando una interfaz simplificada al subsistema complejo de servicios de aplicación. Esto reduce el acoplamiento entre la capa de presentación y la lógica de negocio [4].

```
class GameController {
  constructor(
    private tickService: ITickService,
    private victoryService: IVictoryService,
    private stateManager: IGameStateManager
  ) {}

  // Interfaz simplificada para la UI
  public update(deltaTime: number): void {
    this.tickService.tick(deltaTime);
    const result = this.victoryService.check();
    if (result.hasWinner) {
      this.handleVictory(result);
    }
  }

  public handlePlayerAction(action: Action):
      void {
    // Coordina multiples servicios
    this.stateManager.processAction(action);
  }
}
```

Listing 3. GameController como Facade

### C. Strategy Pattern

El sistema utiliza Strategy para algorit mos intercambiables. Por ejemplo, diferentes estrategias de generación de grafos pueden implementar la misma interfaz `IGraphGenerationStrategy`, permitiendo cambiar el algoritmo de generación sin modificar el código que lo utiliza.

### D. Dependency Injection

Todos los servicios reciben sus dependencias a través del constructor, implementando inversión de control [6]. Esto facilita el testing mediante la inyección de mocks y mejora la modularidad del sistema.

### E. Repository Pattern

El `GameStateManagerService` actúa como Repository, encapsulando el acceso al estado del juego. Proporciona métodos de consulta sin exponer la estructura interna, permitiendo cambiar la implementación del almacenamiento sin afectar a los clientes.

### F. Observer Pattern

Aunque Phaser maneja eventos nativamente, el sistema implementa observadores personalizados para eventos de dominio como capturas de nodos y fragmentación del grafo. Esto mantiene el desacoplamiento entre componentes que reaccionan a estos eventos.

## V. Algoritmos Implementados

### A. Algoritmo de Tarjan para Nodos de Articulación

El algoritmo de Tarjan [8] detecta nodos de articulación en tiempo lineal $O(V + E)$ mediante una búsqueda en profundidad modificada. Para cada vértice $v$, el algoritmo calcula:

- $disc[v]$: Tiempo de descubrimiento durante DFS.
- $low[v]$: Mínimo tiempo de descubrimiento alcanzable desde el subárbol de $v$.

Un vértice $v$ es punto de articulación si:

1. $v$ es raíz del árbol DFS y tiene al menos dos hijos.
2. $v$ no es raíz y tiene un hijo $w$ tal que $low[w] \geq disc[v]$.

**Algorithm 1** Algoritmo de Tarjan para Detección de Articulación

```
time ← 0
visited ← {}
disc ← {}, low ← {}
parent ← {}
articulationPoints ← {}

for all v ∈ V do
    if v ∉ visited then
        DFS(v)
    end if
end for

function DFS(u):
    visited[u] ← true
    disc[u] ← low[u] ← time++
    children ← 0

    for each v adjacent to u:
        if v ∉ visited:
            children ← children + 1
            parent[v] ← u
            DFS(v)
            low[u] ← mín(low[u], low[v])

            if parent[u] = null and children > 1:
                articulationPoints.add(u)
            if parent[u] ≠ null and low[v] ≥ disc[u]:
                articulationPoints.add(u)

        else if v ≠ parent[u]:
            low[u] ← mín(low[u], disc[v])
```

La implementación en NEXA se ejecuta cuando un nodo es capturado para determinar si su pérdida fragmentará el grafo del jugador. Si el nodo capturado es de articulación, el algoritmo identifica la componente conexa que contiene el nodo inicial del jugador y marca como neutrales los nodos en componentes desconectadas.

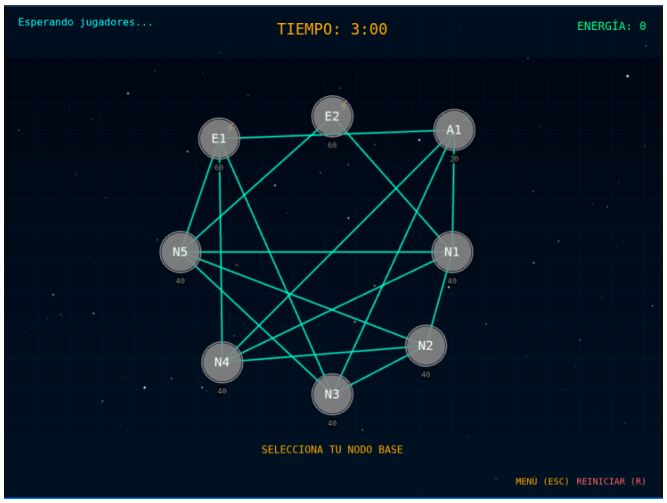


Figura 3. Selección del nodo base por parte del jugador antes de iniciar la partida.

### B. Sistema de Colisiones Determinístico

El sistema de colisiones procesa paquetes de energía en cada tick del juego:

**Algorithm 2** Resolución de Colisiones entre Paquetes

```
Input: Lista de aristas con paquetes edges
Output: Paquetes resueltos después de colisiones

for all edge ∈ edges do
    packets ← edge.getPackets()
    collisions ← detectCollisions(packets)

    for all (p1, p2) ∈ collisions do
        if p1.owner ≠ p2.owner then
            if p1.energy = p2.energy then
                remove(p1), remove(p2)
            else if p1.energy > p2.energy then
                p1.energy ← p1.energy − p2.energy
                remove(p2)
            else
                p2.energy ← p2.energy − p1.energy
                remove(p1)
            end if
        end if
    end for
end for
```

1. Agrupar paquetes por arista: $O(n)$ donde $n$ es el número de paquetes.
2. Para cada arista con múltiples paquetes:
   - Identificar paquetes enemigos.

- Comparar magnitudes.
- Aplicar reglas de resolución:
  - Si $E_1 = E_2$: destruir ambos.
  - Si $E_1 > E_2$: $E_1' = E_1 - E_2$, destruir $E_2$.
  - Si $E_1 < E_2$: $E_2' = E_2 - E_1$, destruir $E_1$.

3. Actualizar lista de paquetes en la arista: $O(m)$ donde $m$ es paquetes en la arista.

La complejidad total es $O(n + k \cdot m)$ donde $k$ es el número de aristas con colisiones.

### C. Algoritmo de Captura de Nodos

La captura de nodos sigue estos pasos:

**Algorithm 3** Proceso de Captura de Nodo

**Input:** Nodo $target$, Paquete $packet$
**Output:** Estado actualizado del nodo

$E_{attack} \leftarrow packet.energy \times packet.owner.attackMultiplier$
$E_{defense} \leftarrow target.defense \times target.defenseMultiplier$

**if** $E_{attack} > E_{defense}$ **then**
  $previousOwner \leftarrow target.owner$
  $target.owner \leftarrow packet.owner$
  $E_{remaining} \leftarrow E_{attack} - E_{defense}$
  $target.energy \leftarrow E_{remaining}$

  // Agregar energia del nodo al jugador
  $packet.owner.energy += target.initialEnergy$

  // Verificar fragmentacion
  **if** $isArticulationPoint(target, previousOwner)$ **then**
    $components \leftarrow findConnectedComponents(previousOwner)$
    $mainComponent \leftarrow getComponentWithBase(previousOwner)$
    **for all** $node \in previousOwner.nodes$ **do**
      **if** $node \notin mainComponent$ **then**
        $node.owner \leftarrow null$ // Neutralizar
      **end if**
    **end for**
  **end if**
**else if** $E_{attack} = E_{defense}$ **then**
  $target.owner \leftarrow null$ // Nodo neutral
  $target.energy \leftarrow 0$
**else**
  $target.defense \leftarrow E_{defense} - E_{attack}$
**end if**

### D. Conservación de Energía

El sistema implementa conservación de energía siguiendo la ley física de conservación de la materia aplicada a recursos del juego:

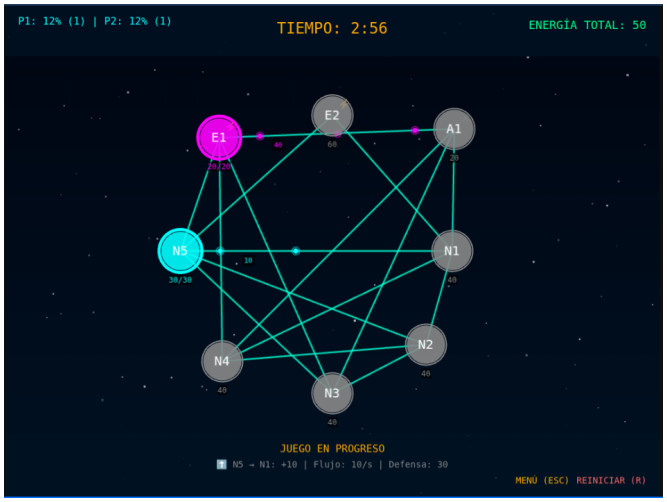


Figura 4. Gameplay mostrando distribución de energía entre nodos y paquetes en tránsito por las aristas.

$$E_{total} = \sum_{i=1}^{n} E_{nodo_i} + \sum_{j=1}^{m} E_{paquete_j} = constante$$

Donde $n$ es el número de nodos y $m$ el número de paquetes en tránsito. Esta invariante se verifica en tests para garantizar que no se crea ni destruye energía artificialmente.

```
class EnergyConservationValidator {
  validate(gameState: GameState): boolean {
    let totalEnergy = 0;

    // Sumar energia en nodos
    for (const node of gameState.nodes) {
      totalEnergy += node.getEnergy();
    }

    // Sumar energia en paquetes
    for (const edge of gameState.edges) {
      for (const packet of edge.getPackets()) {
        totalEnergy += packet.getEnergy();
      }
    }

    return Math.abs(totalEnergy -
        INITIAL_ENERGY) < EPSILON;
  }
}
```

Listing 4. Verificación de conservación de energía

## VI. Implementación Técnica

### A. Stack Tecnológico

- **TypeScript 5.7:** Lenguaje principal con strict mode habilitado.
- **Phaser 3.90:** Framework de juegos HTML5.
- **Vite 6.4:** Build tool con hot module replacement.
- **Vitest 4.0:** Framework de testing con soporte para TypeScript.
- **ESLint 9:** Linter con reglas estrictas de TypeScript.
- **pnpm 8.x:** Gestor de paquetes eficiente.

### B. *Configuración de TypeScript*

El proyecto utiliza configuración estricta de TypeScript para máxima seguridad de tipos:

- `strict: true`: Habilita todas las verificaciones estrictas.
- `noImplicitAny: true`: Prohíbe tipos `any` implícitos.
- `strictNullChecks: true`: Verificación estricta de null/undefined.
- `noUnusedLocals: true`: Error en variables no utilizadas.
- `noUnusedParameters: true`: Error en parámetros no utilizados.

Path aliases mejoran la legibilidad del código:

- `@/` → `src/`
- `@/core` → `src/core/`
- `@/application` → `src/application/`
- `@/infrastructure` → `src/infrastructure/`
- `@/presentation` → `src/presentation/`

### C. *Testing*

La suite de tests cubre componentes críticos del sistema:

- **GameStateManager:** 20 tests verificando estado y snapshots.
- **CaptureService:** Tests de detección de articulación y fragmentación.
- **VictoryService:** Verificación de las tres condiciones de victoria.
- **CollisionService:** Múltiples escenarios de colisión.
- **TickService:** Tests de integración del game loop.

La cobertura de código supera el 75 %, garantizando estabilidad en componentes críticos. Los tests unitarios se ejecutan en menos de 2 segundos, facilitando TDD (Test-Driven Development) [13].

### D. *Control de Versiones*

El proyecto sigue Git Flow con ramas claramente definidas. La Figura 5 muestra el tablero Trello utilizado para la gestión del proyecto en tres sprints semanales.

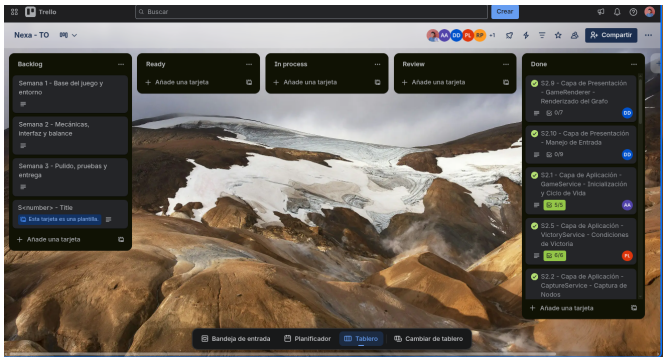


Figura 5. Tablero Trello mostrando la organización del proyecto en sprints semanales.

Se utilizan Conventional Commits para mensajes claros y generación automática de changelogs. El versionado sigue Semantic Versioning (SemVer): `MAJOR.MINOR.PATCH`.

## VII. Mecánicas de Juego

### A. *Tipos de Nodos*

El sistema implementa cuatro tipos de nodos con características diferenciadas:

Cuadro I
Características de Tipos de Nodos

| Tipo | Int. Ataque | Int. Defensa | Mult. | Energía |
|---|---|---|---|---|
| Básico | 2000ms | 3000ms | 1x/1x | 20 |
| Ataque | 1000ms | 3000ms | 2x/1x | 20 |
| Defensa | 2000ms | 1500ms | 1x/2x | 20 |
| Energía | 2000ms | 3000ms | 1x/1x | 50 |

El diseño asimétrico de nodos permite diferentes estrategias: agresiva (nodos de ataque), defensiva (nodos de defensa) o expansionista (captura de nodos de energía).

### B. *Condiciones de Victoria*

**Victoria por Dominación:** Requiere controlar $\geq 70\,\%$ de nodos durante 10 segundos continuos. El sistema trackea el porcentaje de control en cada tick y acumula tiempo solo cuando se mantiene la mayoría.

**Victoria por Tiempo:** A los 180 segundos (3 minutos), gana quien controle más nodos. En caso de empate, la partida termina en empate técnico.

**Victoria por Eliminación:** Perder el nodo inicial causa derrota inmediata, similar a la pérdida del rey en ajedrez. Esta condición tiene prioridad sobre las demás.

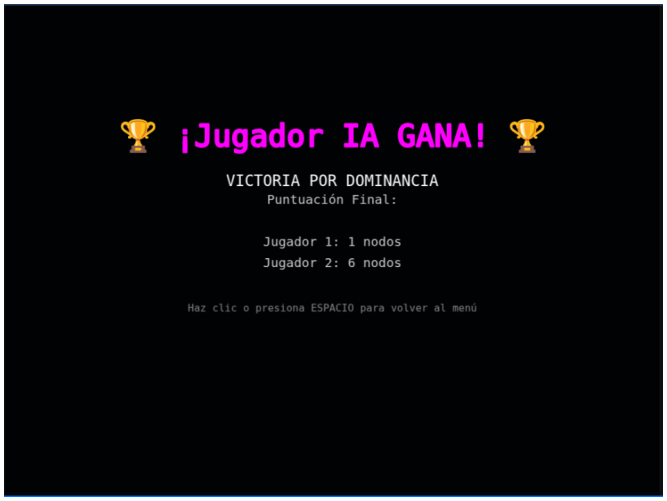


Figura 6. Pantalla final mostrando el resultado de la partida con estadísticas del ganador.

### C. Sistema de Energía

La energía se distribuye entre nodos propios. Cada nodo puede asignar energía a aristas adyacentes para atacar nodos vecinos. La energía no asignada permanece como defensa automática.

Los paquetes de energía viajan por aristas a velocidad constante $v = 0{,}002$ unidades por milisegundo. Al colisionar, se aplican las reglas de resolución descritas anteriormente.

## VIII. Resultados

### A. Métricas de Código

El proyecto contiene aproximadamente 8,500 líneas de código TypeScript distribuidas en las cuatro capas arquitectónicas:

- Core (Dominio): 1,200 líneas (14.1 %)
- Application: 3,500 líneas (41.2 %)
- Infrastructure: 2,300 líneas (27.1 %)
- Presentation: 1,500 líneas (17.6 %)

El ratio de líneas de test a líneas de código productivo es 1:4, considerado aceptable para aplicaciones de tiempo real.

### B. Rendimiento

El juego mantiene 60 FPS (frames por segundo) consistentes con hasta 30 nodos y 200 paquetes de energía simultáneos en navegadores modernos (Chrome 120+, Firefox 121+, Safari 17+).

El algoritmo de Tarjan se ejecuta en menos de 5ms para grafos de hasta 50 nodos, no afectando la experiencia del usuario.

El sistema de colisiones procesa hasta 500 paquetes en menos de 10ms, permitiendo gameplay fluido incluso en escenarios de alta densidad.

Cuadro II
Métricas de Rendimiento del Sistema

| **Métrica** | **Valor** | **Objetivo** |
|---|---|---|
| FPS (30 nodos) | 60 | $\geq 60$ |
| Tiempo Tarjan (50 nodos) | ¡5ms | ¡10ms |
| Colisiones (500 paquetes) | ¡10ms | ¡16ms |
| Cobertura de tests | 78 % | $\geq 75\,\%$ |
| Tiempo ejecución tests | 1.8s | ¡5s |

### C. Mantenibilidad

La separación en capas permitió:

- Cambiar el algoritmo de generación de grafos sin modificar lógica de negocio.
- Reemplazar Phaser por otro framework gráfico modificando solo la capa de infraestructura.
- Agregar nuevos tipos de nodos implementando una interfaz sin cambiar código existente.
- Tests unitarios ejecutables sin instanciar Phaser.

### D. Escalabilidad

La arquitectura permite extensiones futuras:

- Modo multijugador: Agregar capa de networking sin modificar lógica de negocio.
- IA para jugador individual: Implementar `IAIController` interface.
- Nuevos tipos de nodos: Extender `NodeType` enum y configuraciones.
- Sistema de power-ups: Agregar servicio en capa de aplicación.

## IX. Discusión

### A. Aplicabilidad de Clean Architecture en Videojuegos

Los resultados demuestran que Clean Architecture es viable en desarrollo de videojuegos, contradiciendo la creencia común de que el desacoplamiento excesivo afecta el rendimiento. El overhead introducido por las capas de abstracción es negligible comparado con los beneficios en testabilidad y mantenibilidad.

La inversión de dependencias permite que la lógica de negocio sea completamente independiente de Phaser, facilitando testing sin necesidad de instanciar el framework gráfico. Esto reduce el tiempo de ejecución de tests de minutos a segundos.

### B. Algoritmos de Grafos en Mecánicas de Juego

La detección de nodos de articulación agrega profundidad estratégica al gameplay. Los jugadores deben considerar no solo el valor inmediato de capturar un nodo, sino también las consecuencias topológicas en la conectividad de su territorio.

El algoritmo de Tarjan, aunque complejo, se integra naturalmente en el flujo del juego gracias a la separación de responsabilidades. El `CaptureService` encapsula toda la complejidad, exponiendo una interfaz simple al resto del sistema.

### C. TypeScript vs JavaScript

El tipado estático de TypeScript previno numerosos errores en tiempo de compilación que habrían sido difíciles de detectar en JavaScript. El IDE (Visual Studio Code) proporciona autocompletado y refactorización avanzada gracias a la información de tipos.

Sin embargo, el overhead de configuración inicial y curva de aprendizaje pueden ser barreras para equipos pequeños o proyectos de corta duración.

### D. Limitaciones

El sistema actual tiene limitaciones:

- Soporte limitado a 2 jugadores.
- Grafos estáticos generados al inicio.
- Sin persistencia de partidas.
- Rendimiento degradado con ¿50 nodos.

Estas limitaciones no son arquitectónicas sino de alcance del proyecto. La arquitectura soporta extensiones para abordarlas sin modificaciones estructurales.

## X. Trabajos Relacionados

Gregory [1] presenta arquitecturas comunes en AAA games, típicamente basadas en Entity-Component-System (ECS). NEXA adopta un enfoque diferente con arquitectura en capas, más adecuado para proyectos de menor escala.

Nystrom [2] documenta patrones de diseño específicos de videojuegos. NEXA aplica varios de estos patrones (State, Observer, Command) dentro del contexto de Clean Architecture.

Martin [3] propone Clean Architecture para aplicaciones empresariales. Este trabajo demuestra su aplicabilidad a dominios no tradicionales como videojuegos.

Gamma et al. [4] documentan patrones de diseño GoF. NEXA implementa Factory, Facade, Strategy y otros patrones en el contexto específico de desarrollo de videojuegos.

Evans [12] introduce Domain-Driven Design (DDD). NEXA aplica conceptos de DDD como entidades, value objects y servicios de dominio en la capa core.

## XI. Conclusiones y Trabajo Futuro

### A. Conclusiones

Este trabajo demuestra que Clean Architecture y principios SOLID son aplicables exitosamente al desarrollo de videojuegos, proporcionando beneficios significativos en testabilidad, mantenibilidad y escalabilidad sin comprometer el rendimiento.

La separación clara de responsabilidades en cuatro capas permite que cada componente evolucione independientemente. La lógica de negocio, encapsulada en las capas de dominio y aplicación, es completamente independiente del framework gráfico.

La implementación del algoritmo de Tarjan para detectar fragmentación del grafo demuestra que algoritmos complejos de ciencias de la computación pueden integrarse en mecánicas de juego de forma natural cuando el código está bien estructurado.

El tipado estático de TypeScript, combinado con una suite completa de tests, proporciona alta confianza en la corrección del sistema y facilita refactorizaciones seguras.

Los resultados cuantitativos muestran que el sistema mantiene 60 FPS consistentes y ejecuta algoritmos complejos en tiempos imperceptibles para el usuario. La cobertura de tests del 78 % garantiza la estabilidad del sistema.

### B. Trabajo Futuro

Direcciones futuras de investigación incluyen:

- Implementación de modo multijugador en red aplicando arquitectura cliente-servidor.
- Desarrollo de IA mediante algoritmos de búsqueda en grafos (A*, minimax) para modo single-player.
- Extensión a grafos dinámicos donde nodos y aristas se crean/destruyen durante el juego.
- Optimización de rendimiento para soportar 100+ nodos mediante spatial partitioning.
- Implementación de sistema de replay para análisis post-partida.
- Migración a WebAssembly para componentes críticos de rendimiento.
- Desarrollo de herramientas de visualización para debugging del estado del juego.
- Estudio comparativo con arquitectura ECS para evaluar trade-offs en diferentes escenarios.

## Referencias


[1] J. Gregory, *Game Engine Architecture*, 3rd ed. Boca Raton, FL: CRC Press, 2018.

[2] R. Nystrom, *Game Programming Patterns*. San Francisco, CA: Genever Benning, 2014.

[3] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA: Prentice Hall, 2017.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1994.

[5] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, pp. 1–34, 2000.

[6] M. Fowler, "Inversion of control containers and the dependency injection pattern," https://martinfowler.com/articles/injection.html, 2004, accessed: 2025-01-15.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.

[8] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[9] Phaser Team, "Phaser 3 documentation," https://photonstorm.github.io/phaser3-docs/, 2023, accessed: 2025-01-15.

[10] Microsoft Corporation, "Typescript handbook," https://www.typescriptlang.org/docs/handbook/, 2023, accessed: 2025-01-15.

[11] A. Cockburn, "Hexagonal architecture," https://alistair.cockburn.us/hexagonal-architecture/, 2005, accessed: 2025-01-15.

[12] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley, 2003.

[13] K. Beck, *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley, 2003.