Proyecto: Crazy Risk
Materia: Algoritmos y Estructuras de Datos I (CE-1103)
Instituto: Tecnológico de Costa Rica
Escuela: Ingeniería en Computadores
Semestre: II Semestre 2025
Integrantes: Ian Alejandro Bonilla Mena, Brandon Sebastián Meza Chaves
Fecha de entrega: 03/10/2025

Tabla de Contenidos

---

1. Introducción

Crazy Risk es una versión digital del clásico juego de mesa Risk, diseñada para **dos jugadores** en un entorno cliente-servidor usando **C#**. El objetivo del juego es conquistar todos los territorios del mapa mediante estrategias de refuerzos, ataques y planeación. La implementación incluye estructuras de datos propias, algoritmos

para resolución de combates, mecánicas de refuerzos y tarjetas, y comunicación entre jugadores mediante **sockets TCP/IP**.

El desarrollo de este proyecto busca reforzar los conocimientos de algoritmos, estructuras de datos y arquitectura en capas, integrando además una interfaz gráfica desarrollada en **Unity**.

---

2. Breve descripción del problema

El reto consiste en recrear digitalmente Crazy Risk, considerando:

- Distribución equitativa de territorios y tropas iniciales.
- Turnos con fases de **refuerzos, ataques y planeación**.
- Resolución de combates mediante **tiradas de dados aleatorias**.
- Intercambio de tarjetas por refuerzos adicionales.
- Comunicación en tiempo real entre dos computadoras.
- Restricción: **no usar colecciones nativas de .NET**, solo implementaciones propias de listas, colas y pilas.

---

3. Descripción de la solución

3.1 Capa de comunicación

- **Tecnología:** Sockets TCP/IP (Cliente-Servidor).
- El **servidor** mantiene el estado del juego y gestiona los turnos.
- El **cliente** se conecta y recibe actualizaciones del juego en tiempo real.
- Los mensajes están en **JSON** para mantener coherencia y facilitar la actualización de territorios y tropas.

Ejemplo de estructura JSON:

```
{
 "territorioId": 5,
 "jugadorPropietario": "Jugador1",
 "tropas": 4,
 "color": "#FF0000"
```

```
}
```

- Bloqueo de turno: mientras un jugador juega, el otro no puede modificar el estado.

3.2 Capa de lógica del juego

- **Clases principales:** GameManager, Jugador, Territorio, Tarjeta.
- **Funciones clave:**
    - SiguienteTurno()
    - CalcularRefuerzos()
    - ColocarTropas()
    - ResolverAtaque(atacante, defensor, tropasAtacante, tropasDefensor)
- **Mecánica de refuerzos:**

refuerzos = totalTerritorios / 3 + bonusContinente + contadorGlobalTarjetas;

- **Intercambio de tarjetas:**
    - Se sigue la serie de Fibonacci (2, 3, 5, 8, …) para calcular refuerzos adicionales.
    - Un jugador puede acumular hasta 5 tarjetas; al obtener la 6ta debe intercambiar al menos un trío.
- **Ataques:**
    - Atacante lanza 1–3 dados, defensor 1–2.
    - Comparación de dados: mayor contra mayor, segundo mayor contra segundo mayor.
    - Tropas eliminadas según resultado.
- **Planeación:**
    - Permite mover tropas entre territorios propios adyacentes.
    - Se mantiene al menos 1 tropa en el territorio de origen.

3.3 Capa de presentación

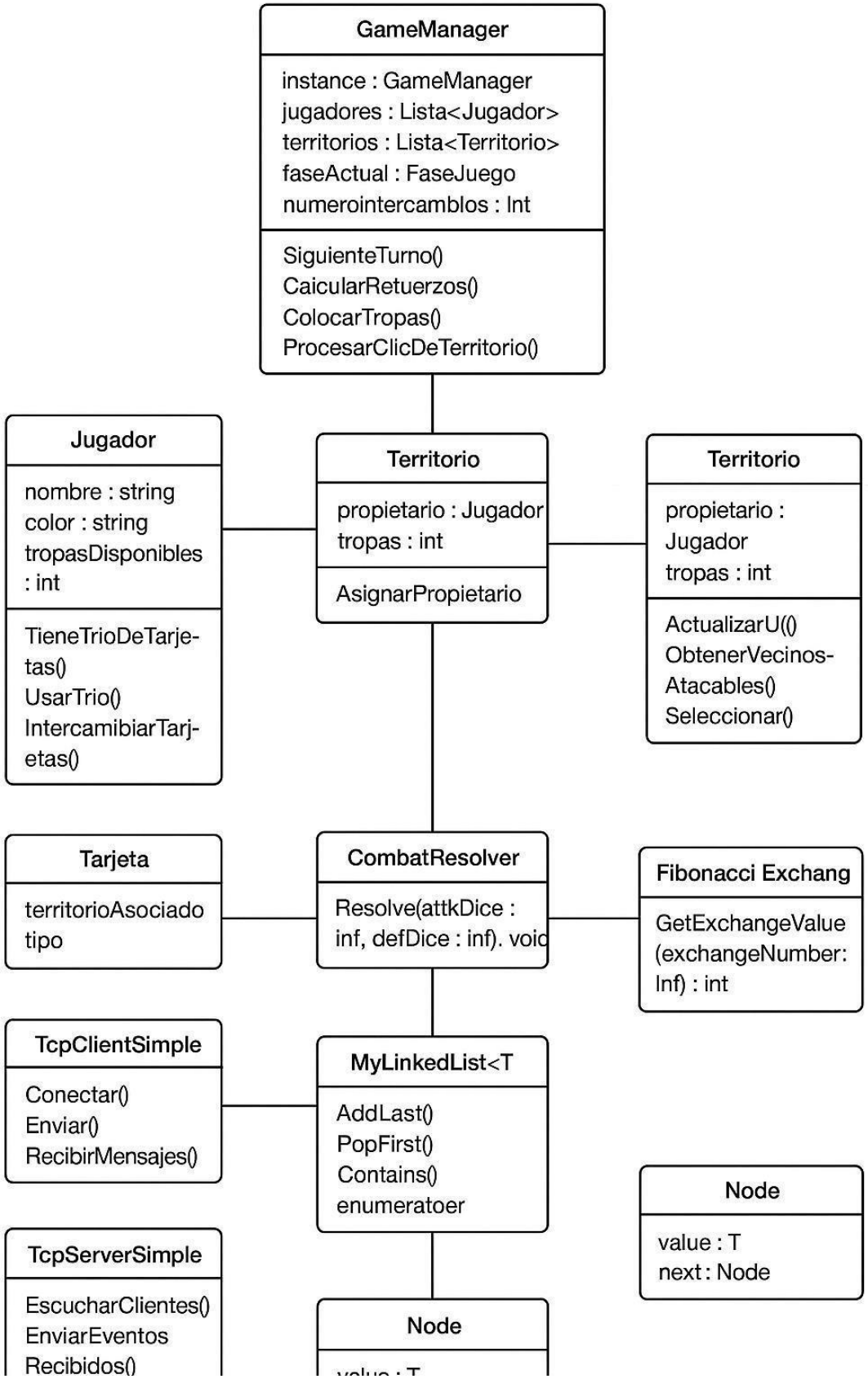
- **Interfaz gráfica** en Unity
- Funcionalidades:
    - Crear y unirse a partidas.
    - Selección de territorios y cantidad de tropas.
    - Visualización en tiempo real de territorios, colores y tropas.
    - Botones para ataques, intercambio de tarjetas y movimientos.

La interfaz busca ser intuitiva, mostrando los territorios coloreados según su propietario y actualizando las tropas de forma dinámica tras cada acción.

---

4. Diseño general

4.1 Diagrama de clases UML


GameManager
instance : GameManager
jugadores : Lista<Jugador>
territorios : Lista<Territorio>
faseActual : FaseJuego
numerointercamblos : Int
SiguienteTurno()
CaicularRetuerzos()
ColocarTropas()
ProcesarClicDeTerritorio()
Jugador
nombre : string
color : string
tropasDisponibles : int
TieneTrioDeTarje-tas()
UsarTrio()
IntercamibiarTarj-etas()
Territorio
propietario : Jugador
tropas : int
AsignarPropietario
Territorio
propietario : Jugador
tropas : int
ActualizarU(()
ObtenerVecinos-Atacables()
Seleccionar()
Tarjeta
territorioAsociado
tipo
CombatResolver
Resolve(attkDice : inf, defDice : inf). voic
Fibonacci Exchang
GetExchangeValue (exchangeNumber: Inf) : int
TcpClientSimple
Conectar()
Enviar()
RecibirMensajes()
MyLinkedList<T
AddLast()
PopFirst()
Contains()
enumeratoer
Node
value : T
next : Node
TcpServerSimple
EscucharClientes()
EnviarEventos Recibidos()
Node

4.2 Explicación del diagrama

- GameManager es un **singleton** que coordina el flujo del juego.
- Jugador controla territorios y posee tarjetas.
- Territorio conoce sus adyacencias y número de tropas.
- Tarjeta permite intercambiar por refuerzos cuando se cumple un trío.

---

5. Uso de Git y GitHub

Durante el desarrollo del proyecto se utilizó Git como sistema de control de versiones y GitHub como repositorio remoto.

El flujo de trabajo siguió una estructura tipo GitFlow:

Rama main: contiene la versión estable del proyecto.
Rama develop: usada para la integración de nuevas funcionalidades.
Ramas feature: para el desarrollo de módulos específicos como lógica de juego, red o interfaz.

Commits frecuentes y descriptivos, indicando cada cambio realizado.

Pull requests y merge requests revisados entre los integrantes.

Esto permitió mantener trazabilidad, revertir errores y trabajar colaborativamente.

---

6. Implementación y aspectos técnicos

- **Lenguaje:** C#
- **IDE recomendado:** Visual Studio
- **Estructuras de datos:** listas, colas y pilas propias, no se usan colecciones nativas.
- **Comunicación:** Sockets TCP/IP para mantener el estado sincronizado entre jugadores.
- **Interfaz gráfica:** para interacción con mapa y turnos.
- **Algoritmos:** resolución de ataques, cálculo de refuerzos, intercambio de tarjetas y movimiento de tropas.

---

7. Limitaciones y problemas encontrados

Complejidad adicional por implementar estructuras propias en lugar de usar .NET.

La implementación manual de estructuras de datos aumentó la complejidad del proyecto.

La sincronización de turnos entre cliente y servidor requirió mecanismos de bloqueo estrictos.

Sincronización de turnos crítico: requiere bloqueo de turnos en cliente-servidor.

Tests de red deben realizarse en LAN o IP pública.

La **interfaz en Unity** presentó retos en la sincronización visual con los datos recibidos del servidor.

---

8. Conclusiones

- Se logró recrear Crazy Risk digital para dos jugadores, cumpliendo la mayoría de los requerimientos.

- La arquitectura en capas permite escalabilidad para añadir un jugador neutro o un espectador.
- El uso de UML y patrones de diseño ayudó a estructurar el proyecto de manera profesional.

---

9. Prompts de IA utilizados

Durante el desarrollo se utilizaron herramientas de inteligencia artificial (ChatGPT - GPT-5) como apoyo para:

- Redacción de documentación técnica y académica.
- Generación de ejemplos de código base en C#.
- Explicación de conceptos sobre sockets TCP/IP y estructuras de datos.
- Revisión y mejora del texto final.

https://github.com/IanBM18/Proyecto1-Datos1.git