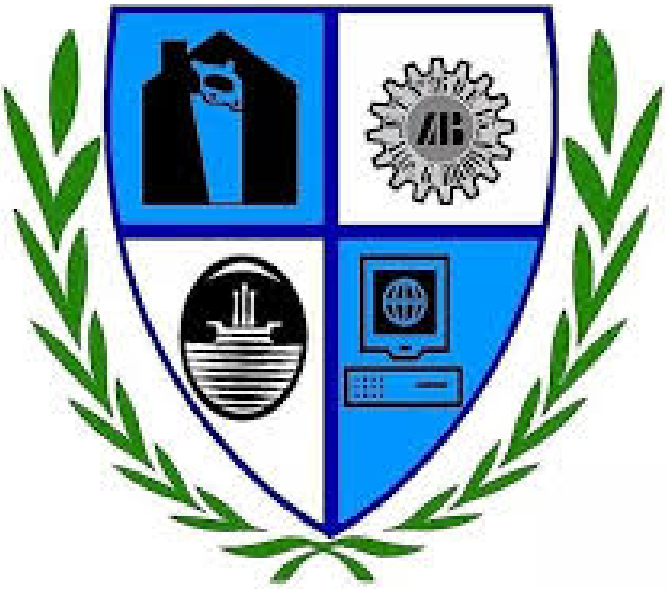
Programación Sobre Redes



Concurrencia

Matias Sebastian Barqui,
Geremias Nicolas Romero Montaner,
Bautista Mateo Vadalá Miranda

Profesor: Uzal, Alan ***Curso: 6to 3ra Mañana***
Fecha de entrega; 14-07

Mitzvah Mayhem: Juego de cartas

Debatimos la idea del proyecto, llegamos a la conclusión de hacer una simulación de un juego de cartas tipo Magic.
Para cumplir las condiciones del proyecto usamos SQLite3 como una biblioteca para almacenar la base de datos de manera local, no usamos interfaz gráfica pero decoramos la terminal para que sea más atractiva a la vista. Nos ayudamos de múltiples librerías que serán detalladas más adelante.

Concurrencia

Para la programación se tuvieron en cuenta conceptos clave de concurrencia que se aplicaron de la siguiente manera:

1. Locks: Para proteger el acceso concurrente a los mazos.
2. Queues: Permiten comunicación segura entre hilos y procesos.
3. Events: Sincronizan los turnos de los jugadores.
4. Timeouts: Evitan que el sistema se bloquee de forma inesperada.
5. Procesos: Aíslan la lógica entre procesos.
6. Hilos: Permiten 5 jugadores.

```
1. GameEngine.iniciar_partida()
   ├─> Crea multiprocessing.Queue()                    # Comunicación entre procesos
   ├─> Crea ProcesoArbitro(jugadores_data, resultado_queue)
   ├─> proceso_arbitro.start()                         # Inicia proceso árbitro
   ├─> resultado_dict = resultado_queue.get()          # Espera resultado del árbitro
   └─> proceso_arbitro.join()                          # Espera fin proceso árbitro

2. ProcesoArbitro.run()
   ├─> Crea queue.Queue()                              # Comunicación entre hilos
   ├─> Crea threading.Lock()                           # Lock compartido para hilos
   ├─> Crea [threading.Event() for _ in range(5)]      # Evento por cada jugador
   ├─> Crea 5 JugadorThread(nombre, mazo, turno_event, resultado_queue, lock, id)
   ├─> jugador_thread.start()                          # Inicia hilos jugadores
   └─> Llama a _ejecutar_partida_5_jugadores(jugadores_threads, resultado_queue, turno_events)
```

```
3. JugadorThread.run()
   ├─> turno_event.wait()                              # Espera su turno (bloquea)
   ├─> with lock_compartido:                           # Sección crítica protegida
   |       └─> mazo.tomar_carta()
   ├─> resultado_queue.put((nombre, carta, id))        # Comunica jugada al árbitro
   └─> turno_event.clear()                             # Libera el turno

4. ProcesoArbitro._ejecutar_partida_5_jugadores()
   ├─> turno_events[i].set()                           # Activa turno jugador i
   ├─> resultado_queue.get(timeout=10.0)               # Espera jugada del hilo
   ├─> jugador_thread.terminar()                        # Señal para terminar hilo
   └─> jugador_thread.join()                           # Espera a que termine hilo
```

```
5. GameEngine (proceso principal)
   ├─> resultado_queue.get()                           # Recibe resultado final partida
   └─> proceso_arbitro.join()                          # Espera fin proceso árbitro
```

```
6. ProcesoReiniciarMazos.run()
   └─> engine.reiniciar_mazos(nombres_jugadores)       # Reinicia mazos en proceso separado

7. ProcesoLimpiarEstadisticas.run()
   └─> repo.limpiar_estadisticas()                     # Limpia estadísticas en proceso separado
```

Ejemplo:

B. Ejecución de Hilos

python

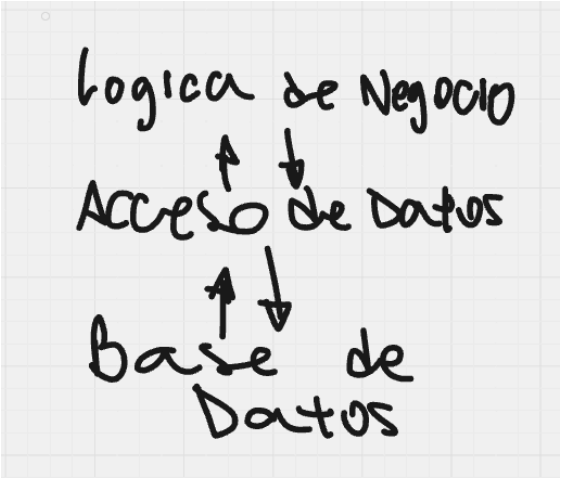
```python
# LÍNEA 30-62 en game_engine.py
def run(self):
    """Ejecuta el hilo del jugador"""
    print(f"🎮 {self.nombre} (Hilo {self.jugador_id}) se ha unido a la partida")

    while self.activo and not self.mazo.esta_vacio():
        # Esperar mi turno
        self.turno_event.wait()  # SINCRONIZACIÓN CON EVENT

        # Simular tiempo de pensamiento
        time.sleep(tiempo_pensamiento)  # PAUSA EN EL HILO

        # Jugar carta
        carta = self.jugar_carta()
        if carta:
            self.resultado_queue.put((self.nombre, carta, self.jugador_id))  # ENVÍA DATOS
```

Librerías

Utilizamos sqlite3 para la base de datos, threading y multiprocessing para los hilos y procesos, typing para indicar explícitamente los tipos esperados para variables, argumentos de funciones y valores de retorno, random para la aleatoridad, queue para comunicar los procesos e hilos, time para la simulación de pensamiento, contextlib para los with, y dataclasses para los datos y métodos.

Librería	Archivos	Propósito Principal
sqlite3	2	Base de datos
threading	1	Hilos de jugadores
multiprocessing	1	Procesos
typing	5	Anotaciones de tipo
random	2	Aleatoriedad
queue	1	Comunicación
time	1	Tiempo y pausas
contextlib	2	Context managers
dataclasses	1	Modelos de datos

Arquitectura



Arquitectura de 3 Capas (Database, Data Access, Business Logic)
Sistema de Torneo con 5 jugadores
Concurrencia con 3 procesos y 5 hilos
Patrón Repository para acceso a datos
Patrón Singleton para conexión a BD
Manejo de errores
Sistema de estadísticas completo
Mazos dinámicos y balanceados

División de tareas

Programación: Bautista Vadalá , Geremias Nicolas Romero Montaner, Matias Barqui
Diagramas: Matias Barqui, Geremias Nicolas Romero Montaner
Investigación e idea: Bautista Vadalá
Documentación: Bautista Vadalá
Manejo de github (versiones, fix, etc): Bautista Vadalá
Recordatorio de conceptos: Geremias Nicolas Romero Montaner y Matias Barqui

Errores

Usamos manejo de errores, y con errores específicos recurrimos a IAs, foros, y compañeros de curso para pedir ayuda tanto como lógica, como conceptual.

POO

Implementamos clases en distintos archivos que cada una tiene sus métodos y atributos, para sumar conceptos de Programación Orientada a Objetos.

Base de datos

Usamos muchos de los métodos integrados con la librería para trabajar con la base de datos.

Control de concurrencia

Finalización y espera

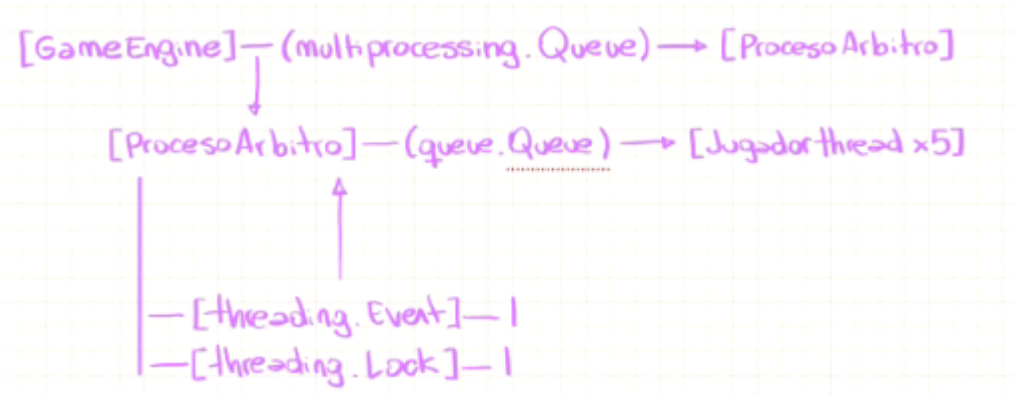
El proceso árbitro espera a que todos los hilos terminen usando join().
El proceso principal espera a que el proceso árbitro termine usando join().

¿Por qué es seguro?

No hay dos hilos modificando el mismo mazo a la vez (por el lock).
No hay dos procesos accediendo a la vez a la misma cola (las colas de multiprocessing y threading son seguras).
Cada jugador solo juega cuando su evento está activo (no hay solapamiento de turnos).

¿Qué problemas evita?

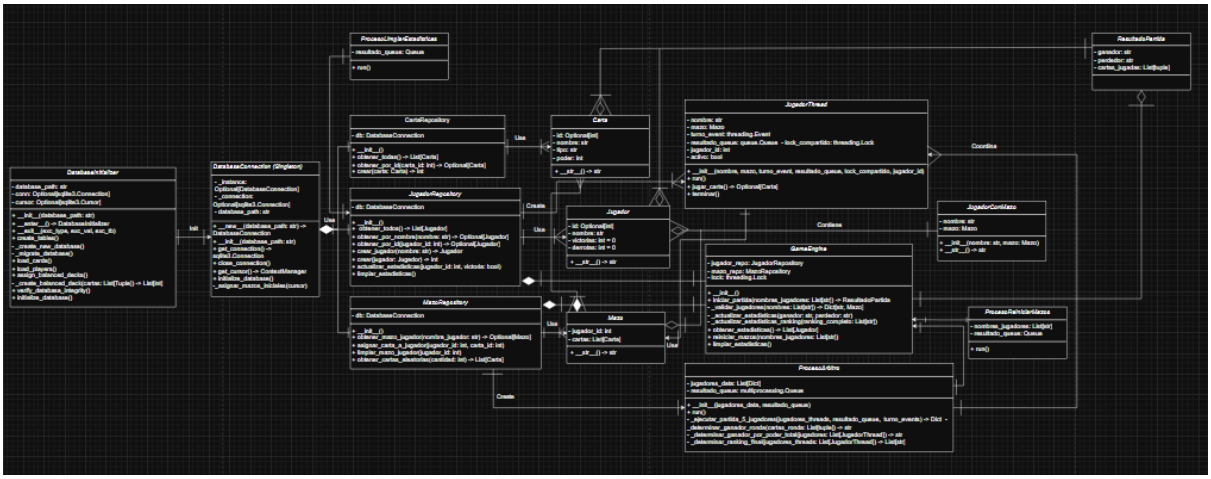
Condiciones de carrera (race conditions)
Interbloqueos (deadlocks), ya que el uso de locks y eventos está bien delimitado.
Corrupción de datos en recursos compartidos (mazos, colas).
Boceto inicial (sin código):

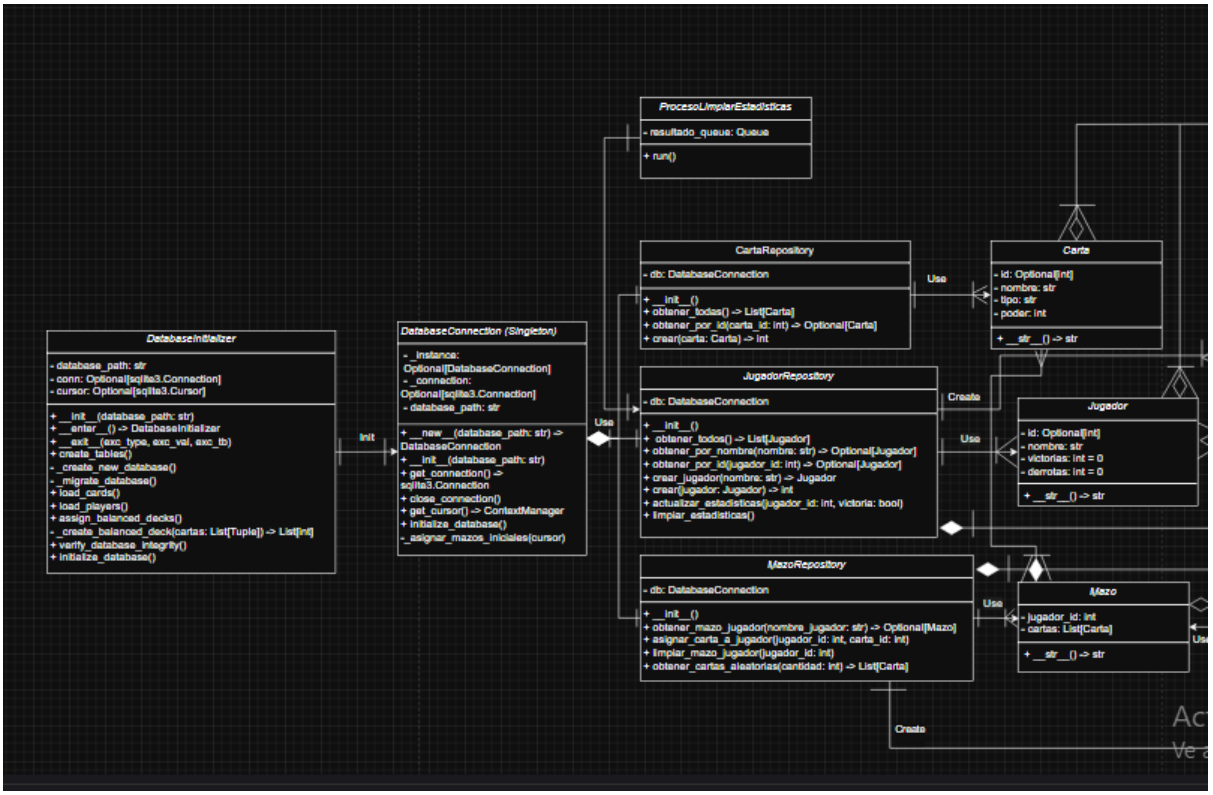


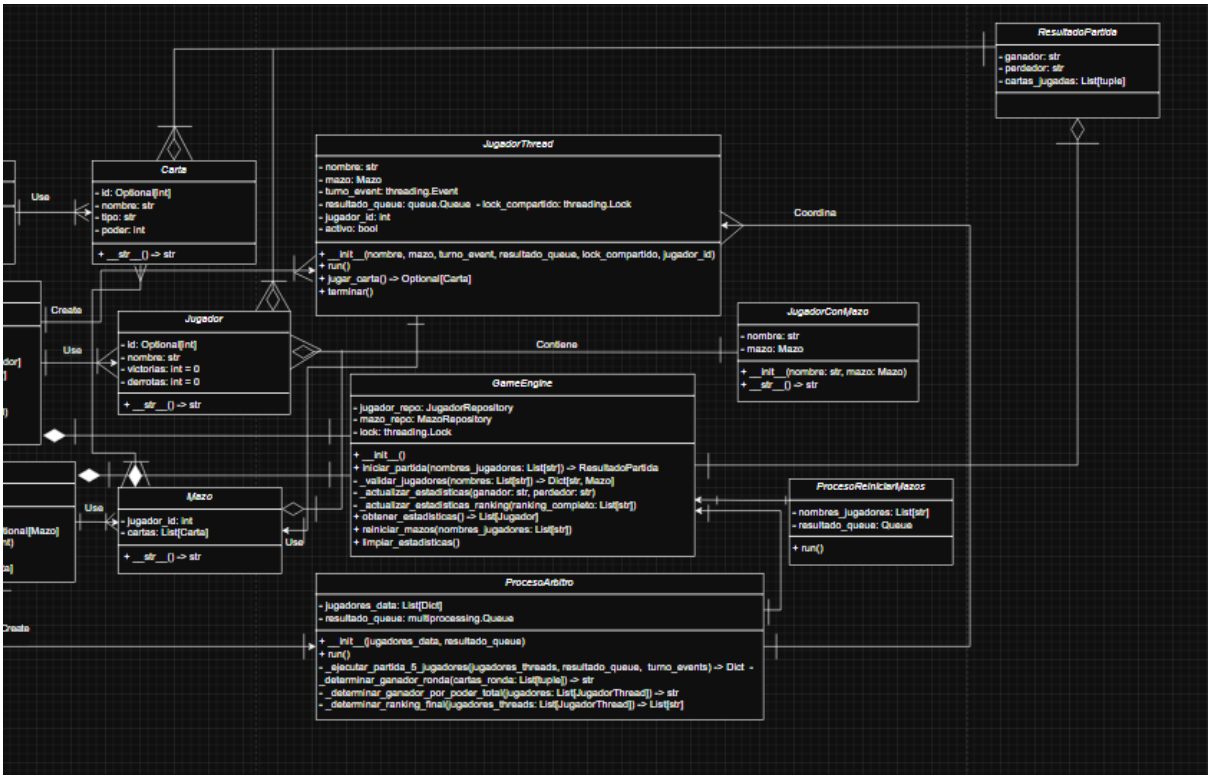
Diagramas

Diagrama de Gantt:

Diagrama de Clases y relaciones:







https://viewer.diagrams.net/?tags=%7B%7D&lightbox=1&highlight=0000ff&edit=_blank&layers=1&nav=1&dark=auto#G1Pwkpbxnoqeu-06we_9s5oXW5pYgz6J0k

Repositorio y correspondiente documentación

https://github.com/vadalabau/Mitzvah_Mayhem.git