Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2233 Programación Avanzada (2022-2)

Tarea 3

Entrega

- **Tarea**
  - **Fecha y hora:** lunes 14 de noviembre de 2022, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`
- `README.md`
  - **Fecha y hora:** lunes 14 de noviembre de 2022, 22:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`

Objetivos

- Tomar decisiones de diseño y modelación en base a un documento de requisitos.
- Aplicar conocimientos de creación de redes para comunicación efectiva entre computadores.
- Diseñar e implementar una arquitectura cliente-servidor.
- Aplicar conocimientos de manejo de *bytes* para seguir un protocolo de comunicación preestablecido.

Índice

1. *DCCard-Jitsu*

¡Hurra! ¡Hurra! Exclama *Crazy Cruz*, a modo de celebración por lograr detener la invasión zombie. Quedaste muy cansado, pero con ganas de fiesta, así que *Crazy Cruz* y tú organizan una por el triunfo reciente, antes de que vuelvas a tu dimensión como todo un héroe. Días después, al momento de regresar, te percatas que recibiste un *DCCorreo* de parte de *Crazy Cruz* (~~¿Mail interdimensional?~~), que contiene unos boletos de hospedaje en el recientemente inaugurado resort ubicado en el *DCCostanera Center*, con un mensaje adjunto: *"Te mereces unas vacaciones* ***guiño***".

Luego de unos días descansando en el resort, escuchas una voz desesperada que se acerca a ti, ¡Es *Crazy Cruz* que viene corriendo! Después de recuperar el aire te explica seriamente que el *multiverso*, tal y como lo conocemos, dejará de existir. *Hernan4444* sigue haciendo de las suyas, y ahora tomó control del misterioso *Dojo* ubicado en el *monte Fuji*, que se dice que permite controlar todos los universos existentes.

Luego de mucha conversación, deciden ejecutar el plan de tomar el control del *Dojo* y derrotar a *Hernan4444* de una vez por todas, pero no será fácil, ya que para llegar a él deberán sortear a los pingüinos guardianes del Dojo, que dominan el arte de guerra conocido como *DCCard-Jitsu* y que ahora lo apoyan en sus planes malvados ~~para aportar a la trama~~. Para esto, junto a profesores, ayudantes y todos los amigos que has hecho en este viaje, deciden crear una simulación para aprender este tipo de combate. Deberás utilizar todas tus habilidades aprendidas haciendo énfasis en *serialización* y *networking* para obtener toda la ayuda posible y así controlar el *Dojo* y salvar el *multiverso*.

Figura 1: Logo de *DCCard-Jitsu*

2. Flujo del programa

DCCard-Jitsu es un juego multijugador en donde cada jugador deberá competir contra el oponente en un mortal duelo de ~~cachipún~~ cartas, emulando el arte de guerra de los guerreros pingüinos del monte Fuji.

Para cumplir con el correcto flujo del juego, el programa debe contar con un **servidor** funcional que permita la transmisión de datos entre jugadores, por lo que siempre se debe comenzar ejecutando este servidor, seguido de los clientes.

Cada jugador debe corresponder a una instancia de la clase **cliente**, la cual será la encargada de iniciar la interfaz gráfica y será el único medio de interacción con el servidor del programa.

2.1. Inicio

Al correr el programa, el jugador debe poder visualizar la Ventana de inicio, donde deberá ingresar su nombre con el cual va a participar. El formato del nombre debe cumplir las siguientes características:

- Largo mayor o igual a 1, y menor o igual a 10 caracteres.
- Debe ser alfanumérico.
- No puede repetirse entre los usuarios existentes en el programa. Para verificar esto, el programa es *case-insensitive*, es decir, no hace distinción entre letras mayúsculas y minúsculas.

Luego, podrás hacer click en el botón de confirmación y **el servidor se encargará** de verificar que el formato del nombre ingresado sea válido. Si el formato es incorrecto, se debe notificar al jugador mediante un mensaje de texto por medio de la interfaz. En caso de cumplir con el formato solicitado, la Ventana de inicio se cerrará y se abrirá la Ventana de espera (solo si esta no se encuentra llena, de lo contrario se debe notificar al usuario y mantenerse en la Ventana de inicio).

2.2. Preparación

Los jugadores ahora se encontrarán en la Ventana de espera, donde se visualizará el nombre de usuario de cada jugador. El primer jugador en llegar a esta sala deberá esperar la llegada de un segundo jugador para poder comenzar el juego. Cuando llegue el segundo jugador, comenzará una cuenta regresiva de `CUENTA_REGRESIVA_INICIO` segundos. Mientras esta cuenta no haya terminado, cualquiera de los dos jugadores podría volver a la Ventana de inicio o salirse del programa. En este caso, para el jugador que sigue en la ventana de espera, la cuenta regresiva se detendrá, y en el momento en que hayan dos jugadores nuevamente esperando, la cuenta regresiva se reiniciará y comenzará a contar de nuevo. Esto impide iniciar una nueva partida hasta que se conecte otro jugador.

Si el temporizador llega a cero, la partida comienza automáticamente.

Como se mencionó antes, en el caso de que un jugador quisiera entrar a la Ventana de espera, pero esta se encuentre llena o ya haya iniciado el juego, se le avisará en la misma Ventana de inicio el evento correspondiente, y no se podrá entrar a la Ventana de espera.

2.3. Juego

Al comenzar la partida, se deberá barajar el mazo, con `15` cartas, de cada jugador y se tomarán las primeras `BARAJA_PANTALLA`[1], que se desplegarán en la pantalla. El jugador debe elegir solo una, dentro de un tiempo equivalente a `CUENTA_REGRESIVA_RONDA` segundos. Cuando ambos jugadores hayan seleccionado y confirmado la carta a jugar, **con el botón**, o la `CUENTA_REGRESIVA_RONDA` sea igual a 0, se jugarán ambas cartas y se determinará el ganador de la ronda. Si la `CUENTA_REGRESIVA_RONDA` es igual a 0 y el jugador aún no ha seleccionado y confirmado una carta, se jugará la carta seleccionada o una al azar en caso contrario. Las reglas para definir al ganador de cada ronda y finalmente del juego se explica con más detalle en Reglas de *DCCard-Jitsu*.

En la Ventana de juego, se deberán mostrar ambos nombres de usuarios, la ronda actual, las cartas ganadoras del jugador[2], las `BARAJA_PANTALLA` cartas disponibles para utilizar en la ronda y el tiempo de ronda de `CUENTA_REGRESIVA_RONDA` segundos, además de un botón para confirmar la carta seleccionada. El juego estará dado por un duelo en tiempo real, en el cual ambos jugadores deben seleccionar y confirmar una carta. Al registrarse ambas confirmaciones, o al agotarse la `CUENTA_REGRESIVA_RONDA`, se deberán mostrar ambas cartas y determinar un ganador. El funcionamiento detallado de cada ronda y de cada elemento que conforman el juego se encuentra en la sección de Reglas de *DCCard-Jitsu*. Cuando se gana una ronda, el jugador guarda la información de la carta con la que ganó, y una vez que algún jugador **junte tres cartas de distinto color, de igual o diferente elemento, se declarará ganador** y el juego finalizará, por lo que se deberá desplegar la Ventana Final para cada uno de los clientes, mostrando un mensaje de si se ganó o perdió el duelo, junto a un botón que permita volver a la Ventana de inicio.

3. Networking

Para poder ganar la partida de *DCCard-Jitsu*, tendrás que usar todos tus conocimientos de *networking*. Deberás desarrollar una arquitectura `cliente - servidor` con el modelo **TCP/IP** haciendo uso del

[1] `BARAJA_PANTALLA` corresponde a un subconjunto del mazo
[2] Representadas por un color y elemento

módulo **socket**.

Tu misión será implementar dos programas separados, uno para el **servidor** y otro para el **cliente**. De los mencionados, **siempre** se deberá ejecutar primero el servidor y este quedará escuchando para que se puedan conectar uno o más clientes. Debes tener en consideración que la comunicación es siempre entre cliente y servidor, **nunca directamente entre clientes**.

3.1. Arquitectura `cliente` - `servidor`

Las siguientes consideraciones **deben ser cumplidas al pie de la letra**. Lo que no esté especificado aquí puedes implementarlo según tu criterio, siempre y cuando cumpla con lo solicitado y no contradiga nada de lo indicado (puedes **preguntar** si algo no está especificado o si no queda completamente claro).

3.1.1. Separación funcional

El cliente y el servidor deben estar separados, esto implica que deben estar en directorios diferentes, uno llamado `cliente` y otro llamado `servidor`. Cada directorio debe contar con los archivos y módulos necesarios para su correcta ejecución, asociados a los recursos que les correspondan, además de un archivo principal `main.py`, el cual inicializa cada una de estas entidades funcionales. La estructura que debes seguir se indica en el siguiente diagrama:

```
T3
├── cliente
│   ├── main.py
│   ├── parametros.json
│   ├── cripto.py
│   ├── sprites
│   └── ...
├── servidor
│   ├── main.py
│   ├── parametros.json
│   ├── cripto.py
│   ├── cartas.py
│   └── ...
├── .gitignore
├── README.md
├── otro_archivo.dcc
└── ...
```

Si bien, las carpetas asociadas al `cliente` y al `servidor` se ubican en el mismo directorio (`T3`), la ejecución del `cliente` **no debe depender de archivos en la carpeta del `servidor`**, y la ejecución del `servidor` **no debe depender de archivos y/o recursos en la carpeta del `cliente`**. Esto significa que debes tratarlos como si estuvieran ejecutándose en computadores diferentes. La figura 2 muestra una representación esperada para los distintos componentes del programa:

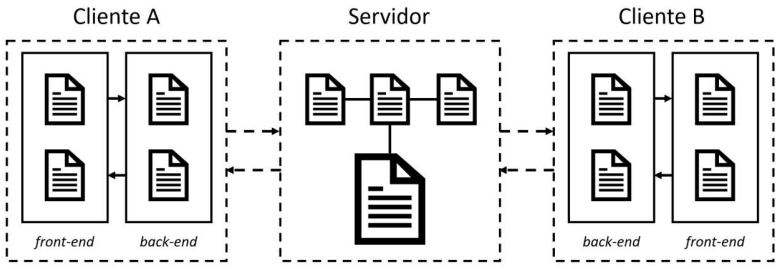

Figura 2: Separación cliente-servidor y *front-end-back-end*.

Cabe destacar que **solo el cliente tendrá una interfaz gráfica**. Por lo tanto, todo cliente debe contar con una separación entre *back-end* y *front-end*, mientras que la comunicación entre el `cliente` y `servidor` debe realizarse mediante *sockets*. Ten en cuenta que la separación deberá ser de caracter **funcional**. Esto quiere decir que **toda tarea** encargada de **validar y verificar acciones** deberá ser **elaborada en el servidor**, mientras que el cliente deberá solamente recibir esta información y actualizar la interfaz.

3.1.2. Conexión

El `servidor` contará con un archivo de formato `JSON`, ubicado en la carpeta del `servidor`. Este archivo debe contener los datos necesarios para instanciar un *socket*. El archivo debe llevar el siguiente formato:

```
{
    "host": <direccion_ip>,
    "port": <puerto>,
    ...
}
```

Por otra parte, el `cliente` deberá conectarse al *socket* abierto por el `servidor` haciendo uso de los datos encontrados en el archivo `JSON` de la carpeta del `cliente`. Es importante recalcar que el `cliente` y el `servidor` **no deben usar el mismo archivo `JSON`** para obtener estos parámetros.

3.1.3. Método de codificación

Cuando se establece la conexión entre el `cliente` y el `servidor`, deberán encargarse de intercambiar información constantemente entre sí. Por ejemplo, el `cliente` le comunica al `servidor` la carta que es lanzada y este le responderá con un mensaje indicando los efectos que tuvo esta carta en el juego. ¡Pero cuidado! Como no queremos que un contrincante pueda *hackear* nuestro mensaje y conocer nuestra baraja, debemos asegurarnos de encriptar el contenido antes de enviarlo, de tal forma que si alguien lo intercepta no pueda descifrarlo. El método de encriptación se explicará en detalle más adelante. **Luego** de encriptar el contenido, deberás codificar los mensajes enviados entre el `cliente` y el `servidor` según la siguiente estructura:

- Los primeros 4 *bytes* del mensaje deben indicar **largo del contenido del mensaje ya encriptado** que se enviarán. Estos 4 bytes deben estar en formato *big endian*.[3]
- A continuación, debes enviar el contenido encriptado del mensaje. Este debe separarse en bloques de 32 *bytes*. Cada bloque debe estar precedido de 4 *bytes* que indiquen el número del bloque, comenzando a contar desde uno y codificados en *little endian*. Si para el último bloque no alcanza

[3]El *endianness* es el orden en el que se guardan los bytes en un respectivo espacio de memoria. Esto es relevante porque cuando intentes usar los métodos `int.from_bytes` e `int.to_bytes` deberás proporcionar el *endianness* que quieras usar, además de la cantidad de bytes que quieres usar para representarlo. Para más información puedes revisar este **enlace**.

el mensaje para completar los 32 *bytes*, deberás rellenar los espacios restantes del bloque con *bytes* ceros (b'\x00'):

| 4 bytes | **4bytes** | bloque1 | **4bytes** | bloque2 | **4bytes** | bloque3 | ... |

Cuadro 1: Método de codificación

3.1.4. Método de encriptación

Finalmente, el **método de encriptación** que deberás realizar para esconder el contenido de tus mensajes y evitar *hackeos* es el siguiente:

- El mensaje deberá ser separado en **tres partes**. La primera parte es construida con todos los bytes que se encuentran a tres posiciones de distancia partiendo desde el byte con índice *cero*. Para la segunda parte se realiza lo mismo pero comenzado desde el índice *uno*, y para la tercera comenzando desde índice *dos*.

Llamemos a X la secuencia de bytes que se quieren encriptar, y a A , B y C a las 3 partes por construir. Ahora se utilizará el algoritmo descrito quedando las tres partes de la siguiente manera:

$$X = b_0\ b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8\ b_9\ b_{10}$$
$$A = b_0\ b_3\ b_6\ b_9$$
$$B = b_1\ b_4\ b_7\ b_{10}$$
$$C = b_2\ b_5\ b_8$$

Notar que A , B y C no tienen porqué tener el mismo largo y además notar que la suma del largo de las tres partes es igual al largo de X.

- Luego, deberás analizar la suma entre el primer byte de A, el byte o bytes centrales[4] de B y el último byte de C.
  - Si la suma de los valores es número par, el resultado encriptado será:

    $$nCAB$$

    donde n es igual a 0 para indicar que se cumplió esta condición.
  - Si la suma de los valores es un número impar, el resultado encriptado será:

    $$nACB$$

    donde n es igual a 1 para indicar que se cumplió esta condición.

Ahora, lo último que necesitarás para completar la comunicación segura es **diseñar una forma para desencriptar el mensaje** desde el *receptor* (`cliente` o `servidor`).

Para la corrección de Método de encriptación, se proveerá de un archivo, detallado en la sección Scripts, con **dos funciones** (`encriptar` y `desencriptar`) que deberás completar para realizar el proceso de

[4]Se considera byte central como el byte que se ubica en la mitad de la secuencia. Si la secuencia tiene un largo impar se tendrá solo un byte central. En el caso de que el largo sea par se tendrán dos bytes centrales. Por ejemplo en el caso dado los bytes centrales de B (secuencia de largo par) serian $b_4\ b_7$, mientras que el byte central de C (secuencia de largo impar) seria solo b_5.

encriptación y desencriptación de los mensajes. Este ítem será evaluado de forma automatizada en donde se comparará el resultado entregado por estas funciones con el resultado esperado. En el mismo archivo se incluye un ejemplo de cómo se verificará el correcto funcionamiento de ambas funciones.

3.1.5. Ejemplo de encriptación

Para ayudarte a entender de mejor forma el método de encriptación que **debes** implementar, utilizaremos como ejemplo el siguiente contenido de mensaje (previo a la encriptación):

b'\x05\x08\x03\x02\x04\x03\x05\x09\x05\x09\x01'

- Lo primero que hacemos es separar el ***bytearray*** en 3 partes de la siguiente forma:

A = b'\x05\x02\x05\x09'
B = b'\x08\x04\x09\x01'
C = b'\x03\x03\x05'

- Se toma el primer *byte* de A, los *bytes* centrales de B y el último *byte* de C, y son sumados:

$$5_A + (4_B + 9_B) + 5_C = 23$$

- Como 23 es un número impar el resultado de la encriptación sería el siguiente:

nACB

b'\x01\x05\x02\x05\x09\x08\x04\x09\x01\x03\x03\x05'

3.1.6. Ejemplo de codificación

Para ayudarte a entender el método de codificación explicado anteriormente, supongamos que se quiere enviar el mensaje ya encriptado mostrado en el ejemplo anterior desde el `servidor` al `cliente`.

- Según lo especificado, lo primero que se enviará será un mensaje de 4 *bytes* en formato *big endian* que contiene el largo del contenido. En este caso el largo es de 12 *bytes*.
- Luego se separa el contenido en bloques de 32 *bytes*. En este caso el contenido del mensaje solo tiene 12 *bytes*, por lo que será necesario rellenar el mensaje con *bytes* ceros hasta completar el bloque.
- Para cada uno de los bloques deberás enviar el número del bloque correspondiente en un mensaje de 4 *bytes* en formato *little endian*. Posterior a ello, se debe enviar el bloque de 32 *bytes* que contiene parte del mensaje.
- Una vez que envíes todos los bloques el proceso de envió habrá finalizado. En este caso se envía un único bloque y luego finaliza el proceso .

En la siguiente figura se puede observar la estructura para enviar el mensaje ya encriptado mostrado en el ejemplo anterior:

3.1.7. *Logs* del servidor

Como el servidor no cuenta con interfaz gráfica, debes indicar constantemente lo que ocurre en él mediante la consola. Estos mensajes se conocen como mensajes de *log*, es decir, deberás llamar a la función `print` cuando ocurra un evento importante en el servidor. Estos mensajes se deben mostrar para cada uno de los siguientes eventos:

- Se conecta o desconecta un cliente al servidor.

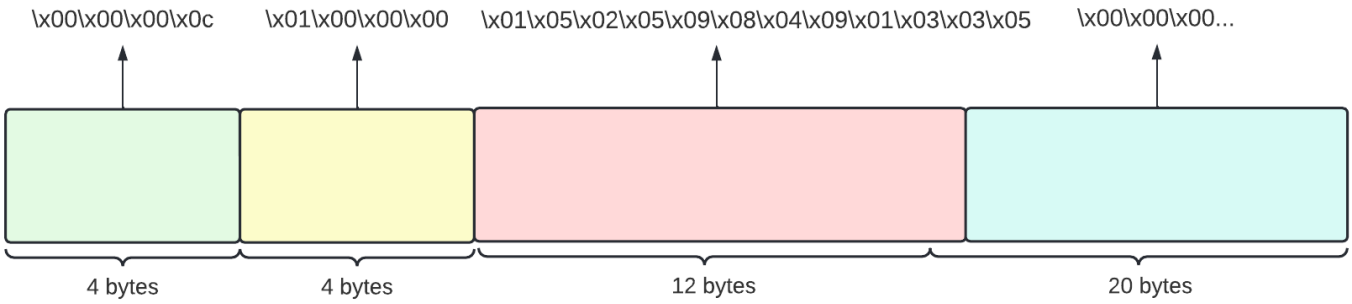

Figura 3: Ejemplo de estructura del *bytearray* para un mensaje codificado

- Un cliente ingresa un nombre de usuario. Debes indicar el nombre y si es válido o no.
- Un cliente intenta ingresar a la sala de espera, mencionar si la sala esta llena o no .
- Cuando se comienza una partida. Debe indicar quienes están en esa partida.
- Cuando un jugador lanza una carta. Debe indicar quien la lanzo y el tipo de carta.
- Cuando a un jugador se le acaba el tiempo de ronda.
- Cuando finaliza una ronda. Se debe indicar el ganador o si ocurrió un empate.
- Se termina la partida, debes indicar cual fue el ganador de la partida.

Es de **libre elección** la forma en que representes los *logs* en la consola, un ejemplo de esto es el siguiente formato:

```
     Cliente     |        Evento          |      Detalles
-----------------|------------------------|------------------------
     drcid98     |       Conectarse       |          -
-----------------|------------------------|------------------------
    gvfigueroa   |      Carta lanzada     |     Carta tipo fuego
-----------------|------------------------|------------------------
    drcid98      |      Carta lanzada     |     Carta tipo nieve
-----------------|------------------------|------------------------
        -        |      Fin de ronda      |  Ganador:  gvfigueroa
-----------------|------------------------|------------------------
        -        |   Término de partida   |  Ganador:  gvfigueroa
-----------------|------------------------|------------------------
```

3.1.8. Desconexión repentina

En caso de que algún ente se desconecte, ya sea por error o a la fuerza, tu programa debe reaccionar para resolverlo:

- Si es el `servidor` quien se desconecta, cada cliente conectado debe mostrar un mensaje explicando la situación antes de cerrar el programa.
- Si es un `cliente` quien se desconecta, se descarta su conexión. Si se desconecta mientras está en una partida, el oponente ganara automáticamente.

3.2. Roles

A continuación se detallan las funcionalidades que deben ser manejadas por el servidor y las que deben ser manejadas por el cliente.

3.2.1. Servidor

- **Procesar y validar** las acciones realizadas por los clientes. Por ejemplo, si deseamos comprobar que el jugador ingresa un nombre correcto, el servidor es quien debe verificarlo y enviarle una respuesta al cliente según corresponda.
- **Distribuir y actualizar** en tiempo real los cambios correspondientes a cada uno de los participantes del juego, con el fin de mantener sus interfaces actualizadas y que puedan reaccionar de manera adecuada. Por ejemplo, si un jugador utiliza una carta en una partida, el servidor deberá notificar de esto a todos los clientes en la partida y se deberá reflejar en la interfaz de cada uno de ellos.
- **Almacenar y actualizar** la información de cada cliente y sus recursos. Por ejemplo, el servidor manejará la información de las barajas que tiene cada uno de los clientes conectados y las actualizará una vez que utilicen una carta.

3.2.2. Cliente

Todos los clientes cuentan con una interfaz gráfica que le permitirá interactuar con el programa y enviar mensajes al servidor. Maneja las siguientes acciones:

- **Enviar todas las acciones** realizadas por el usuario hacia el servidor.
- **Recibir e interpretar** las respuestas y actualizaciones que envía el servidor.
- **Actualizar** la interfaz gráfica de acuerdo a las respuesta que recibe del servidor.

4. Reglas de *DCCard-Jitsu*

4.1. Juego

DCCard-Jitsu es un juego que se desarrolla en rondas al estilo del *cachipún*, donde cada jugador debe elegir una carta de su baraja para jugar. Al principio de cada partida, se deberán barajar todas las cartas disponibles del jugador y se mostrarán en pantalla solo las primeras `BARAJA_PANTALLA`. [5]

Existen tres elementos distintos de cartas: Fuego, Agua y Nieve. La ventaja de cada uno se muestra a continuación.

- Fuego le gana a Nieve.
- Nieve le gana a Agua.
- Agua le gana a Fuego.

Además de poseer un elemento, cada carta es de un único color (rojo, verde o azul) y tendrá un número natural del 1 al 5, que indica su poder de ataque.

[5] Corresponde a un número de cartas para utilizar en la ronda

Figura 4: Ejemplo de carta *DCCard-Jitsu*

En el caso de que ambos jugadores jueguen una carta del mismo elemento, ganará la carta que tenga mayor valor de poder de ataque. En el caso de que haya un empate entre estos valores, se descartarán ambas cartas (empate) y se procederá a jugar la siguiente ronda sin ganador.

Cada vez que un jugador gane una ronda, la carta que utilizó en esa ronda pasará a su mazo de triunfos, y la carta del perdedor se descartará. El objetivo de *DCCard-Jitsu* es ganarle al oponente consiguiendo la combinación correcta de cartas ganadoras en su mazo de triunfos. Para ganar una partida, el jugador deberá tener tres cartas de colores distintos en su mazo de triunfos, **cumpliendo con una de las siguientes opciones**:

- Opción A: todas del mismo elemento.
- Opción B: todas de distintos elementos.

La partida continuará hasta que un jugador haya alcanzado una de las opciones de combinaciones mencionadas. En el momento en que un jugador juegue una carta, automáticamente se le entregará, o bien, sacará, la primera carta de su mazo para poder seguir jugando. Manteniendo así un número de cartas igual a `BARAJA_PANTALLA` en su baraja mostrada en pantalla, utilizando las cartas que se encuentren en su mazo.

Cada vez que se descarte alguna carta de un jugador, esto es, cuando pierde o empata en una ronda, la carta descartada pasará al final de su mazo, donde se encuentran el resto de sus cartas que no se están mostrando en pantalla actualmente. De esta manera, siempre habrán cartas para rellenar la baraja.

4.2. Ronda

Al comenzar cada ronda, partirá una cuenta regresiva de `CUENTA_REGRESIVA_RONDA` (medida en segundos), que indicará el tiempo restante que tienen los jugadores para elegir sus respectivas cartas. Los jugadores deben seleccionar una carta, luego confirmar esta carta mediante un botón de confirmación. Si, al acabarse el tiempo de la cuenta regresiva, algún jugador todavía no ha seleccionado **y confirmado** una carta, entonces se escogerá una carta aleatoriamente de su baraja para jugar. Es decir, si el jugador tiene seleccionada una carta, pero no la confirmó dentro del límite de tiempo, también se escogerá una aleatoriamente. Luego de que ambos jugadores tengan sus cartas, se procederá a mostrar ambas y ver quién fue el ganador o si fue un empate.

5. Interfaz gráfica

5.1. Ventana de inicio

Esta ventana se despliega al inicio del programa y pedirá que se introduzca el nombre del jugador actual. Este será el nombre que se mostrará a los demás jugadores de la partida y **debe cumplir** con los requisitos indicados en Flujo del programa. Para ingresar se debe hacer click en un botón de confirmación o usar los *sprites* creados para usar la **puerta del Dojo** como botón de confirmación (la que se contorneará con un color amarillo al poner el mouse sobre ella). Si el nombre es válido, se permitirá continuar a la ventana de espera, mientras que en caso contrario se debe notificar al usuario y permitir ingresar otro nombre. Elementos mínimos de la ventana de inicio:

- Label de input de texto.
- Botón de confirmación del *nombre* y entrada al juego.

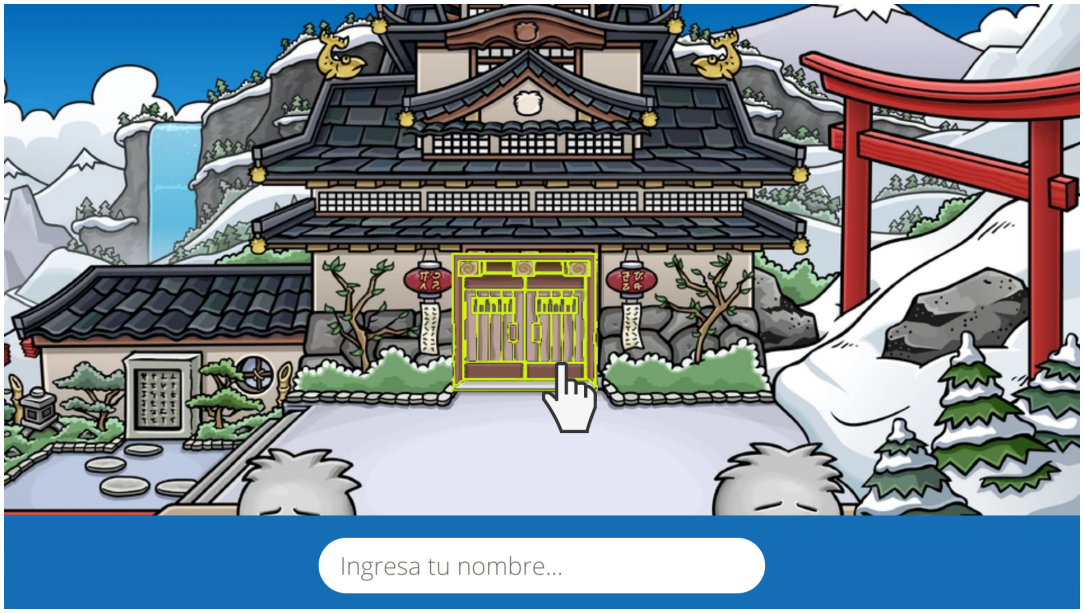

Figura 5: Ejemplo de ventana de inicio

En la figura 5, la puerta representa el botón de confirmación. En el caso de que un usuario se conecte y la sala de espera esté llena, se le debe notificar y mantener en la ventana de inicio hasta que esta sea liberada.

5.2. Ventana de espera

Esta ventana se despliega luego de introducir el nombre en el inicio. En esta se deben desplegar los nombres de los dos jugadores que ingresarán a la partida y un botón que permita salir de esta ventana, para volver a la venta de inicio. En el caso de que haya un solo jugador, se debe mostrar en pantalla que se está esperando a otro jugador para iniciar la partida.

Cuando otro jugador ingrese a la sala, debe aparecer un timer que indique los `CUENTA_REGRESIVA_INICIO` segundos restantes para que la partida se inicie de forma automática.

Luego de iniciar partida, todos los jugadores avanzan a la ventana de juego.

Elementos mínimos de la ventana de espera:

- Nombres de jugadores.
- Timer de la cuenta regresiva.
- Botón para volver a la ventana de inicio.

Figura 6: Ejemplo de ventana de espera jugador

5.3. Ventana de juego

En esta sala es donde se desarrollará la partida. Debe contar con 3 secciones:

- Sección de cartas disponibles para jugar junto a un botón que permita confirmar la carta elegida.
- Sección donde se muestra la carta del oponente sin revelar junto a la carta de selección previa. Es importante indicar a quién pertenece cada carta. Además, junto a las cartas del oponente deben mostrarse sus `BARAJA_PANTALLA` cartas ocultas (en la ventana de ejemplo hay 5 cartas por baraja). Por último debe haber un timer que indique los `CUENTA_REGRESIVA_RONDA` segundos restantes para realizar la jugada antes de que el programa seleccione una carta al azar.
- Cartas de victoria de cada jugador en cada ronda, que indiquen el color y el elemento de la carta que ha ganado cada jugador.

Figura 7: Ejemplo de ventana de juego

5.4. Ventana Final

Esta ventana debe mostrar únicamente si el jugador ganó o perdió la partida y además debe contemplar un botón para volver a la ventana de inicio.

Figura 8: Ejemplo de ventana final

6. Archivos

Para desarrollar tu programa de manera correcta, deberás crear o utilizar los siguientes archivos:

6.1. Archivos entregados

6.1.1. Sprites

Carpeta que contiene todos los elementos visuales que necesitas para llevar a cabo tu tarea, entre ellos encontrarás las subcarpetas:

- **Cartas:** Contienen las cartas del juego. Están en formato color_elemento_numero.png, siendo número el valor del poder de ataque. Además se encuentra el reverso de las cartas back.png.

Figura 9: back.png

- **Botones:** Contienen los algunos botones del juego.

Figura 10: botones

- **Background:** Contiene los sprites de los fondos disponibles para las distintas ventanas.
- **Elementos:** Contiene los elementos del juego.

(a) *Agua*

(b) *Fuego*

(c) *Nieve*

Figura 11: *Sprites* de los elementos posibles para cada carta

Además **Elementos** contiene la subcarpeta *Fichas* que contiene las fichas posibles que se usan para controlar la cantidad de puntos de los jugadores.

(a) *Agua con fondo Azul*

(b) *Fuego con fondo Rojo*

(c) *Nieve con fondo Azul*

Figura 12: *Sprites* de ejemplos de algunas fichas posibles.

- **Bonus:** Contiene los *sprites* necesarios para implementar el bonus.

6.1.2. Scripts

Carpeta que contiene todos los scripts extras que necesitas para llevar a cabo tu tarea, entre ellos encontrarás:

- **cartas.py:** Contiene la función get_penguins() que retorna un diccionario de 15 diccionarios con la información necesaria para crear cada carta. Cada subdiccionario tiene la estructura:
  - elemento = Elemento de la carta.
  - color = Color de la carta.
  - puntos = Valor poder de ataque de la carta.
- **cripto.py:** Contiene **dos funciones que deberás completar** para testear la correcta encriptación y desencriptación del mensaje respectivamente. Este archivo se utilizará para evaluar los ítems asociados a la correcta implementación del Método de encriptación, además de ser utilizado en la comunicación **cliente - servidor**. Este archivo deberá estar presenta tanto en la carpeta de servidor como del cliente[6]. Para su corrección, se ejecutará **únicamente** el archivo presente en la carpeta del cliente.

6.2. Archivos a crear

6.2.1. `parametros.json`

Dentro de tu tarea deben existir dos archivos de parámetros independientes entre sí: uno para el servidor, y otro para el cliente. Cada archivo de parámetros debe contener solamente los parámetros que corresponden a su respectiva parte, es decir, `parametros.json` en la carpeta `cliente/` deberá contener solamente

[6]Deberás duplicar este archivo

parámetros útiles para el cliente (como los *paths*), mientras que `parametros.json` de `servidor/` contendrá solamente parámetros útiles para el servidor. A diferencia de las tareas pasadas, los parámetros en `ESTE_FORMATO` deberán ir en este archivo, junto con su respectivo valor. Los archivos deben encontrarse en formato `JSON` (**NO un archivo de extensión `.py`**), como se ilustra a continuación:

```
{
    "host": <direccion_ip>,
    "port": <puerto>,
    ...
}
```

7. *Bonus*

En esta tarea habrá una serie de *bonus* que podrás obtener. Cabe recalcar que necesitas cumplir los siguientes requerimientos para poder obtener *bonus*:

1. Para todos los bonus, la acción debe ser validada por servidor.
2. La nota en tu tarea (sin bonus) debe ser **igual o superior a 4.0**[7].
3. El bonus debe estar implementado **en su totalidad**, es decir, **no se dará puntaje intermedio**.

Finalmente, la cantidad máxima de décimas de *bonus* que se podrá obtener serán 5 décimas. Deberás indicar en tu `README` si implementaste alguno de los bonus, y cuáles fueron implementados.

7.1. Bonus Cheatcodes (1 décima)

Si notas que la partida no va a tu favor, puedes perjudicar a tus contrincantes implementando **dos cheatcode**:

- **V + E + O**: Al presionar esta combinación de teclas, podrás visualizar la baraja de tu contrincante en la ventana de juego.

7.2. Botón de bienestar (3 décima)

Puede ocurrir que a veces te encuentres en una situación difícil al no saber qué carta utilizar por su bajo poder de ataque. Es por esto que deseas implementar un botón para ayudarte a ganar.

El botón de bienestar permitirá ganar automáticamente la ronda al jugador que lo utilice, pero posee ciertas restricciones:

- Se puede utilizar si y sólo si, el jugador, que oprime el botón, ya jugó tres cartas de distinto color.
- Solamente se podrá utilizar una vez en la partida.

7.3. Chat (5 décimas)

Para lograr hacer un juego más interactivo, deberás implementar un chat que te permitirá conversar con el otro jugador.

La implementación del chat deberá ser a través de una ventana adicional, que se deberá abrir junto a la ventana de juego.

En esta se deben cumplir las siguientes funcionalidades:

[7] Esta nota es sin considerar posibles descuentos.

- Visualizar los mensajes junto con el jugador que envió cada mensaje.
- Permitir escribir un mensaje.
- Actualización en tiempo real.

8. `.gitignore`

Para esta tarea **deberás utilizar un** `.gitignore` para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `Tareas/T3/`. Puedes encontrar un ejemplo de `.gitignore` en el siguiente ***link***.

Para esta tarea, deberás ignorar en específico los siguientes archivos:

- Enunciados
- `Sprites`

Se espera que no se suban archivos autogenerados por las interfaces de desarrollo o los entornos virtuales de Python, como por ejemplo: la carpeta `__pycache__`.

Para este punto es importante que hagan un correcto uso del archivo `.gitignore`, es decir, los archivos no **deben** subirse al repositorio debido al archivo `.gitignore` y no debido a otros medios.

9. Entregas atrasadas

Posterior a la fecha de entrega de la tarea se abrirá un formulario de Google Form, en caso de que desees que se corrija un *commit* posterior al recolectado, deberás señalar el nuevo *commit* en el *form*.

El plazo para rellenar el *form* será de 24 horas, en caso de que no lo contestes en dicho plazo, se procederá a corregir el *commit* recolectado.

10. Importante: Corrección de la tarea

Para esta tarea, el carácter funcional del programa será el pilar de la corrección, es decir, **sólo se corrigen tareas que se puedan ejecutar**. Por lo tanto, se recomienda hacer periódicamente pruebas de ejecución de su tarea y *push* en sus repositorios.

En la **distribución de puntajes**, se señalará con color:

- **Amarillo**: cada ítem que será evaluado a nivel de código, todo aquel que no esté pintado de amarillo significa que será evaluado si y sólo si se puede probar con la ejecución de su tarea.
- **Azul**: cada ítem en el que se evaluará a través del archivo **cripto.py**. Si el ítem está implementado, pero no se implementa este archivo, según lo especificado en el enunciado, no se evaluará con el puntaje completo.

En tu archivo `README.md` deberás señalar el archivo y la línea donde se encuentran definidas las funciones o clases relacionados a esos ítems.

Finalmente, si durante la realización de tu tarea se te presenta algún problema o situación que pueda afectar tu rendimiento, no dudes en contactar a la ayudante jefa de Bienestar al siguiente correo: **bienestar.iic2233@ing.puc.cl**.

11. Restricciones y alcances

- Esta tarea es **estrictamente individual**, y está regida por el **Código de honor de Ingeniería**.
- Tu programa debe ser desarrollado en Python 3.10.
- Tu programa debe estar compuesto por uno o más archivos de extensión `.py`.
- Si no se encuentra especificado en el enunciado, supón que el uso de cualquier librería Python está prohibido. Pregunta en la *issue* especial del **foro** si es que es posible utilizar alguna librería en particular.
- Debes adjuntar un archivo `README.md` **conciso y claro**, donde describas los alcances de tu programa, cómo correrlo, las librerías usadas, los supuestos hechos, y las referencias a código externo. **Tendrás hasta 2 horas después del plazo de entrega** de la tarea para subir el `README` a tu repositorio.
- Tu tarea podría sufrir los descuentos descritos en la **guía de descuentos**.
- Entregas con atraso de más de 24 horas tendrán calificación mínima (1,0).
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro.

Las tareas que no cumplan con las restricciones del enunciado obtendrán la calificación mínima (1,0).