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

IIC2233 Programación Avanzada (2020-1)

Tarea 03

Entrega

- **Avance de tarea**
  - **Fecha y hora:** miércoles 24 de junio de 2020, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T03/`
- **Tarea**
  - **Fecha y hora:** sábado 4 de julio de 2020, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T03/`
- `README.md`
  - **Fecha y hora:** lunes 6 de julio de 2020, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T03/`

Objetivos

- 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. *DCCuatro*

Se habían cumplido tres meses de distanciamiento social, y ya no sabías qué hacer en tu tiempo libre: habías agotado completamente el catálogo de Netflix, has jugado un centenar de partidas de Ludo e incluso ya te sabías el diccionario al derecho y al revés tras tantas horas de Scrabble *online* con tus amigos. Un día, la inspiración llegó a ti y tuviste la idea de, con tus nuevas habilidades de *networking* y serialización, crear el mejor juego de mesa que la humanidad hubiera visto. Su nombre sería... ***DCCuatro***.

Figura 1: Logo de *DCCuatro*.

DCCuatro es un juego de cartas por **turnos** en donde un grupo de jugadores deberá competir por mantenerse en el juego y ser el primero en quedarse sin cartas en su mano. Para jugar, se utiliza un **mazo** que contiene tanto cartas con valores numéricos y colores, como cartas especiales que tienen efectos que influirán en el flujo del juego.

2. Flujo del programa

Al ser la interacción entre jugadores vital para *DCCuatro*, es necesario que el juego cuente con un **servidor** que pueda transmitir datos entre jugadores y manejar el flujo del juego en sí, por lo que siempre se debe comenzar ejecutando este servidor previo a la ejecución del programa por parte de los jugadores.

Cada jugador debe correr una instancia de **cliente**, programa que iniciará una interfaz gráfica la cual será su único medio de interacción con el juego. Inicialmente, el usuario se encontrará con una **Ventana de Inicio** donde deberá ingresar su **nombre**. En caso de que este sea válido, no esté siendo utilizado por otro cliente conectado, y la partida todavía tenga cupos para un nuevo integrante, el jugador procederá a la **Sala de Espera**. En caso contrario, la interfaz deberá mantenerse en la Ventana de Inicio y notificar al jugador que no ha sido posible continuar, dando la posibilidad de intentarlo nuevamente.

Una vez dentro de la **Sala de Espera**, los jugadores deben esperar a que ingrese el número necesario de jugadores para iniciar la partida, valor definido por el parámetro `CANTIDAD_JUGADORES_PARTIDA`. Cuando este valor se cumple, el servidor redirige inmediatamente a todos los jugadores a la **Sala de Juego**, donde podrán dar inicio a una nueva partida de *DCCuatro*.

Al iniciar la partida, se define el orden de turnos en que los jugadores participarán, y a cada uno se le entrega una **mano** de cartas inicial que usará en la partida. Estas cartas provienen del **mazo**, siendo generadas aleatoriamente a partir de una función ya implementada. También se encuentra el **pozo**, que corresponde al grupo de cartas que han sido jugadas en turnos pasados. De ellas, la última carta jugada es la única que es visible. Al iniciar la partida, el pozo siempre tiene una carta.

Al iniciar el turno de un jugador, este tiene dos posibilidades de acción: puede **jugar una carta de su mano**, siempre y cuando las reglas del juego la consideren válida; o **robar una carta del mazo**, la cual

se agrega a sus cartas en mano. Además, ya sea el turno del jugador o no puede **gritar ¡*DCCuatro*!**. Cada acción que puede traer distintas consecuencias. Los específicos del flujo del juego en si y sus turnos se detallan en Reglas del *DCCuatro*.

Un jugador gana en *DCCuatro* cuando **logra despojarse de todas las cartas de su mano**, o cuando **es el último jugador participando activamente en la partida**. Por otro lado, un jugador es eliminado cuando **sus cartas en mano sobrepasan la cantidad máxima permitida**, o cuando **otro jugador logra despojarse de todas sus cartas primero**. Si se pierde mediante la primera opción, el jugador permanece en la partida como espectador, pero no puede participar de ella.

Una vez que el juego ha definido a un **ganador**, la partida se da por acabada. A cada participante se le informará mediante la interfaz si ganó o perdió la partida, y le dará la opción de regresar a la **Ventana de Inicio** para que pueda unirse a una nueva partida.

3. Reglas del *DCCuatro*

Como ya se ha mencionado, *DCCuatro* posee un mazo de cartas bastante particular y un conjunto de reglas a seguir durante la partida. A continuación se explican el funcionamiento de las cartas del juego, y las reglas por las que tu programa se debe regir durante la partida:

3.1. Tipos de cartas

DCCuatro ofrece dos categorías de cartas: **cartas regulares** y **cartas especiales**. Las cartas regulares poseen un número del 0 al 4 inclusive, y un color que puede ser rojo, azul, amarillo, o verde.

3.1.1. Cartas especiales

Existen tres cartas especiales:

- **Cambio de sentido**: al jugar esta carta, se invierte el orden de los turnos de los jugadores. Esta carta posee un color, y solo se puede jugar si la que está en el pozo es otra carta de tipo cambio de sentido o la carta en el pozo posee el mismo **color** y **no sea** un +2.
- **Cambio de color**: al jugar esta carta, el jugador puede elegir un color. Esto hace que en el siguiente turno solo se puedan jugar cartas del color que fue seleccionado[1] u otra carta de cambio de color. Se puede jugar esta carta siempre y cuando **no** se haya jugado una carta +2 en el turno anterior.
- **+2** : al jugar esta carta, el siguiente jugador tiene solo dos opciones: jugar otra carta +2 de su mano[2]; o robar un acumulado de cartas del mazo. El número de cartas acumuladas es dos veces la cantidad de cartas +2 jugadas sucesivamente sin que ningún jugador robe. Si juega un +2 o roba el acumulado, el turno del jugador termina inmediatamente. Esta carta tiene un color, y solo puede ser jugada si la que está en el pozo posee el mismo color o es otra carta +2.

(a) Carta regular (b) +2 (c) Cambio de sentido (d) Cambio de color

Figura 2: Diferentes tipos de cartas del *DCCuatro*.

[1] Incluyendo las cartas especiales +2 y cambio de sentido.
[2] Independiente del color del +2 jugado en el turno anterior.

3.2. Jugar una carta

Durante el turno de un jugador, este solo puede jugar una carta en su mano si la carta es *jugable*. Una carta se considera jugable si se cumple **alguna** de las siguientes condiciones:

- Tiene el mismo **número** que la carta en el pozo.
- Tiene el mismo **color** que la carta en el pozo.
- Es una **carta especial** y se cumplen sus condiciones para ser jugada.

Después de que un jugador juega una carta, su turno termina.

3.3. Robar cartas del mazo

Una mecánica muy importante en *DCCuatro* es el **robo** de cartas. Un jugador puede decidir robar una carta del mazo siempre y cuando no haya jugado una carta en el mismo turno[3]. Si un jugador no posee ninguna carta jugable en su mano, debe robar una carta del mazo para poder terminar su turno. Alternativamente, un jugador puede decidir robar una carta del mazo a pesar de tener una o más cartas jugables en su mano.

Al implementar esta mecánica en *DCCuatro* tienes que tener en cuenta las siguientes consideraciones:

- La única situación en que un jugador puede robar múltiples cartas es por el efecto de la carta especial +2. En cualquier otro caso, un jugador roba **una** sola carta.
- Como regla general, cuando un jugador roba una o múltiples cartas del mazo, independiente del motivo, su turno termina automáticamente.
- El juego nunca robará una carta del mazo automáticamente por un jugador[4]. Esto solo ocurre cuando el jugador decide robar del mazo.

3.4. Límite de cartas y término de juego

Un jugador puede tener como máximo hasta `10` cartas en su mano, esto será así en todo momento de la partida. En el caso de que un jugador robe una o múltiples cartas y supere este límite, **pierde** automáticamente. Esto significa que el juego salta el turno de este jugador de este momento hasta que termine la partida.

Por otro lado, cuando un jugador se queda sin cartas en su mano, éste automáticamente **gana** y el juego se acaba. Alternativamente, un jugador puede ganar si todos los demás jugadores **pierden** antes que él.

3.5. Grita ¡*DCCuatro*!

Todo jugador activo tiene la capacidad de gritar ¡*DCCuatro*! en cualquier momento de la partida con el objetivo de evitar que otros jugadores ganen:

- Cuando un jugador se queda con una carta en su mano debe gritar ¡*DCCuatro*! lo antes posible.
- Si otro jugador grita ¡*DCCuatro*! antes que el jugador con una carta, este último roba `4` cartas.
- Si un jugador grita ¡*DCCuatro*! erróneamente[5], este debe robar `4` cartas.
- Un jugador que haya perdido **NO** puede gritar ¡*DCCuatro*!.

[3] Porque luego jugar una carta, el turno termina automáticamente.

[4] A pesar de que la única opción que este tenga sea robar una carta.

[5] Ya sea porque no hay jugadores con una carta en su mano o el jugador con una carta en su mano logró decir ¡*DCCuatro*! antes.

4. *Networking*

Para lograr que los jugadores de *DCCuatro* puedan disfrutar de esta experiencia durante la cuarentena, deberás implementar una arquitectura cliente-servidor. Para esto, tendrás que hacer uso del *stack* de protocolos de red **TCP/IP** a través del módulo `socket`, el cual proporciona las herramientas necesarias para administrar la conexión entre dos computadores conectados a una misma red local[6].

Debes implementar dos programas: el programa del **servidor** y el programa del **cliente**. El servidor es el primero que debe ejecutarse. Luego de empezar, el servidor queda a la espera de que uno o más clientes se conecten a él. En este modelo, la comunicación ocurre exclusivamente entre cada cliente con un único servidor, es decir, **nunca directamente entre dos clientes**.

4.1. Arquitectura cliente-servidor

En esta sección se presentarán consideraciones que deberás aplicar en la elaboración de tu tarea. **Todo lo que se indique deberás seguirlo al pie de la letra**, mientras que lo que no se encuentre especificado podrás implementarlo de la forma en que te sea más fácil mientras cumpla con lo mínimo pedido (siéntete libre de **preguntar** por este mínimo si no está especificado o si no queda completamente claro).

4.1.1. Separación funcional

El cliente y el servidor deberán estar separados, lo que implica que deben estar contenidos en directorios diferentes, uno llamado `cliente`, y el otro llamado `servidor`. Cada uno debe tener un archivo llamado `main.py` (archivo principal a ejecutar) y debe contar con todos los módulos y demás archivos necesarios para su correcta ejecución. El siguiente diagrama ilustra esta estructura.

```
T03
├── cliente
│   ├── main.py
│   ├── parametros.json
│   ├── un_archivo_solo_del_cliente.uwu
│   └── ...
├── servidor
│   ├── main.py
│   ├── parametros.json
│   ├── un_archivo_solo_del_servidor.uwu
│   └── ...
├── .gitignore
├── README.md
├── un_archivo_que_no_es_del_cliente_ni_del_servidor.uwu
└── ...
```

Aunque el cliente y el servidor compartan el mismo directorio padre (`T03`), esto es así solo para efectos de la entrega de la tarea y la ejecución de uno no podrá depender en absoluto de ningún archivo del otro, es decir, el cliente y el servidor deben estar implementados como si estuvieran en computadores diferentes. En la siguiente figura se muestra un diagrama que explica la separación esperada entre los distintos componentes de tus programas:

[6] Una red local es una infraestructura que conecta uno o más dispositvos dentro un espacio físico reducido como una casa, departamento u oficina. Puedes leer más en **este artículo en Wikipedia (Red de área local)**.

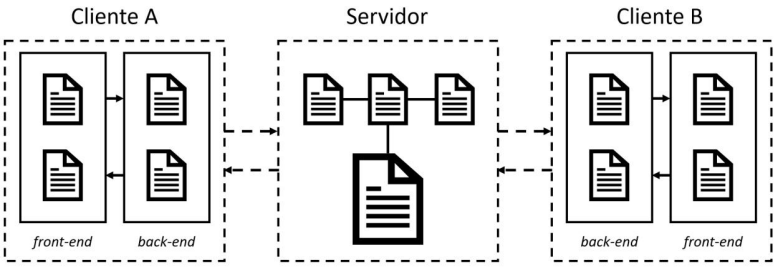


Figura 3: Separación cliente-servidor y *front-end-back-end*.

La comunicación entre el *back-end* y el *front-end* se debe realizar mediante señales, mientras que la comunicación entre el cliente y el servidor debe realizarse mediante *sockets*. Únicamente los clientes presentarán una interfaz gráfica.

4.1.2. Conexión

El servidor abrirá un *socket* haciendo uso de los datos encontrados en el archivo `servidor/parametros.json`. Tal como sugiere su extensión, éste será de tipo JSON, y seguirá el formato mostrado a continuación. Por otra parte, el cliente deberá conectarse al *socket* abierto por el servidor haciendo uso de los datos encontrados en `cliente/parametros.json`.

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

4.1.3. Envío de información

Dada la estructura cliente-servidor en la que se basa *DCCuatro*, a lo largo de la ejecución de tu programa el servidor y el cliente deberán intercambiar información en varias situaciones (descritas en más detalle en Roles). Por ejemplo, el cliente debe comunicarle al servidor cuando un jugador quiera entrar a la partida y este debe responderle al cliente con una confirmación o rechazo.

Queda a tu criterio cómo estructurar la comunicación entre el cliente y el servidor, siempre y cuando respete todo lo indicado en *Networking*. Sin embargo, **la única excepción a lo anterior es al implementar el envío de cartas** entre el cliente y el servidor. Ese envío debes implementarlo según la siguiente especificación:

Envío de las cartas

Una de las principales ventajas de jugar *DCCuatro* es que el cliente no necesita tener acceso a las imágenes de manera local. Cada vez que un jugador tenga que robar cartas o sea necesario actualizar la última carta del pozo, el servidor debe proporcionar la imagen de dicha carta junto con su información. Para lograr esto, el servidor debe serializar:

- El color de la carta.
- El número o tipo de la carta.
- La imagen asociada a dicha carta.

Todo esto debe ser codificado en un único *bytearray* según la siguiente estructura:

- Los primeros 4 *bytes* indican el tipo de objeto serializado. Para lograr esto, debes asignar un identificador numérico en el formato *big endian*[7] a cada tipo de objeto enviado. El identificador para cada tipo de objeto es el siguiente:
  - Color de la carta: `1`
  - Número o tipo de carta: `2`
  - Imagen de la carta: `3`
- Los siguientes 4 *bytes* indican el largo X de *bytes* asociado al siguiente bloque y tendrán que estar en formato *little endian*.
- Los siguientes X *bytes* corresponden al objeto serializado.
- Esto se repite para el resto de información (que involucre a cartas) que se envía al cliente.

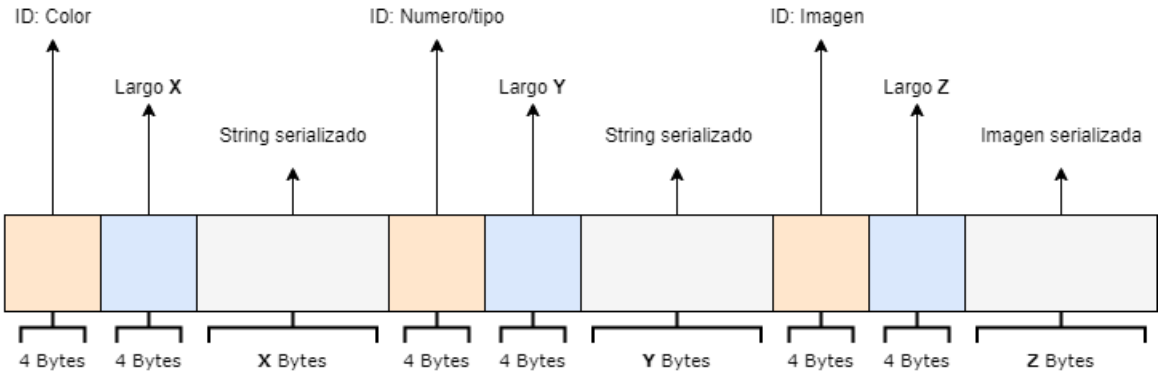


Figura 4: Ilustración del *bytearray*.

4.1.4. *Logs* del servidor

El servidor **NO** cuenta con una interfaz gráfica. En su lugar, debe dejar registros de lo que ocurre en él constantemente mediante la consola. Estos mensajes se conocen como mensajes de *log*, que para efectos prácticos se traducen en llamar a `print` cada vez que ocurra un evento importante en el servidor. Específicamente, se debe imprimir un mensaje de *log* al menos cada vez que:

- Un cliente se conecte al o desconecte del servidor, identificando el cliente. Para identificarlo, se le debe asignar un nombre o *id* al cliente al conectarse.
- Un cliente intente ingresar a la sala de espera.
- El cliente robe una carta del mazo. Se deberá indicar el nombre del cliente y la acción en cuestión.
- El cliente juegue una carta. Se deberá indicar el nombre del cliente, la acción y la carta jugada por parte del cliente.
- El servidor mande una carta a algún cliente. Se deberá indicar las características de la carta.
- El servidor mande o reciba algún otro tipo de mensaje a algún cliente. Debe indicar a qué cliente, el tamaño del mensaje y su contenido.

[7] El *endianness* es el orden en el que se guardan los bytes que representan un numero. Esto es relevante porque cuando intentes usar los métodos `int.from_bytes` e `int.to_bytes` deberá proporciones el *endianness* que debes usar, además de la cantidad de bytes que quieres usar para representarlo. Para más información puedes revisar este **enlace**.

El formato para hacer los *logs* **queda a tu criterio mientras exponga toda la información de forma ordenada**, pero se recomienda un formato similar a:

```
 Cliente   |          Evento          |         Detalles
---------- | ------------------------ | --------------------------------
   Kuky    |        Conectarse        |               -
  Gaspar   |        Pedir carta       |               -
  Gaspar   |       Enviar carta       |           0 rojo
   Hugo    |        Jugar carta       |           1 verde
  Cachu    |      Recibir mensaje     |  Acción: Gritar DCCuatro
```

La implementación de estos *logs* puede ser de mucha ayuda, ya que te permitirá comprobar el correcto funcionamiento del servidor y potencialmente te puede ayudar a encontrar *bugs* de funcionamiento.

4.1.5. Desconexión repentina

En caso de que algún ente se desconecte ya sea por un 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 sala de espera, entonces deja de ser considerado para el juego y el cupo queda disponible. Si se desconecta mientras hay una partida en curso, este automáticamente pierde, de forma que el resto pueda continuar con la partida.

4.2. Roles

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

4.2.1. Servidor

Dentro de *DCCuatro* existe una única sala de juego, que se abre inmediatamente luego de iniciado el servidor. Esta sala tiene una capacidad máxima de jugadores que está definida por un parámetro cuyo valor se encontrará **siempre** entre 2 y 4. Una vez que la cantidad de jugadores dentro de la sala llegue al máximo permitido según el parámetro, la partida iniciará. Los nombres de los jugadores dentro de esta sala no pueden estar repetidos, y cada nombre debe contener solamente caracteres alfanuméricos, sin espacios. Una vez que un cliente ingresa un nombre válido, inmediatamente ingresa a la sala y se convierte en un jugador de *DCCuatro*. En específico, las acciones de las que se debe encargar el servidor son:

- **Procesar y validar** las acciones realizadas por todos los participantes. Por ejemplo: Si se juega una carta que no es jugable, el servidor deberá avisarle al usuario que la jugada no es válida y se mantendrá el turno del jugador, hasta que realice una acción válida.
- Verificar que el **nombre de un nuevo jugador** sea válido y que no exista dentro de la partida, para evitar nombres repetidos.
- **Generar y contener** el mazo de cartas de las cuales robarán los distintos jugadores, es decir, cada vez que un usuario robe una carta, el servidor sacará la primera carta del mazo y procederá a enviársela al jugador correspondiente. Para generar el mazo, el servidor utilizará el archivo entregado `generador_de_mazos.py`.

- **Distribuir los cambios** en tiempo real a los jugadores conectados correspondientes, con el fin de mantener sus interfaces actualizadas y consistentes entre sí. Por ejemplo: cuando un usuario juega una carta exitosamente, todos los participantes deben ver que dicho usuario tiene una carta menos en su mano y además ver la carta que jugó.
- **Contener y distribuir** todas las *sprites* a los distintos usuarios, en el momento en que las soliciten. Por ejemplo: Al robar una carta, el cliente envía la solicitud al servidor, quien se encarga de robarla y enviarla específicamente a ese cliente para que la despliegue.

4.2.2. Cliente

Cada cliente contará con una interfaz gráfica que le permitirá realizar acciones, y a su vez, comunicarse con el servidor. Las acciones que maneja el cliente son las siguientes:

- **Ingresar a la aplicación**. Al comienzo del programa, el usuario puede ingresar un nombre y el cliente debe enviarlo al servidor para la comprobación de su validez.
- **Solicitar una o más cartas** al servidor. Cuando el usuario seleccione el botón para robar una o más cartas del mazo, se procederá a pedirlas al servidor, y posteriormente, esté le responderá con la información solicitada.
- **Seleccionar una carta**. Cuando el jugador seleccione una carta de su mano para ponerla en el pozo, debe notificar la carta al servidor, quien verificará si la jugada es válida o no. En caso de ser válida, la carta va al pozo. En caso contrario, se muestra un aviso de jugada no válida y se da la opción de intentar con otra jugada.
- **Gritar ¡*DCCuatro*!** El jugador debe poder gritar ¡*DCCuatro*! en cualquier momento de la partida, por lo que debe enviar la solicitud al servidor, quien verificará si el grito es válido o no, y posteriormente distribuir el efecto de dicha acción a los jugadores afectados.

5. Interfaz

Para una experiencia más amena, los jugadores de *DCCuatro* jugarán mediante una interfaz gráfica que guíe al usuario a lo largo del juego. Cabe destacar que todo lo descrito en esta sección hace referencia a una interfaz gráfica implementada solo para el cliente, haciendo uso de la librería `PyQt5`.

Los ejemplos expuestos en esta sección solo son para que tengas una idea de cómo podría verse cada ventana. Eres libre de modificar o diseñar las ventanas a tu gusto, siempre y cuando cumplan con los requerimientos mínimos.

Las ventanas a implementar son:

5.1. Ventana de Inicio

La ventana de inicio debe ser la primera en mostrarse al iniciar el juego. Esta debe contener como mínimo un campo de texto para que el jugador ingrese su **nombre de usuario** y un botón para **ingresar al juego**. Solo se debe permitir el ingreso a los usuarios que cumplan con las siguientes condiciones:

- El nombre debe contener solamente caracteres alfanuméricos, sin espacios.
- No debe haber otro usuario dentro del juego con el mismo nombre.
- Deben haber cupos disponibles en la sala.
- No debe haber una partida en progreso.

Figura 5: Ejemplo de la Ventana de Inicio.

5.2. Sala de espera

La sala de espera es la ventana que se muestra cuando un usuario validado logra ingresar al juego. Esta ventana tiene por objetivo reunir a todos los jugadores en sesión mientras esperan a que la partida comience. Esta sala debe mostrar en todo momento la **lista de los jugadores conectados en la sala**.

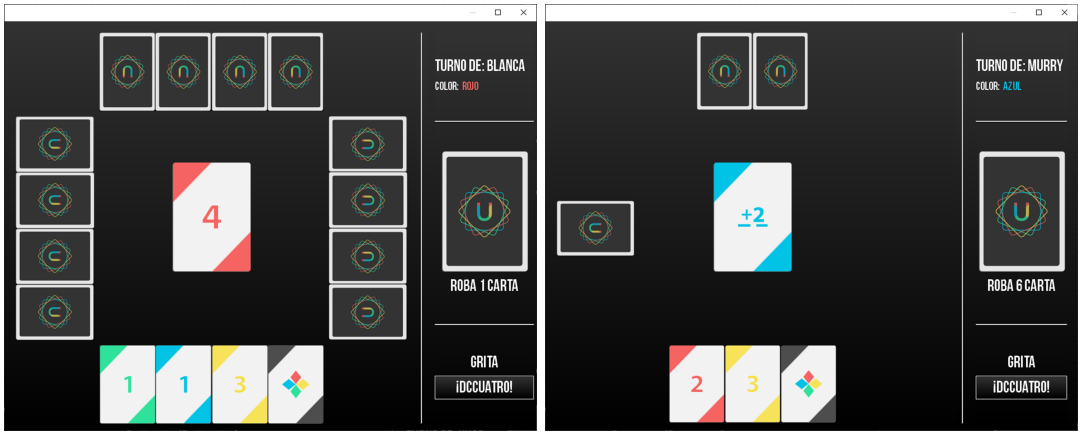
Figura 6: Ejemplo de la Sala de espera de 4 jugadores.

5.3. Sala de juego

Esta es la ventana donde se desarrollará el aclamado juego *DCCuatro*. En ella se deben visualizar como mínimo los siguientes elementos:

- Cartas propias, ubicadas boca arriba.
- Cartas de otros jugadores boca abajo, esto le permitirá al usuario ver cuántas cartas les quedan a sus adversarios.
- Pozo, el cual siempre debe mostrar la última carta jugada.
- Nombre de usuario del jugador cuyo turno está en progreso, y color actual en juego.
- Botón para sacar una carta del mazo.
- Botón para gritar ¡*DCCuatro*!
- Opción para cerrar la ventana[8].

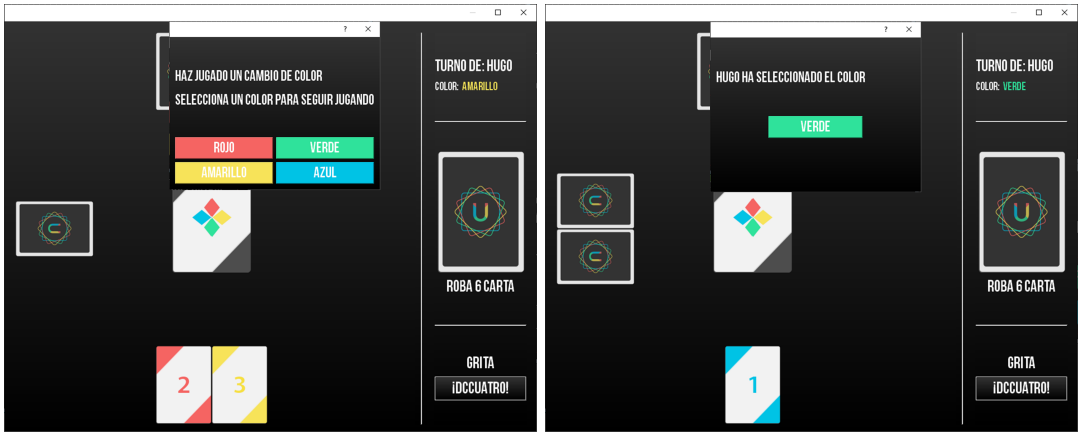
[8] Quedará a tu criterio implementar un botón que cierre la ventana o utilizar la opción "Cerrar" de la ventana.



(a) Partida de 4 jugadores (b) Partida de 3 jugadores

Figura 7: Ejemplo de la Sala de juego.

En caso de que se juegue la carta **Cambio de color** y sea una jugada válida, debe haber algún mecanismo que le permita al usuario seleccionar el color con el que se deberá seguir jugando, como un *pop-up*[9]. A su vez, se puede avisar al resto de los participantes el color seleccionado de alguna forma.



(a) *Pop-up* para seleccionar color (b) *Pop-up* para avisar color

Figura 8: Ejemplo de *pop-ups* al momento de jugar un Cambio de color.

Finalmente, en caso de que un jugador pierda, este se convierte en **espectador** mientras siga efectuándose la partida. En estos casos, se deben realizar la siguientes modificaciones a su **Sala de Juego**:

- El área donde se encuentran las cartas propias debe ser reemplazado por un *label* que indique su derrota. Esta modificación debe reflejarse también en las ventanas de los otros jugadores.
- Los botones de **Robar Cartas** y de **Gritar ¡*DCCuatro*!** deben inhabilitarse.

5.4. Fin de partida

Esta ventana o *pop-up* debe mostrarse una vez la partida llegue a su fin. A cada jugador deberá mostrarse una *label* indicando si perdió o ganó la partida, en conjunto con un botón que lo redirija a la **Ventana de Inicio.**

[9]Para implementarlo te recomendamos investigar sobre la clase `QDialog` de la librería `PyQt5`.

6. Archivos

Para desarrollar tu programa de manera correcta, debes crear y utilizar los siguientes archivos.

6.1. Archivos a crear

6.1.1. `parametros.json`

Similar a las tareas anteriores, a lo largo del enunciado se han ido presentando distintos números y palabras en `ESTE_FORMATO`, estos son conocidos como **parámetros** del programa y son valores que permanecerán constantes a lo largo de toda la ejecución de tu código.

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` contendrá solamente parámetros útiles para el cliente y parámetros de servidor contendrá solamente parámetros útiles para el servidor, tal como se explica en detalle en la sección *Networking*.

Los archivos deben estar en formato JSON, como se ilustra a continuación:

```
{
    "host": <dirección_ip>,
    "port": <puerto>,
    "jugadores": <valor_jugadores>,
    "path_1": <dirección_path_1>,
    ...
}
```

6.2. Archivos entregados

6.2.1. `generador_de_mazos.py`

Este archivo contiene la función `sacar_cartas(cantidad_cartas)`, que como dice su nombre, permite realizar la acción de sacar cartas del mazo. Para llamar esta función, deberás entregarle un número entero positivo, el cual indicará la cantidad de cartas que te retorne la función. El formato en el que retorna el mazo es una lista de la forma:

```
[("1", "rojo"), ("4", "azul"), ("+2", "verde"), ("color", "") ...]
```

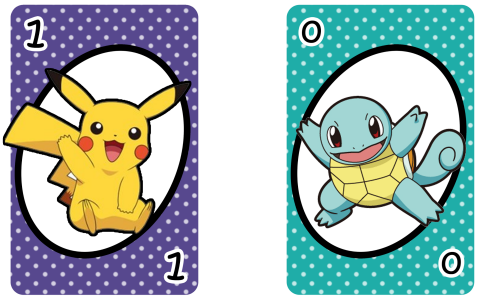
Es importante notar que **no debes cambiar este archivo, sino solamente utilizarlo.**

6.2.2. *Sprites*

La carpeta `sprites` contiene una gran variedad de mazos para utilizar en tu tarea. Cada mazo contendrá las cartas necesarias para jugar una partida de *DCCuatro* y los nombres de estas vendrán en el formato:

- `tipo_color`, en caso de ser una carta con color[10], donde tipo puede ser el número de la carta, `+2` o `sentido`.
- `tipo`, en caso de ser una carta sin un color asociado, donde tipo puede ser `color`.
- `reverso`, el cual será la vista boca abajo de las cartas

[10] Puede que el fondo de algunas cartas no coincida con el `color` que aparezca en su nombre, esto es simplemente para facilitar el uso de distintos tipos de mazos.



(a) Carta 1 verde[10] (b) Carta 0 azul

Figura 9: Cartas de *DCCuatro*.

Como podrás notar, existe una gran variedad de mazos, esto es para darte mayor libertad al momento de realizar tu tarea, ya que no es necesario que los utilices todos. Además, te invitamos a buscar tus propias *sprites* en caso de que quieras darle un toque personal a tu tarea.

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. La nota en tu tarea (sin bonus) debe ser **igual o superior a 4.0**[11].
2. 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 8 décimas. Deberás indicar en tu `README` si implementaste alguno de los bonus, y cuáles fueron implementados.

7.1. Tiempo límite (3 décimas)

A veces los jugadores toman un largo tiempo en decidir su próxima acción, lo que puede volver una partida un poco tediosa. Para solucionar esto, decides implementar turnos con **tiempo límite**. Cada jugador tiene un tiempo máximo de `TIEMPO_TURNO` para realizar su jugada cuando es su turno, y si este tiempo se agota, el turno del jugador se acaba automáticamente y se pasa al del siguiente.

Para implementar esto, deberás incluir en la **Sala de Juego** una cuenta regresiva que indique el tiempo restante del turno actual. Este contador debe como mínimo actualizarse para el jugador cuyo turno está en proceso, pero si así lo prefieres, también puede actualizarse para todos los jugadores de la partida y así todos pueden conocer el tiempo restante de sus contrincantes.

7.2. *Anonymous* (3 décimas)

Un poco de ayuda no le hace mal a nadie, y es por ello que decides implementar una nueva funcionalidad que hará las partidas más lúdicas.

Durante su propio turno, cada jugador tendrá **una vez por partida** la opción de ver todas las cartas en mano de todos los jugadores. Esto deberá activarse mediante un botón en la **Sala de Juego**, el cual se deshabilitará temporalmente mientras no sea su turno, y permanentemente una vez este haya sido usado. Gráficamente, deberás reemplazar los *sprites* de cartas boca abajo de cada oponente por su equivalente

[11] Esta nota es sin considerar posibles descuentos.

boca arriba. Una vez activada, esta habilidad durará hasta que se acabe el turno del jugador, momento en el cual las cartas de los otros competidores deben volver a ubicarse boca abajo.

7.3. Relámpago (2 décimas)

Como el *DCCuatro* es muy sencillo y le falta emoción, con tus amigos deciden inventar una nueva regla... **El Relámpago**. Ahora, cada vez que un jugador tenga exactamente la misma carta que la que se encuentra en la parte superior del pozo, y se trate de una **carta regular**, podrá jugarla aunque no sea su turno, saltándose así el turno de todos los otros jugadores anteriores a él.

Esta jugada puede provocar una descoordinación en los turnos que seguirán, por lo que deberás encargarte de reajustar los turnos según quien haya usado una carta relámpago.

7.4. Selector de mazos (2 décimas)

Cansado de jugar siempre con el mismo mazo, se te ocurre la fantástica idea de dar la opción de elegir uno de los diversos mazos disponibles antes de empezar a jugar una partida de *DCCuatro*.

Para esto, deberás integrar en la **Ventana de Inicio** un *dropdown selector*[12] el cual le permita elegir al usuario el tipo de mazo que utilizará en la partida.

El mazo que utilice cada usuario deberá ser independiente del que escojan los otros jugadores, por lo que únicamente deberá ver cartas del mazo que haya seleccionado.

7.5. *Forever alone* (4 décimas)

Para las veces donde deseas jugar una partida de *DCCuatro*, pero no hay nadie con quien jugar, existe la posibilidad de implementar este *bonus*.

En la **Ventana de Inicio** deberás incluir la opción de jugar una partida por tí mismo, siendo tus oponentes manejados por inteligencias artificiales altamente entrenadas[13]. Para comenzar una partida *forever alone*, no debe haber ninguna partida en progreso y la sala de espera debe estar vacía. Si esto se cumple, el jugador deberá pasar inmediatamente a la **Sala de Juego** y comenzar a jugar versus otros oponentes manejados por el servidor[14].

Los oponentes se comportarán siempre de la siguiente manera:

- Primero, revisan si dentro de las cartas de su mano hay alguna *jugable*. Si es así, juegan esa carta. En caso de que haya más de una carta *jugable*, queda bajo tu criterio el decidir cuál jugará.
- Si no hay ninguna carta jugable, robarán de forma automática.
- Si un oponente juega un **cambio de color**, elegirá el color al cual cambiar de forma aleatoria.
- Los oponentes son un poco tímidos, así que **nunca** gritarán ¡*DCCuatro*!

7.6. *Chat* (5 décimas)

La comunicación es esencial para cualquier juego, por lo que decides agregarle un *chat* a *DCCuatro* para poder hablar libremente con el resto de los jugadores. El *chat* debe mostrarse tanto en la Sala de Espera como en la Sala de Juego, y debe incluir como participantes solo a quienes estén en la sala de espera o jugando una partida.

[12] Para implementarlo te recomendamos utilizar la clase `QComboBox` de la librería `PyQt5`.
[13] A.K.A `random`.
[14] La cantidad de contrincante estará dado por la `CANTIDAD_JUGADORES_PARTIDA`, de forma que la Sala de Juego este llena.

El diseño del *chat* queda a tu libre criterio (siendo, por ejemplo, una ventana aparte que esté siempre abierta cuando se esté en las salas de espera y juego, o incorporado dentro de las mismas ventanas anteriores), siempre y cuando este sea intuitivo de usar. Cada mensaje de *chat* debe seguir el siguiente formato:

```
Usuario : Mensaje
```

Debes asegurarte de que los mensajes se muestren siguiendo el orden cronológico en que fueron enviados.

8. Avance de tarea

Para esta tarea, el avance corresponderá a implementar el inicio de interacción en *DCCuatro*, en tanto servidor como cliente.

Específicamente, se pide para cumplir con este avance entregar un servidor sea capaz de conectarse a múltiples clientes, que sea capaz de recibir nombres para ingresar a sala de espera, y les permita entrar. Por el lado del cliente, deben mostrar la Ventana de Inicio y la Sala de Espera, e interactuar con el servidor para poder ingresar a la última.

No es necesario implementar (para este avance) la revisión de nombres válidos antes de entrar a la Sala de Espera, ni una actualización en vivo de los usuarios dentro de ella.

A partir de los avances entregados, se les brindará un *feedback* general de lo que implementaron en sus programas y además, les permitirá optar por **hasta 2 décimas** adicionales en la nota final de su tarea.

9. `.gitignore`

Para esta tarea **deberás utilizar un `.gitignore`** para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `T03/`. Puedes encontrar un ejemplo de `.gitignore` en el siguiente ***link***.

Deberás ignorar el enunciado, los archivo entregados `generados_de_mazos.py` y **la carpeta llamada `sprites`**. Los archivos `parametros.json` **no deben ser ignorados. Deben ser entregados**.

En caso de que hagas uso de tus propias *sprites*, deberás guardarlas en otra carpeta y asegurarte que no sean ignoradas por el `.gitignore`.

Se espera que no se suban archivos autogenerados por programas como PyCharm, o los generados por 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.

10. 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.

11. 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.

Cuando se publique 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.

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

Se recomienda el uso de *Logs* del servidor para ver los estados del sistema para apoyar la corrección. De todas formas, ningún ítem de corrección se corrige puramente por consola, ya que esto no valida que un resultado sea correcto necesariamente. Para el cliente, todo debe verse reflejado en la interfaz, pero el uso de *logs* también es recomendado.

Finalmente, en caso de que no logres completar el Envío de las cartas, puedes enviar los mensajes mediante codificación directa, para que así no afecte otras funcionalidades de la tarea. Pero implicará que **no obtengas puntaje en los ítems relacionados**. Si lo llegases a implementar de esta forma, recuerda indicarlo en tu `README.md`.

12. 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 3.7.
- 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 48 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).