UNIVERSIDAD DE BURGOS

# MEMORIA DEL PROYECTO

Diseño y Mantenimiento del Software

Álvaro Delgado

Xinglong Ji

Alisson Romero

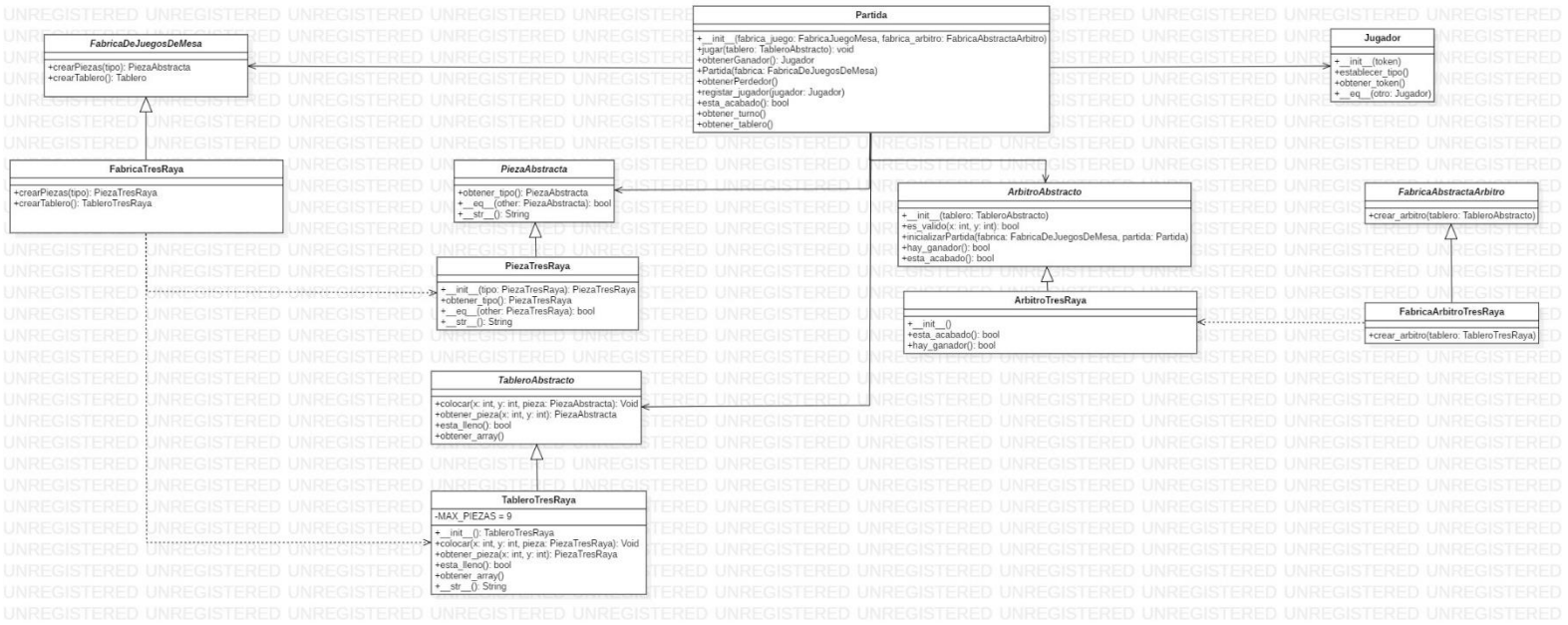
Sergio Santamaría

Zoe Calvo

# Contenido

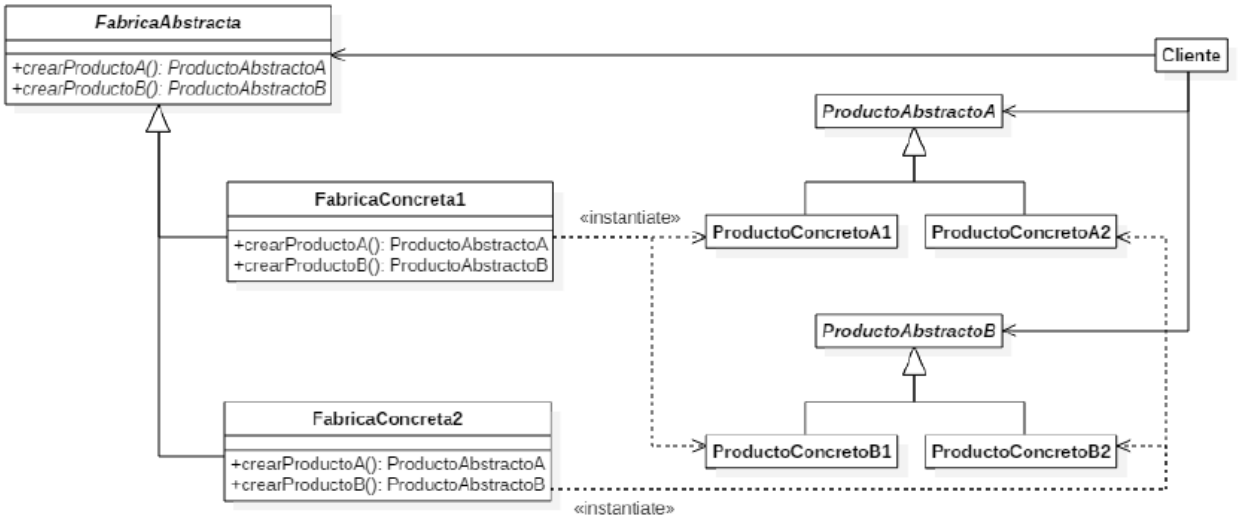
# Arquitectura y diseño del servidor

El juego ha sido diseñado mezclando dos patrones creacionales, la FabricaAbstracta y el Método Fábrica.




## Fábrica Abstracta

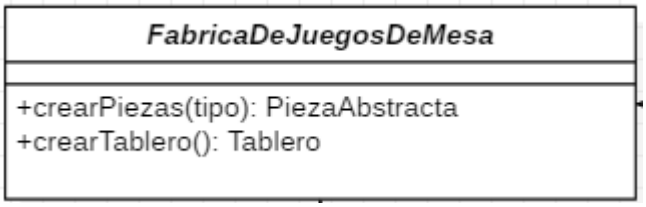
La intención de este patrón provee una interfaz para crear familias de objetos u objetos dependientes entre sí, sin necesidad de especificar sus clases concretas.



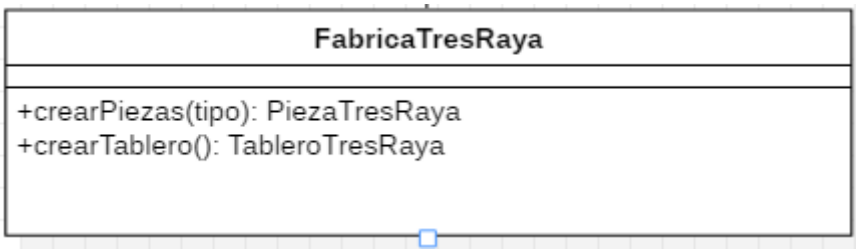
En nuestro modelo este patrón tiene como participantes:

## PARTICIPANTES

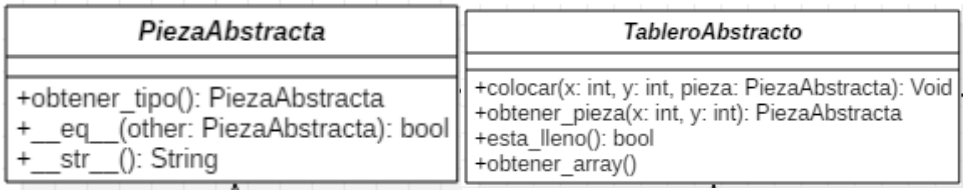
- **FabricaDeJuegosDeMesa.** Juega el papel de FabricaAbstracta que es una interfaz con operaciones que permiten crear productos, en nuestro caso distintos juegos.



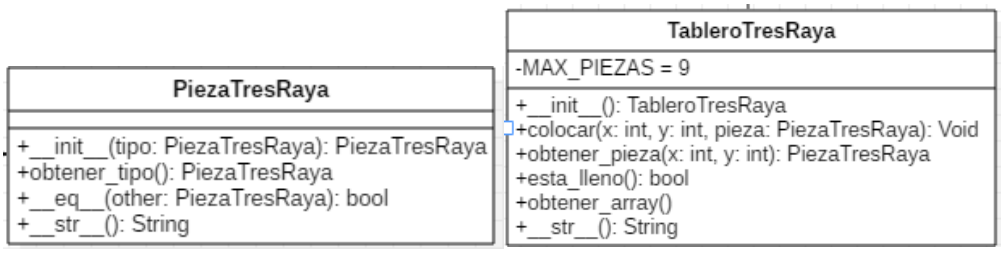

- **Fabrica3EnRaya.** Juega el papel de FabricaConcreta que implementa las operaciones para crear productos concretos, piezas, tablero y arbitro.



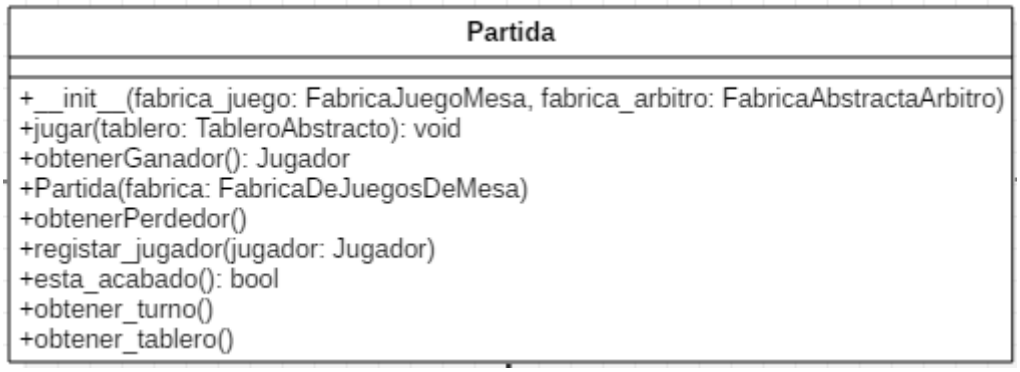

- **PiezaAbstracta, TableroAbstracto.** Los dos juegan el papel de ProductoAbstracto, son interfaces que crean objetos para los distintos juegos que implementemos.




- **PiezaTresRaya, TableroTresRaya.** Los dos juegan el papel de ProductoConcreto, definen un producto que será creado con la FabricaConcreta, en este caso con la Fabrica3EnRaya. Además, implementan las interfaces de ProductosAbstractos.



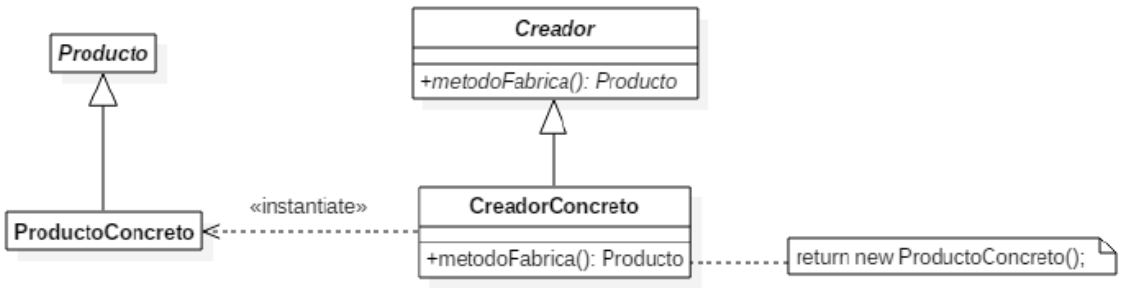
- **Partida.** Está pensado como el Cliente, se relaciona con las interfaces de FabricaAbstracta(FabricaDeJuegosDeMesa) y de los ProductosAbstractos (PiezaAbstracta y TableroAbstracto).




La Partida actúa como nexo entre la Fábrica Abstracta y el Método Fábrica además de ser la que se conecta con el Jugador.

## Método Fábrica

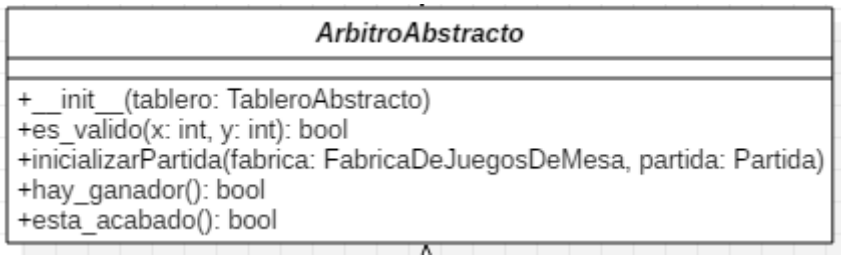
Tiene como intención definir una interfaz para crear un objeto, pero las subclases son las que deciden qué clase instanciar.



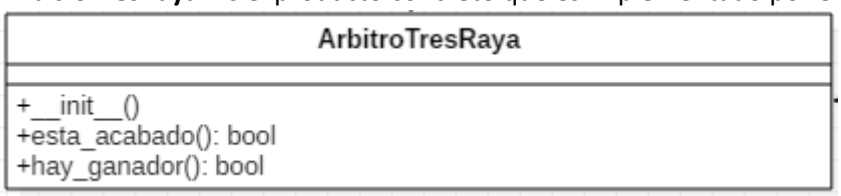

Hemos utilizado este patrón para crear una fábrica de árbitros.

### PARTICIPANTES

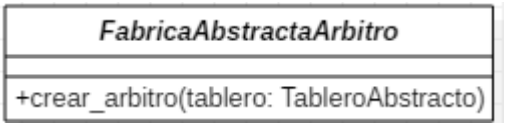
- **ArbitroAbstracto**. Será el producto, que es el que define la interfaz de los objetos que crea el método fábrica.



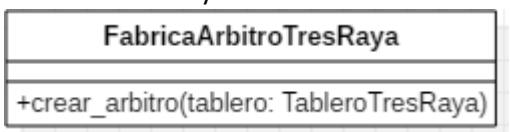

- **ArbitroTresRaya**. Es el producto concreto que es implementado por el Arbitro.



- **FabricaAbstractaArbitro**. Declara el método fábrica, que devuelve un objeto de tipo Producto (Arbitro).




- **FabricaArbitroTresRaya**. Implementa y sobreescribe la FabricaAbstractaArbitro para devolver una instancia de ArbitroTresRaya.




## Fachada

Por último, englobamos todo nuestro modelo en un patrón Fachada.

Tiene como intención dar una interfaz unificada de alto nivel para un conjunto de interfaces de un subsistema que lo hace más fácil de usar.

Este patrón está representado por la clase *RestApi*, la cual unifica todas las operaciones que son proporcionadas a través de la API REST.

```
from flask import Flask, escape, request, abort
from auth.auth_client import AuthClient

from juego.partida import Partida
from juego.jugador import Jugador
from tres_raya.fabrica_tres_raya import FabricaTresRaya
from tres_raya.fabrica_arbitro_tres_raya import FabricaArbitroTresRaya

import json

class RestApi:
    """ Clase fachada de la API REST
    ---
    Esta clase es una fachada con las operaciones proporcionadas a través de la API REST.
    """
```

En un primer momento habíamos pensado utilizar solamente el patrón Fábrica abstracta en el cual el árbitro era otro ProductoAbstracto con sus clases concretas.

El problema de esta implementación es que la función del árbitro es controlar que la partida está siendo válida y anteriormente se encargaba de hacer la función lógica de la partida como tal.

Por ello, decidimos independizar el árbitro de forma que se conecte con la Partida y sea ajeno a lo que ocurre dentro.

# Arquitectura y diseño del cliente

Para la arquitectura del cliente hemos hecho uso del patrón Singleton.

## Singleton

Tiene como intención garantizar que una clase tenga una y solo una instancia, y que haya un acceso global a la misma.

Como se puede ver en el código de la clase *AuthClient* tenemos un método estático *instance()* que comprueba que haya una única instancia de la clase.

.

```
@staticmethod
def instance():
    """ Singleton instance access method.
    ---
    Do NOT use the constructor. Use this method instead.

    Returns:
        The singleton instance of this class.
    """
    if (AuthClient.__instance is None):
        AuthClient.__instance = AuthClient()
    return AuthClient.__instance
```

También hacemos uso de este patrón en la clase *HubClient* con un método estático *instance()* que comprueba lo mismo que la anterior, que haya una única instancia de la clase.

```
@staticmethod
def instance():
    """ Singleton instance access method.
    ---
    Do NOT use the constructor. Use this method instead.

    Returns:
        The singleton instance of this class.
    """
    if (HubClient.__instance is None):
        HubClient.__instance = HubClient()
    return HubClient.__instance
```

# Documentación del protocolo de comunicaciones cliente-servidor

Para facilitar el acceso al juego hemos creado el siguiente servicio: *dms1920-game-server*

- **dms1920-game-server**: El punto de acceso al servidor de juego, escuchando en el puerto 6789 con un API REST (ver más abajo)

Los protocolos de comunicación son los siguientes:

## SERVICIO

**dms1920-game-server**

Es el servidor del juego.

## API REST

La comunicación con el servicio se realiza a través de un API REST:

- `/`: Verificación del estado del servidor. No realiza ninguna operación, pero permite conocer si el servidor está funcionando sin miedo a alterar su estado en modo alguno.
  - **Método**: `GET`
  - **Respuesta**:
    - `200`: El servidor está funcionando correctamente.
- `/juego/registrar`: Endpoint de registros de jugadores.
  - **Método**: `POST`
  - **Parámetros**:
    - `token`: El token de usuario.
  - **Respuesta**:
    - `200`: Se ha registrado correctamente.
    - `401`: El token dado es incorrecto.
    - `500`: La partida ya tiene 2 jugadores.
- `/juego/turno`: Endpoint de comprobacion del turno.
  - **Método**: `GET`
  - **Parámetros**:
    - `token`: El token de usuario.
  - **Respuesta**:
    - `200`: Devuelve **true** si el jugador tiene el turno, sino **false**.
    - `401`: El token dado es incorrecto.

- `/juego/jugada`: Endpoint de realizacion de jugadas.
    - **Método**: `POST`
    - **Parámetros**:
        - `token`: El token de usuario.
        - `x`: Coordenada x del tablero.
        - `y:` Coordenada y del tablero.
    - **Respuesta**:
        - `200`: Jugada correcta.
        - `400`: La jugada no es valida (coordenadas incorrectas o jugador sin turno).
        - `401`: El token dado es incorrecto.
- `/juego/tablero`: Endpoint del estado del tablero.
    - **Método**: `GET`
    - **Respuesta**:
        - `200`: El tablero codificado en JSON en el contenido de la respuesta.
- `/juego/acabado`: Endpoint que comprueba si la partida esta acabada.
    - **Método**: `GET`
    - **Respuesta**:
        - `200`: Devuelve **true** si la partida esta acabada, sino **false**.
- `/juego/resultado`**:** Endpoint informa del resultado de la partida.
    - **Método:** `GET`
    - **Parámetros**:
        - `token:` El token de usuario.
    - **Respuesta**:
        - `200`: Devuelve el resultado del jugador ('Ganador', 'Perdedor' o 'Empate').
        - `400`: La partida no esta acabada.
        - `401`: El token dado es incorrecto.
- `/juego/finalizar`: Endpoint que termina una partida e inicializa otra nueva.
    - **Método**: `POST`
    - **Parámetros:**
        - token: El token de usuario

- **Respuesta**:
  - `200`: Partida finalizada correctamente.
  - `401`: El token dado es incorrecto

## CONFIGURACIÓN

El servidor usa las siguientes variables de entorno para su configuración:

- `GAME`: Nombre del juego.
- `GAME_SERVER_PORT`: El puerto en el que publicará su API REST.
- `AUTH_SERVER_HOST`: El host en el que se encuentra el servidor de autenticación.
- `AUTH_SERVER_PORT`: El puerto en el que está publicado el API REST del servidor de autenticación.
- `HUB_SERVER_HOST`: El host en el que se encuentra el hub.
- `HUB_SERVER_PORT`: El puerto en el que está publicado el API REST del hub.

# Manual de uso

## CONSTRUCCIÓN Y CONTROL DE SERVICIOS

1. Abrimos un terminal y accedemos al directorio donde tengamos el proyecto a través del comando: *cd nombreDirectorio*

```
alisson@alisson-virtual-machine:~$ cd Escritorio/practica-dms-2019-2020-master/
alisson@alisson-virtual-machine:~/Escritorio/practica-dms-2019-2020-master$
```

2. Para poder construir las imágenes de los servicios utilizaremos Docker- compose, ejecutando el siguiente comando: *sudo docker-compose -f docker/config/base.yml build*

```
alisson@alisson-virtual-machine:~/Escritorio/practica-dms-2019-2020-master$ sudo
 docker-compose -f docker/config/base.yml build
[sudo] password for alisson:
```

Tras introducir la contraseña, nos enseña unos mensajes indicándonos que las imágenes se han construido con éxito.

```
Building base
Successfully built 45d37c58748a
Successfully tagged dms1920-base:latest
Building auth-server
Successfully built c246f052cc95
Successfully tagged dms1920-auth-server:latest
Building hub
Successfully built 7164bca736cb
Successfully tagged dms1920-hub:latest
Building game-server
Successfully built f8eb179b3473
Successfully tagged dms1920-game-server:latest
```

3. Una vez configurado el sistema levantaremos los servicios a través del comando: sudo docker-compose -f docker/config/*base.yml up -d*

```
alisson@alisson-virtual-machine:~/Escritorio/practica-dms-2019-2020-master$ sudo
 docker-compose -f docker/config/base.yml up -d
[sudo] password for alisson:
```

Tras introducir la contraseña, nos enseña unos mensajes indicándonos que los servicios han sido levantados.

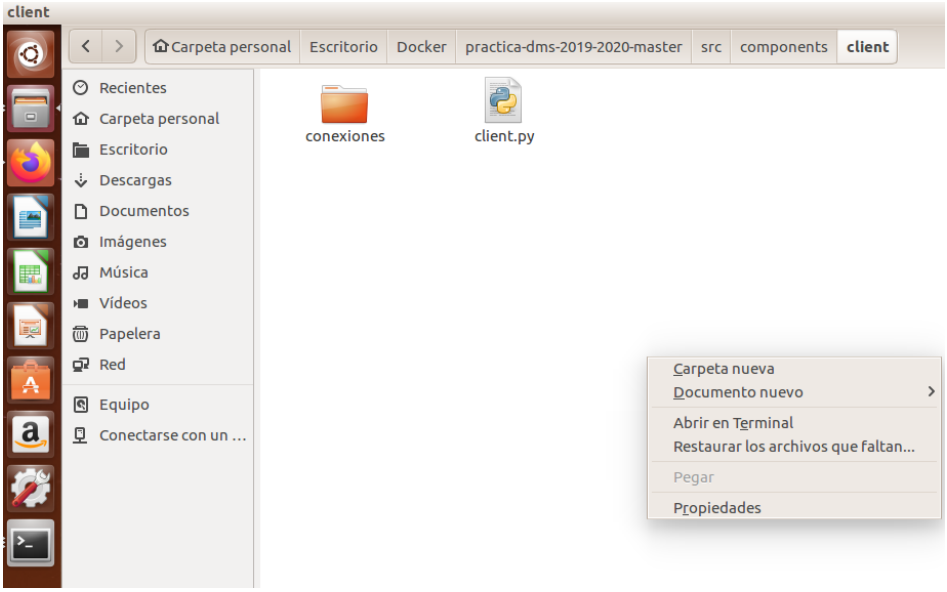
```
Recreating config_base_1 ... done
Recreating dms1920-auth-server ... done
Recreating dms1920-hub         ... done
Recreating dms1920-game-server ... done
```

## SERVICIOS

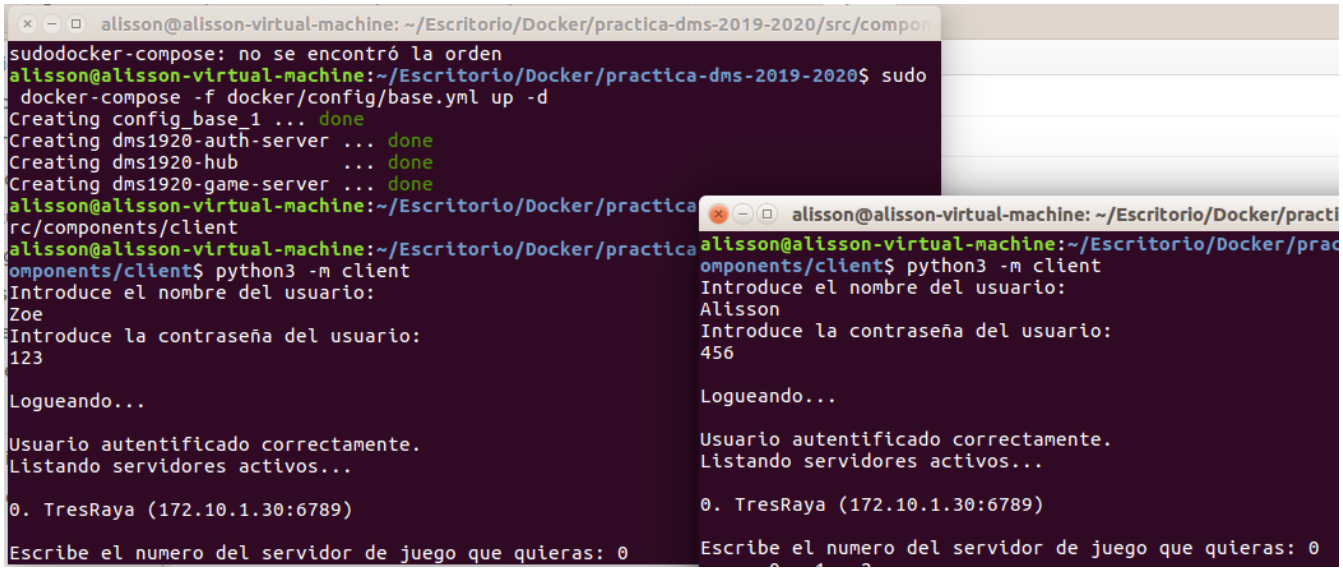
1. Para ejecutar nuestro cliente tenemos que acceder a su correspondiente directorio a través del comando: cd *nombreDirectorio*

```
alisson@alisson-virtual-machine:~/Escritorio/practica-dms-2019-2020-master$ cd s
rc/components/client
```

También se puede acceder si estas de forma gráfica en la carpeta pulsando botón derecho y seleccionando la opción “Abrir en terminal”.



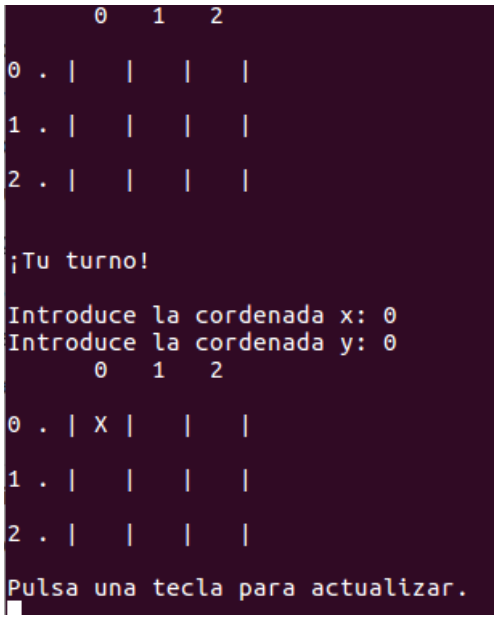

2. Es necesario abrir un terminal por cada jugador. Después ejecutamos el cliente con el comando*: python3 -m client.* El parámetro *-m* se utiliza para no tener que añadir la extensión *.py*




3. Tras la creación de los jugadores aparece un mensaje para poder escoger servidor. En nuestro caso, siempre es 0.

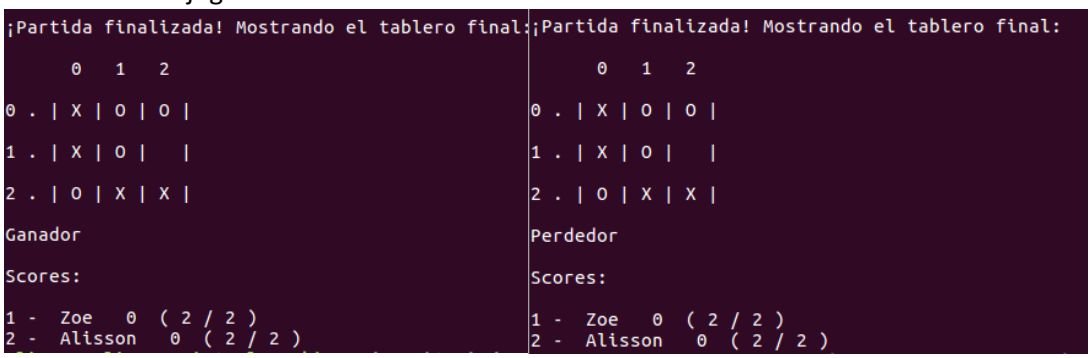
```
Escribe el numero del servidor de juego que quieras: 0
```

4. Ahora comenzamos a jugar.
    a. Lo primero será introducir la coordenada “x”, que será un valor entre 0 y 2.
    b. Después introducimos la coordenada “y”, que será un valor entre 0 y 2.

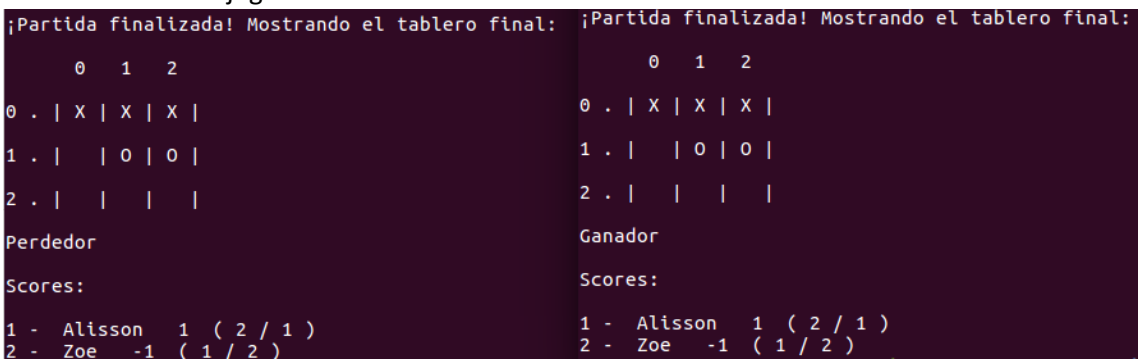



5. Final de juego
   Se muestra el tablero final y los resultados hasta el momento.
    - Entre paréntesis aparecen el número de partidas ganadas que lleva cada jugador.
    - El número que aparece a la derecha del nombre es un contador de *número de partidas ganadas – número de partidas perdidas*

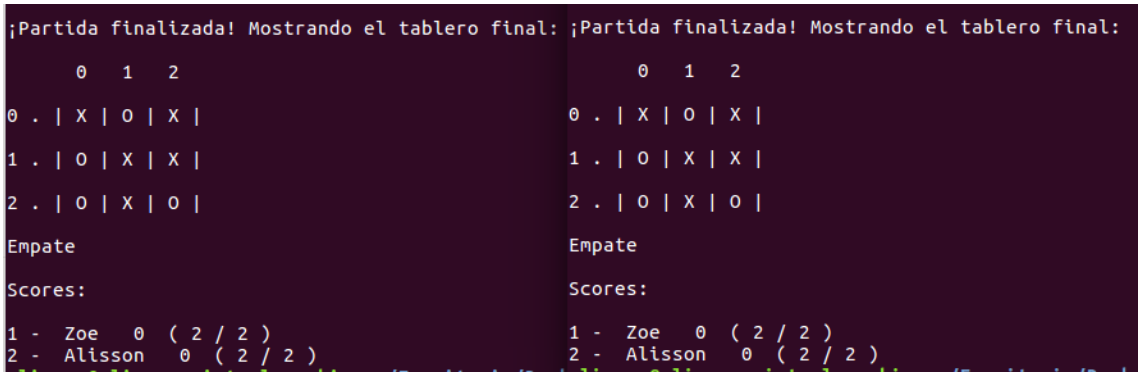
    a. Gana jugador 1



**b.** Gana jugador 2

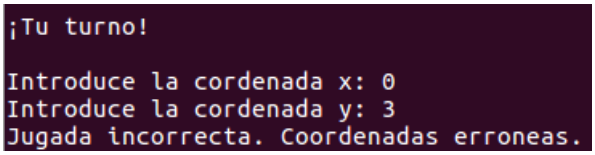



**c.** Empate

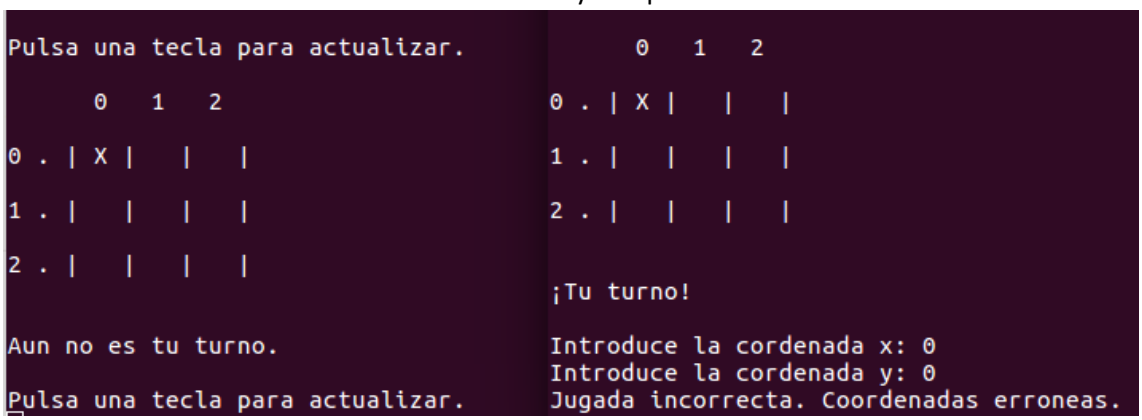



**6.** Control de errores

**a.** En caso de introducir una coordenada que no esté contemplada en el tablero.




**b.** En caso de introducir una coordenada donde hay una pieza.




**c.** Si el jugador que no tiene el turno intenta actualizar la partida, le saltará un mensaje por pantalla avisándole que todavía no puede mover.