

# Universidad ORT Uruguay

## **Obligatorio 1**

### **Programación de Redes**

Ivan Monjardin 239850

Agustina Disiot 221025

2021

# Índice

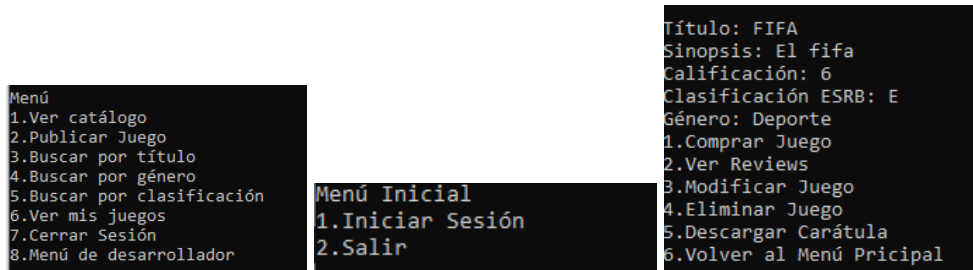
<b>Descripción de la arquitectura</b>	<b>2</b>
Funcionamiento de la aplicación	2
Comunicación entre servidor y cliente	3
<b>Clases del dominio</b>	<b>4</b>
Game	4
User	4
Review	4
Structs de Common	5
GamePage	5
ReviewPage	5
GameView	6
<b>Documentación de diseño</b>	<b>7</b>
Decisiones de diseño	7
Diagramas	9
Diagrama de Structs Pages	9
Diagrama de Clases Common	10
Diagrama de Clases Cliente	12
Diagrama de Clases Servidor	15
Algunos comandos, business logic y data access detallados	16
Diagrama de secuencia de un comando (Buy_Game)	18
Control de errores desde el Client	19
Validación	19
Control de errores desde el Server	20
<b>Documentación de mecanismos de comunicación</b>	<b>21</b>
Protocolo:Protocolo orientado a caracteres.	21
Header	21
Comandos	21
Largo	22
Datos	22
Comandos de texto	22
Pedidos Cliente	22
Respuesta del Servidor	23
Datos de archivo	25
<b>Errores encontrados</b>	<b>27</b>
<b>Posibles mejoras</b>	<b>27</b>

## *Descripción de la arquitectura*

### Funcionamiento de la aplicación

El cliente cuenta con una aplicación de consola con una interfaz a partir de menús, donde tiene que escribir el número de la opción que aparece en pantalla o ingresar algún dato que se le pide. En función de las opciones que elija se realizarán diferentes acciones, como comprar un juego, ver lista de juegos etc.

Cada menú puede llevar a un menú más específico y/o realizar una acción en el servidor. A modo de ejemplo, estos son algunos menús:



El cliente ya cuenta con los menús, no se los envía el servidor. Esto permitiría en un futuro cambiar el cliente sin tener que modificar el servidor.

Al abrir la aplicación por primera vez se le pide al usuario iniciar sesión escribiendo su nombre de usuario. Si no tenía una cuenta anteriormente, se crea una nueva. La aplicación permite hacer iniciar sesión y cerrar sesión para entrar con otra cuenta o salir de la aplicación (cerrando la conexión con el servidor).

También, se puede cerrar la aplicación del lado del servidor, escribiendo “exit” en la consola. Se cierran todas las conexiones a los clientes antes de cerrar el servidor. Así evitamos que el cliente siga mandando o esperando por datos del servidor.

Antes de ejecutar la aplicación modificar el app.config del servidor y cliente. Continúen la ip/puerto de ambos y la carpeta donde se guardan las portadas del lado del servidor.

## Comunicación entre servidor y cliente

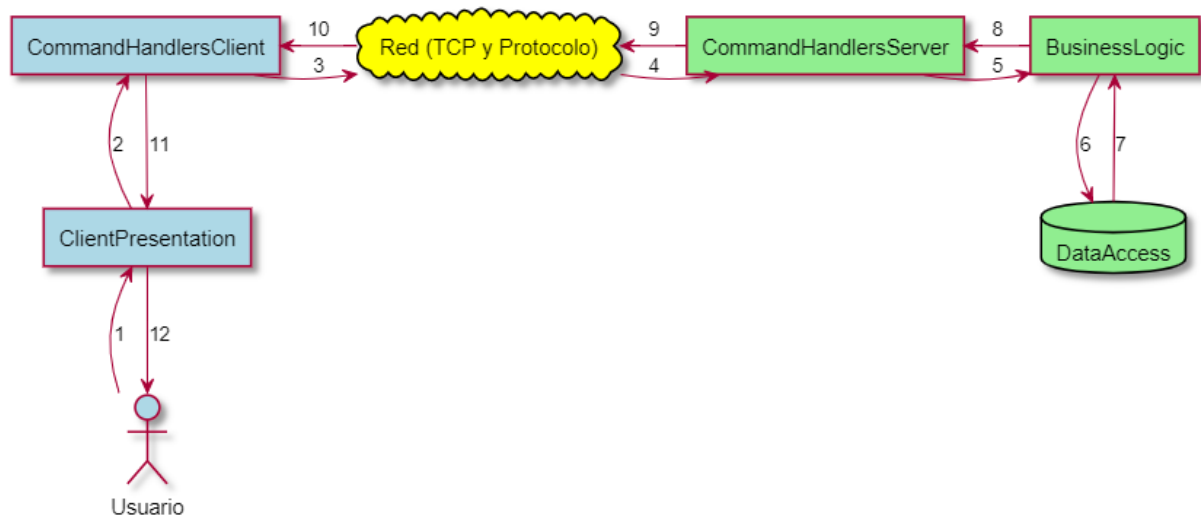
Nuestra aplicación soporta uno o varios clientes simultáneos que se pueden comunicar con un único servidor mediante el protocolo TCP y un protocolo de aplicación que diseñamos nosotros y que se explicará en detalle más adelante.

Lo más importante de nuestro protocolo es que cuenta con diferentes “comandos” cada uno el cual se encarga de una acción. Por ejemplo, comprar un juego, publicar juego, buscar juego por género, etc.

Por cada comando existe una clase correspondiente tanto en el cliente como en el servidor, pero con diferente lógica, una que hace los pedidos de información y otra que responde respectivamente.

Los comandos del lado del cliente y del servidor se comunican utilizando el stream TCP mencionado anteriormente.

Una arquitectura de alto nivel sería la siguiente:



Este diagrama solo representa el orden de los pasos desde que el cliente hace un pedido hasta que recibe la respuesta. En azul se muestra el lado del cliente, en verde el servidor y en amarillo el stream que se utiliza para comunicarse uno con el otro.

Más adelante vamos a explicar en detalle el protocolo y arquitectura de la comunicación entre el cliente y el servidor. Esto era una introducción para que se entiendan algunas decisiones de diseño.

## Clases del dominio

### Game

**genres** = { "Acción", "Aventura", "Juego de Rol", "Estrategia", "Deporte", "Carreras", "Otros" };

Se definió una lista fija de géneros accesible tanto por el servidor como el cliente para verificar que el género de un juego sea válido.

**Id** : Número entero, aumenta desde 0 en forma consecutiva.

Único de cada juego. Cuando se elimina un juego el id próximo no se modifica. Esto facilita a la hora de, por ejemplo, modificarse un juego, aunque un usuario esté viendo una versión vieja del juego, si le da ver reseñas, se mostrarán las reseñas correspondientes. Ya que al servidor se envía la id del juego y no el título (el cual puede ser modificado)

**Title**: Título del juego.

**Synopsis**: Descripción del juego en una línea sin límite.

**ReviewsRatings**: Número entero entre 1 y 10 que se calcula a partir de todas las calificaciones del juego publicadas por los usuarios. Si no hay reviews en el cliente se muestra un “-”

**CoverFilePath**: path de la portada del juego. Se genera el nombre del path con un GUID y están guardadas en una carpeta del lado del servidor

**ESRBRating**: Un ESRBRating perteneciente al Enum ESRBRating.

**Genre**: Género del juego. Tiene que pertenecer al array “genres”

**Publisher**: Usuario que publicó el juego. Cuando se publica el juego el publisher será el usuario que está logueado en esa conexión con el servidor.

**Reviews**: Lista de reseñas, con calificación numérica del 1 al 10 y un texto, hechas por usuarios que tienen el juego adquirido.

### User

**Name**: Nombre de usuario.

**GamesOwned**: Lista de los juegos que el usuario adquirió.

- El nombre de usuario es único pero se puede conectar el mismo usuario al mismo tiempo pero desde dos consolas diferentes. Si compra un juego en un cliente, se va a sincronizar con el otro cliente, al igual que con el resto de operaciones. No nos pareció necesario limitar al usuario a que solo pueda tener una sesión abierta a la vez.
- Si se modifica el juego, el juego en la lista de GamesOwned se queda actualizado.

### Review

**Text**: Comentario realizado por el usuario sobre el juego.

**Author**: Usuario que escribe la reseña.

**Rating**: Calificación numérica del juego del 1 al 10.

## Structs de Common

### GamePage

El struct GamePage se utiliza para los datos que decidimos mostrar por página en lugar de imprimir todos por consola, así existe un límite de la cantidad de juegos que se muestran. Este struct es utilizado tanto para el catálogo completo como para cada una de las tres opciones de búsqueda de juegos.

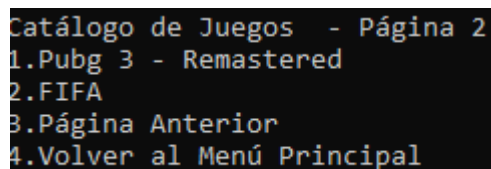
**GameTitles:** Lista con los títulos de los juegos. Estos pueden ser todos los juegos del servidor, o filtrados por título, género o rating, o todos los juegos del usuario logueado.

**GameId:** Lista de los ids de los juegos correspondiendo el orden con la lista de títulos.

**HasNextPage:** booleano indicando si existe siguiente página del listado de juegos.

**HasPreviousPage:** booleano indicando si existe página anterior.

**CurrentPage:** Número entero indicando que página del listado se está viendo actualmente.



```
Catálogo de Juegos - Página 2
1.Pubg 3 - Remastered
2.FIFA
3.Página Anterior
4.Volver al Menú Principal
```

### ReviewPage

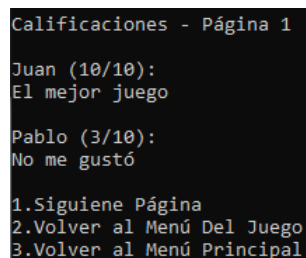
El struct ReviewPage se utiliza para mostrar las reviews existentes de un juego que se muestran por página. Al igual que en GamePage, se definió un límite a la cantidad de reviews por página (la cantidad que se le muestran al usuario a la vez). El límite es el mismo para ReviewPage y GamePage, está definido como una constante en el código, pudiendo cambiar si se quiere.

**Reviews:** Lista de reviews de un juego.

**HasNextPage:** booleano indicando si existe siguiente página del listado de juegos.

**HasPreviousPage:** booleano indicando si existe página anterior.

**CurrentPage:** Número entero indicando que página del listado se está viendo actualmente.



```
Calificaciones - Página 1
Juan (10/10):
El mejor juego
Pablo (3/10):
No me gustó
1.Sigue Página
2.Volver al Menú Del Juego
3.Volver al Menú Principal
```

## GameView

El struct GameView se utiliza para el menú de un juego y las opciones disponibles de ese juego. La opción de comprar solo se muestra si el usuario no tiene el juego, si el usuario tiene el juego en su lugar se mostrará la opción de escribir una review del juego. Si el usuario es quien publicó el juego entonces también tendrá la opción de modificar o borrar el juego. El resto de las opciones se muestran para todos los usuarios, como por ejemplo “descargar carátula”.

**Game:** Juego del que quiero ver la información.

**IsOwned:** booleano que representa si el usuario que está logueado tiene el juego adquirido.

**IsPublisher:** booleano que representa si el usuario que está logueado es quien publicó el juego.

## *Documentación de diseño*

### Decisiones de diseño

- Se utiliza un GUID (string aleatorio) para el nombre de los archivos de la portada del lado del servidor. Esto garantiza que los nombres no se repitan. Las chances de que se repitan los nombres sucede cuando se tienen demasiados archivos descargados al punto de que me quedo sin espacio antes que tener la chance que se repita un mismo GUID.
- Por la implementación, se aceptan solo archivos “.jpg” para la portada del juego. Preferimos solo aceptar imágenes, y no cualquier archivo. Primero, para no tener que mandar el nombre del archivo al servidor, lo cual no es algo que le interese al Server, en todo caso se podría mandar solo la extensión. Optamos por dejar una extensión fija ya que consideramos que sería lo más común para un servidor considerando el dominio del problema planteado, seria esperable que todas las portadas tengan el mismo formato.

Para aceptar otros formatos deberíamos pedirle que mande la extensión y tener una lista de las extensión válidas. Decidimos mandar solo el archivo y verificar que sea “.jpg”.
- Las imágenes se guardan en la carpeta “C:/GameCovers” la cual se puede modificar en el App Config del servidor.
- En los Ap Config tanto del cliente como el servidor también se pueden modificar las ip y puertos de cada uno.
- Al descargar una portada, en el Client se le pide la carpeta donde quiere guardarlo y el nombre con el que quiere descargar el archivo, sin la extensión.
- No hay un “crear usuario” separado del login. En el login se crea el usuario por primera vez o se busca en la capa de acceso a datos si ya existía.
- Se definieron nuevas funciones en INetworkStreamHanlder, tanto específicas al dominio del problema ( ej, ReadCommand) como más generales pero que simplifican el resto del código (ej. WriteString)
- Se utilizaron locks sobre las listas de juegos, usuario y conexiones para asegurar que cualquier elemento dentro de ellas no sea modificado mientras está siendo usando baja mutua exclusión por otro hilo.
- Utilizamos los locks en Business Logic y no en Data Access. Entendemos que en la práctica, es más probable que los mecánicos de mutua exclusión, como los locks se encuentren más en la capa de acceso a datos que en la lógica de negocios. Por ejemplo, una base de datos es la encargada de la concurrencia de las operaciones. Sin embargo, tomamos la decisión de utilizar como mecanismos las primitivas lock y eso llevó a que no tengamos tanta flexibilidad. Muchas operaciones, si se realiza el lock en el Data Access no se podía conservar en la lógica. La otra opción era poner más lógica en el Data Access, pero decidimos hacer una separación clara de las capas y darle prioridad a dejar la lógica en el BusinessLogic.



- Dentro del código de `NetworkStreamHandler` se puede activar la opción de debugging con un booleano. Con esto activado se muestra en el cliente y el servidor toda la información del request/response correspondiente (header, cmd, length, data).
- Para permitir mandar caracteres que ocupen más de un byte (ej. ñ), se calcula el largo de los datos en función de su conversión de bytes y no del largo de la string

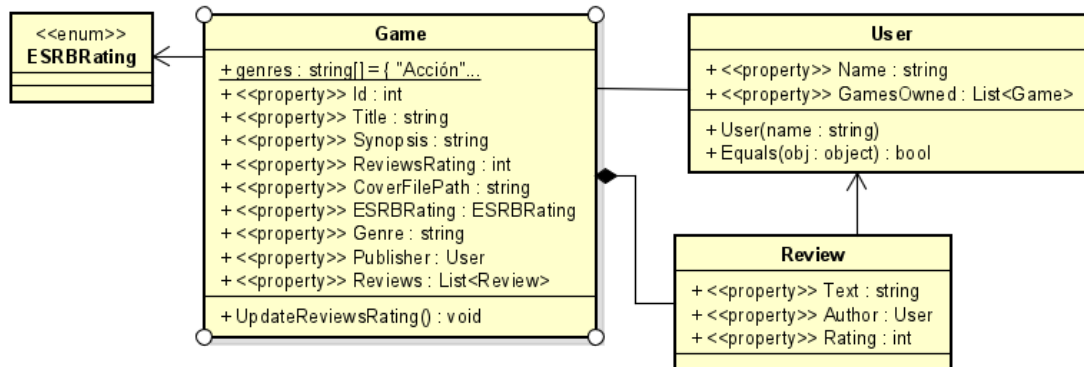
```
protected void SendData(string data)
{
    int dataLengthInBytes = Encoding.UTF8.GetBytes(data).Length;
    networkStreamHandler.WriteInt(dataLengthInBytes);
    networkStreamHandler.WriteString(data);
}
```

- Se agregó un Menú desarrollador con opciones que pueden ser útiles a la hora de probar la aplicación. Incluye carga de datos de prueba (juegos y reseñas) y mandar un dato invalido algo que en general no se puede hacer con nuestro cliente. Esto es para probar el Server Error que hablaremos más adelante.
- Se optó por que el usuario elija por que parametro poder filtrar y no que pueda buscar por título, calificación y género a la vez. Tanto a nivel de cliente como de servidor, los 3 comandos (4 si incluimos ver catálogo) utilizan abstracciones (ej. herencias) para no repetir código y llegar a una solución extensible por si en el futuro se quiere agregar otro filtro posible. Consideramos que esta opción simplifica la interfaz de usuario y hace el código más sencillo y mantenible.
- Hay 2 caracteres que no son aceptadas como entradas para la información que ingrese el usuario. Son “~” y “[”, ambos son usados como delimitadores en nuestro protocolo. Si en un futuro se quisiera utilizar como por ejemplo incluir en el título de algún juego, se pueden modificar en la especificación del protocolo (Specification) siempre y cuando no existan ya juegos con los nuevos delimitadores. Se buscó caracteres que no sean muy usados pero que no sean tan difíciles de escribir como para poder explicar el protocolo por escrito si fuera necesario.
- Siempre se intentó mandar solo la información necesaria entre el cliente y el servidor. Como decíamos antes, no se manda el nombre del archivo al servidor, tampoco se manda la página actual al pedir la siguiente `GamePage` entre otros.
- Se incluyen imágenes para pruebas y la defensa en la entrega.

## Diagramas

Se adjuntan los diagramas en una carpeta en documentación.

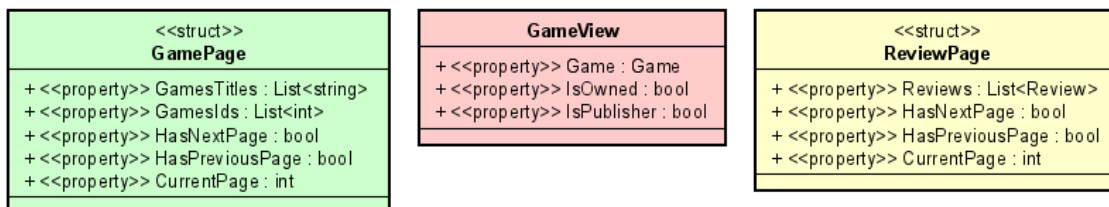
Clases del Dominio Common: (ClasesDelDominioUML.svg)



Las reviews son parte de un juego, no se conservan las reviews si el juego es eliminado, aunque el usuario que la publicó siga guardado.

## Diagrama de Structs Pages

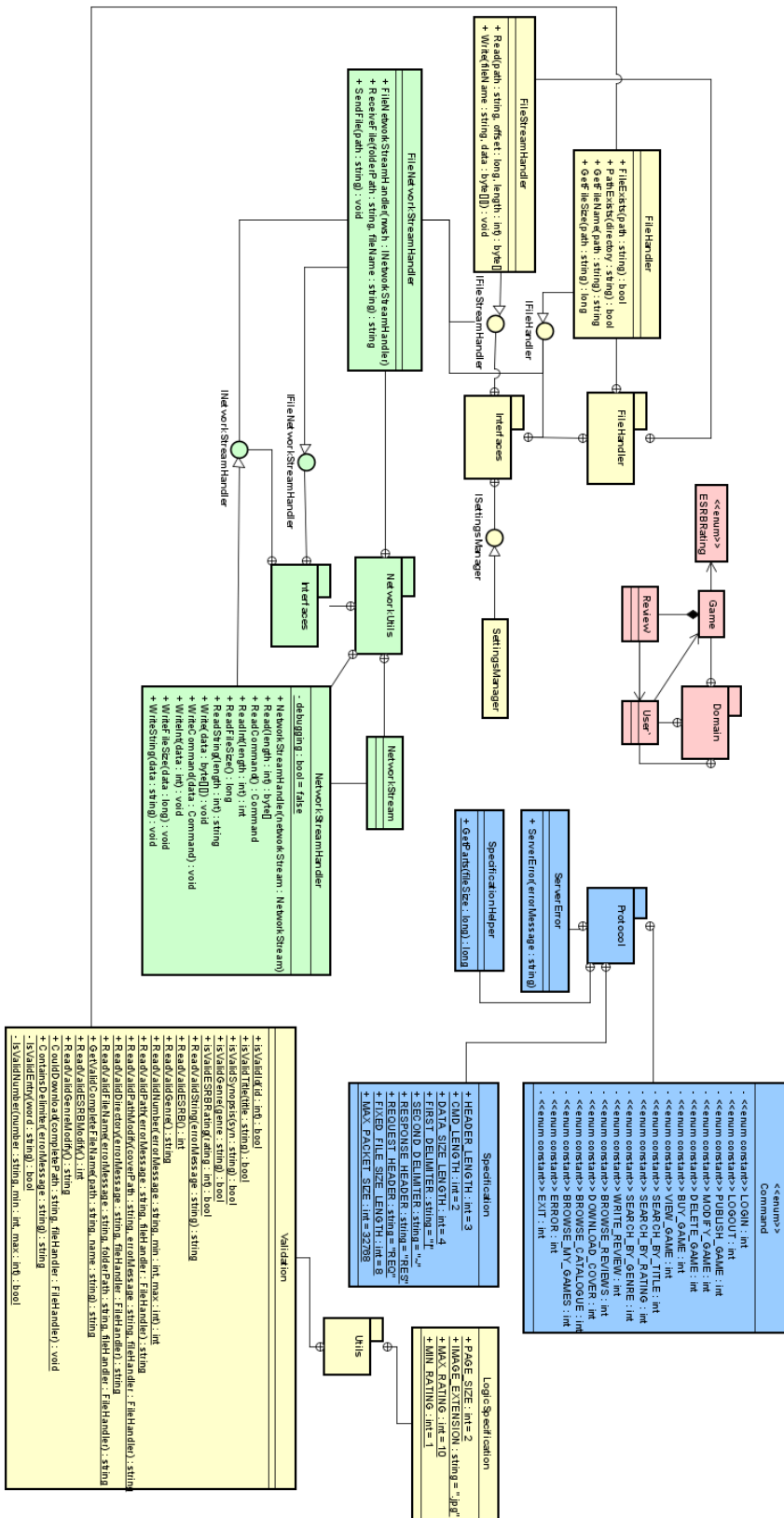
(StructPages.svg)



Se utilizaron booleanos para que el servidor le pueda indicar si ciertas opciones deberían mostrarse al cliente o no.

Se utilizaron listas para mandar los datos del servidor, así facilita mostrarlos al cliente mediante `CliMenu`, que tiene el método de mostrar el menú y sus opciones a partir de un diccionario, usado por `ClientPresentation`.

## Diagrama de Clases Common (CommonUML.svg)

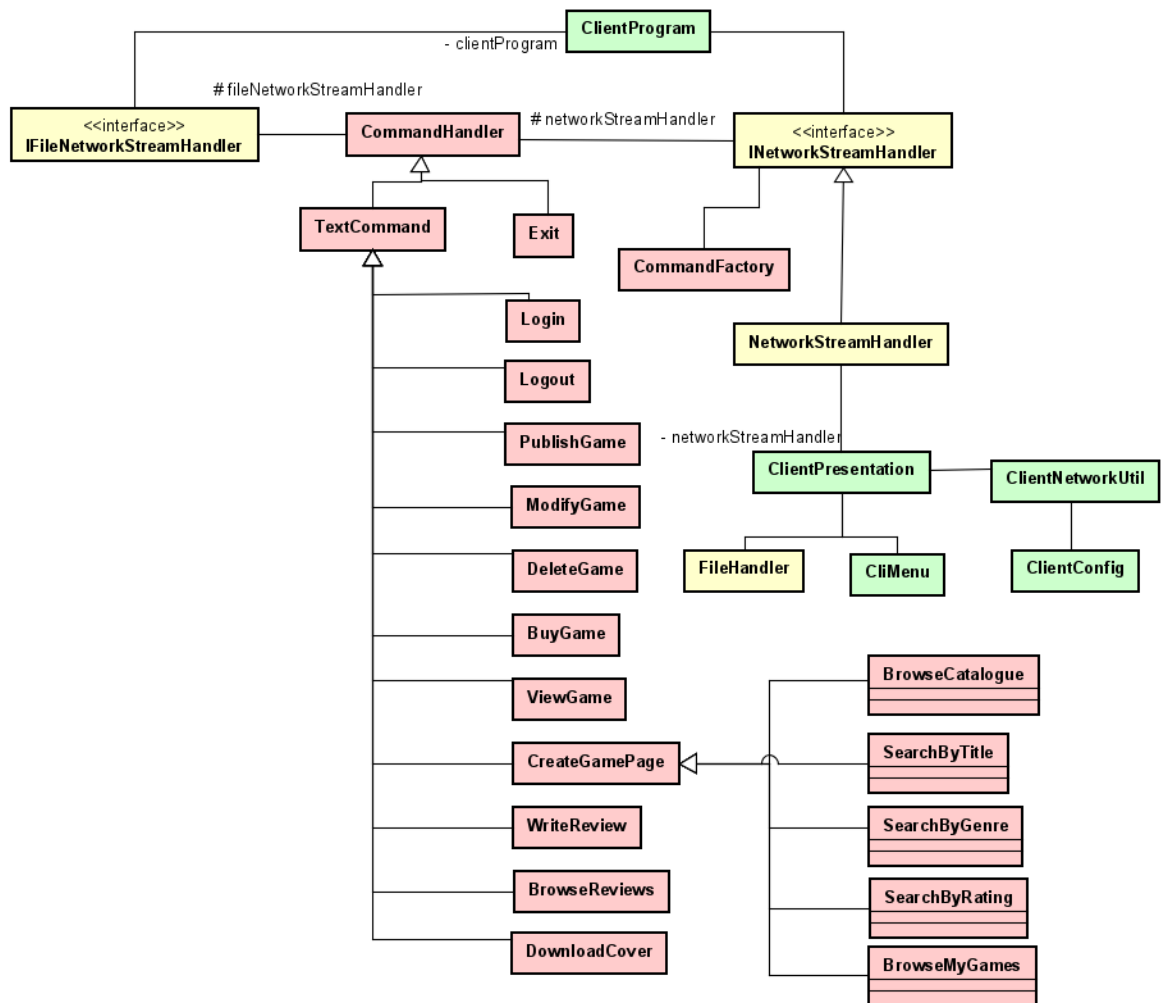


Estos archivos son accedidos tanto por el servidor como por el cliente. Se utilizó para no repetir código y para tener consistencia entre ambas partes.

Se incluyen clases como:

- Validation, ya que la validación se hace en ambos lados
- Command, ya que la lista de comandos es compartida, el servidor debe poder leer cualquier comando que mande el cliente y el cliente debe saber que comandos puede mandar
- Especificaciones generales del protocolo
- Otras especificaciones pero del problema en sí, como la extensión de la imagen que se acepta, el rating, el tamaño de página (de la GamePage y la ReviewPage)
- ServerError, el cual es mandado por el servidor y “catcheado por el cliente” (entre medio está el networkStream handler y los CommandHandler que lo mandan del servidor al cliente)
- También están las clases que ofrecen funciones para administrar los archivos y las conexiones entre cliente y servidor de una manera más sencillo
  - FileHandler: Se encarga de validar, escribir y leer archivos del filesystem del cliente y del servidor
  - NetoworkStreamHandler: Se encarga de enviar datos del cliente al servidor y viceversa
  - FileNetowrkStreamhandler: Se encarga de enviar archivos del cliente al servidor y viceversa

## Diagrama de Clases Cliente (ClienteUML.svg)



Se incluyen clases como:

- **ClientPresentation**: Se encarga de la interfaz “gráfica” del usuario, de los pedidos de datos al usuario y enviarlos a la capa de **CommandHandlers**
- **CliMenu**: Utilizado por el **ClientPresentation** para imprimir en pantalla menús donde el usuario solo precise escribir un número para elegir la opción que quiere. Se utilizan diccionarios <Nombre de la opción, Función del código que se va a llamar cuando se elija esa opción> . Ejemplo:

```

Menú
1.Ver catálogo
2.Publicar Juego
3.Buscar por título
4.Buscar por género
5.Buscar por clasificación
6.Ver mis juegos
7.Cerrar Sesión
8.Menú de desarrollador
    
```

```

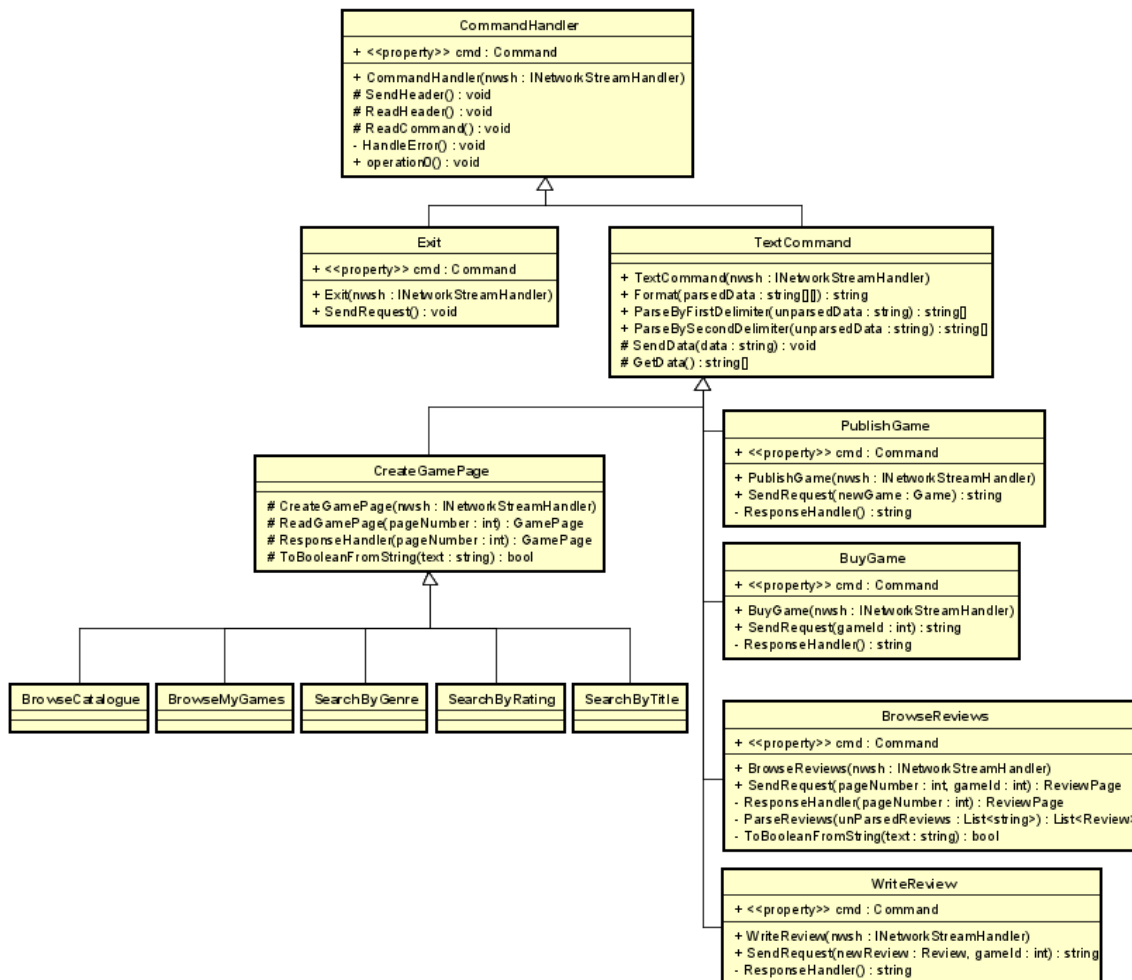
7.Cerrar Sesión
8.Menú de desarrollador
9
Eliga un número entre 1 y 8
    
```

Cuenta con validación para que el usuario escriba un número correcto.

- ClientConfig: Utilizado para leer los datos como ip/puerto del cliente/servidor para realizar la conexión
- Los comandos del cliente que se encarga de “parsear” los datos que escribe el cliente para que cumplan con el protocolo y después mandarlos al servidor. Cuando el servidor responde, los comandos pasan la respuesta de el formato del protocolo a una clase de dominio/valor esperado por el ClientPresentation
- CommandFactory: Utilizado para facilitar la creación de una instancia de un comando solo teniendo el Command.

Algunos comandos detallados/ampliados del cliente

(CommandsClienteDetalleUML.svg)

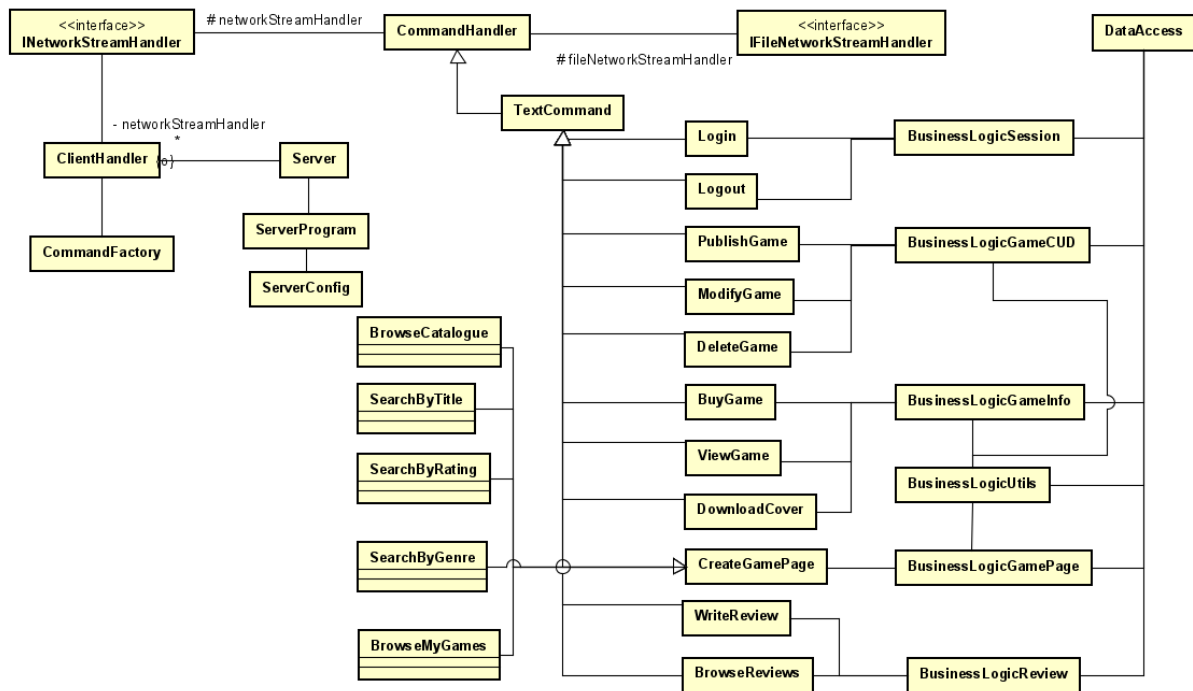


- Todos los comandos heredan de text command, menos Exit ya que exit no envía datos de texto al servidor.
- Exit solo lo envía el cliente al servidor, no el servidor al cliente

- CreateGamePage utiliza el struct GamePage mencionado anteriormente. Se encarga de agarrar la string enviada por el servidor y convertirla en GamePage para poder mandar al ClientPresentation

## Diagrama de Clases Servidor

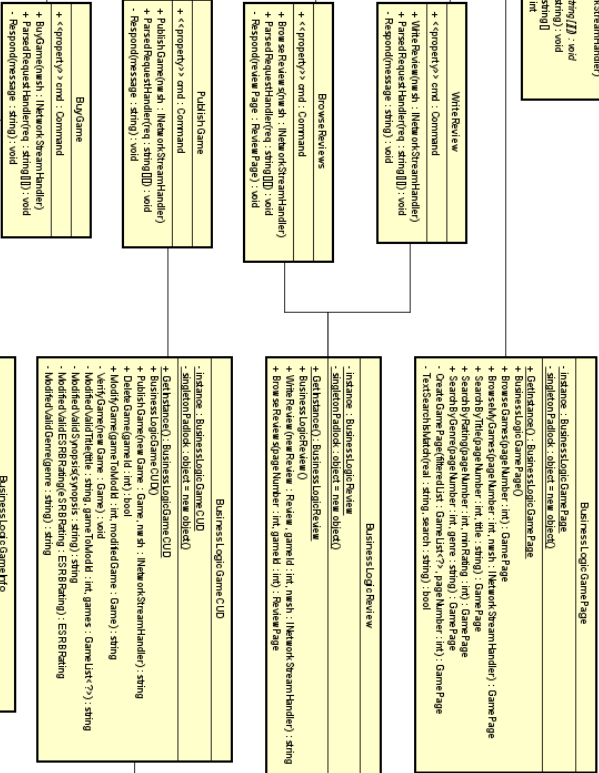
(ServerUML.svg)



- Al igual que en cliente existe `CommandFactory`, así no tenemos que tener un switch de todos los comandos dentro de otra clase
- También hay un comando por acción, solo que estos reciben la información del cliente se la mandan a la businesslogic, reciben la respuesta y se la mandan al cliente
- **BusinessLogic:** La lógica de negocio, se separa en varios archivos para dividir responsabilidades. Por ejemplo, la creación, modificación y baja de juegos de esta junta (CUD). Todas comparten la misma instancia de acceso a datos utilizando un singleton
- El `Data Access` cuenta con las listas como `games`, `users` y lleva la cuenta de cuál es la próxima `gameId` disponible. Además se encuentra la lista de conexiones activas y a que usuario logueado corresponden, se utiliza esa lista para verificar que operaciones puede hacer cada usuario.
- `ClientHandler` es el encargado de administrar una conexión/cliente mediante un tcp stream compartido entre ambas partes. Cuando se cierra el servidor, se cierran todos los `clientHandler` antes. Además cada cliente handler corre en un thread a parte para que el servidor puede tener más de un cliente conectado a la vez y aceptar/responder las request de todos los clientes en simultaneo.



**(CommandsServerDetalleUML.svg)**

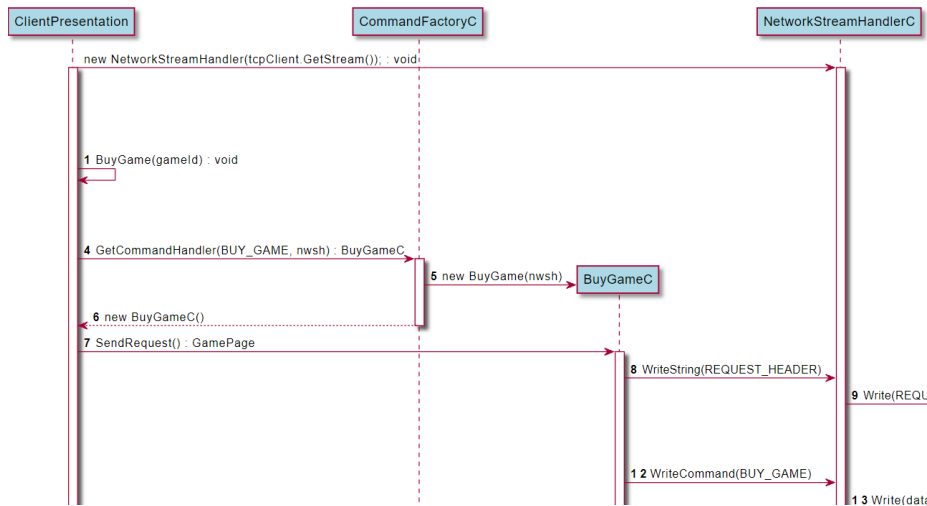


- Así como en el cliente se tiene que transformar del protocolo al GamePage, en el servidor se transforma la GamePage conseguida de la businessLogic a una string del protocolo.
- Cada business logic cuenta con su propio singleton, todos los cuales cuentan con un padlock para evitar que se crean más de una.

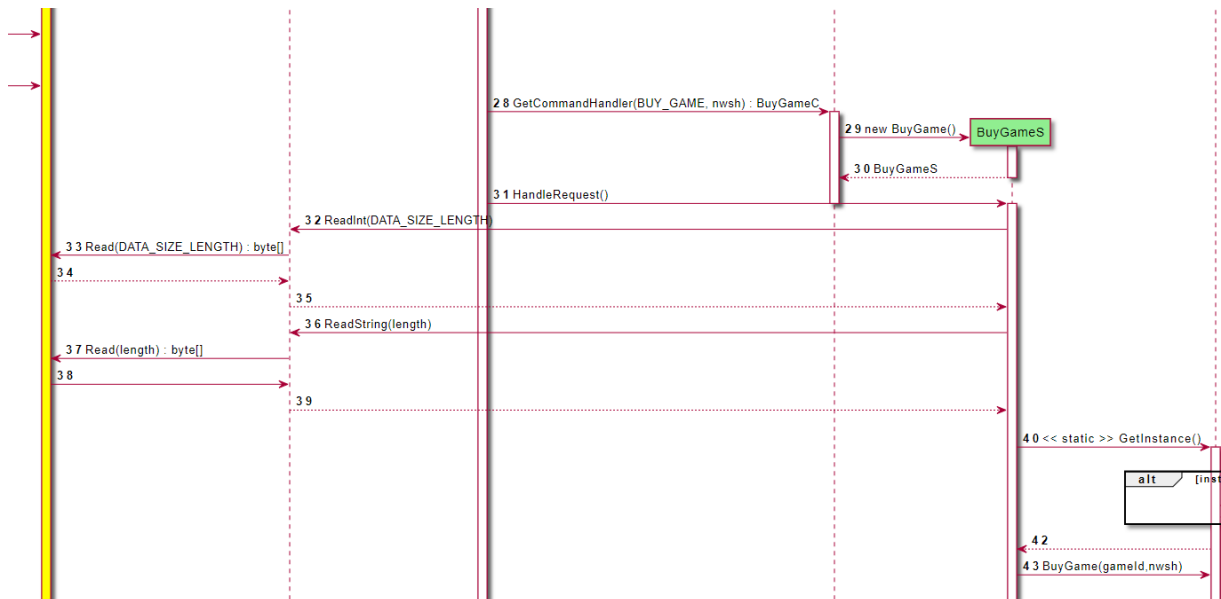
## Diagrama de secuencia de un comando (Buy\_Game)

Recordamos que está adjunto el diagrama completo en BuyGameSecuenciaCompleta.svg. De todos modos vamos a comentar algunas partes

El diagrama comienza cuando se llama a la función BuyGame(gameId). No incluyendo la parte de CliMenu utilizada en ClientePresentation para mostrar las parte de los resultados.



Ahi podemos ver como ClientPresentation utiliza CommandFactory para crear el Comando correcto y le SendRequest de BuyGame empieza a mandar datos al server



Del lado del servidor se lee la información del networkStream (amarillo) y se procesa hasta llegar a BuyGame. Ese BuyGame va a ser el que se encargue de hablar con la BusinessLogic, la BusinessLogic hablar con el data Access y eventualmente el BuyGame, utilizando networkStreamHandler, enviará la información devuelta para el cliente.

Para el diagrama completo utilizamos S y C para diferenciar clases del servidor y del cliente con el mismo nombre respectivamente. Nwsh quiere decir NetworkStreamHandler. Para ahorrar espacio no se escribió la clase cuando se hacía referencia a constantes, ej. en Command.BUY\_GAME, no se escribió Command.

## Control de errores desde el Client

### Validación

Clase estática en Common llamada Validation. Se encarga de validar todo tipo de entradas que pueda ingresar el cliente.

Cuando se piden los datos del juego esta clase se encarga de no aceptar datos vacíos, por ejemplo título o sinopsis vacía. También de datos numéricos se encarga de parsearlos y además que la entrada numérica esté dentro del rango adecuado. En el caso del file path se encarga de verificar que el path exista y que sea con extensión “.jpg”. También en el caso de género del juego y ESRBRating de un juego se encarga de mostrar las listas de opciones y que solo se pueda elegir una opción eligiendo el número del rango de las listas.

Cuando se piden los datos del juego para modificar uno ya existente el caso es similar al anterior pero puede quedar vacío, que sería que ese parámetro no se quiere modificar.

Como las opciones de los menús están dados por opciones numéricas se verifica en todos los casos que la opción que se selecciona está dentro del rango de opciones.

Para todos los casos de una entrada incorrecta se le pide al cliente que reingrese hasta que sea correcto.

Con estas verificaciones se asegura que los request al server se hacen con la información correcta desde el cliente.

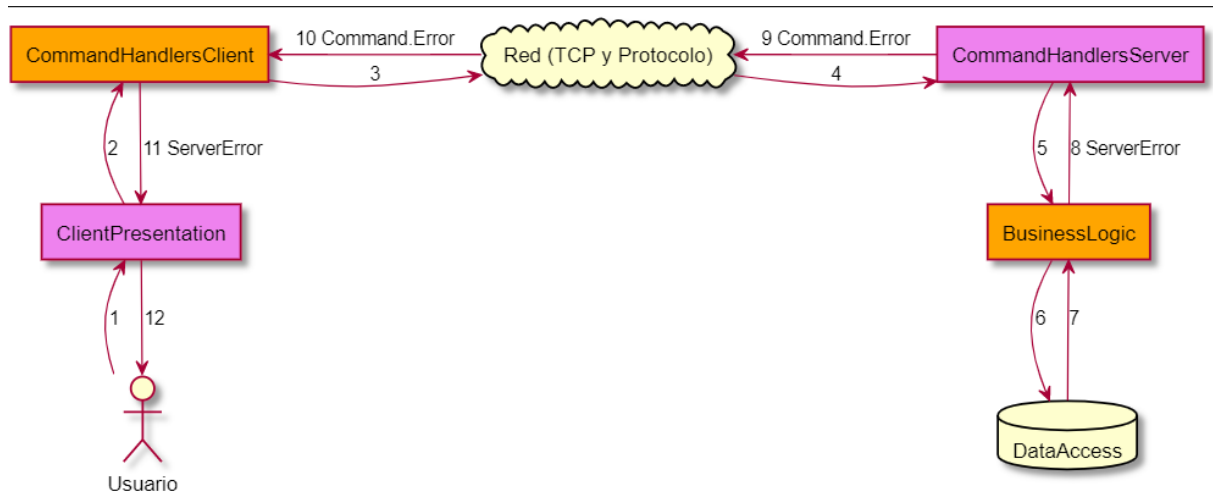
## Control de errores desde el Server

Los errores que ocurren en el servidor se dividen en dos grupos:

1. Funcionamiento normal de la aplicación pero validación del lado del servidor tira error. Ejemplo, si el cliente quiere publicar un juego e ingresa todos los datos correctos. Sin embargo el título es igual al de un juego que ya existe. El servidor tira una excepción pero al cliente solo se le manda un mensaje explicando el error
2. Errores que no forman parte del curso normal. Estos errores son difíciles que ocurran con el cliente que creamos. Por ejemplo, si se manda una id de juego negativa, si bien esto no es posible con nuestro cliente, el servidor realiza el chequeo y la manda el error correspondiente al cliente.

En este caso, la lógica de negocios tira una excepción `ServerError`, es catcheado por los `CommandHandler` del lado del servidor. Los `commandHandler` envían un `Command.ERROR` al cliente. El cliente detecta que le llegó un `ERROR` (y no el comando que estaba esperando). El `commandhandler` del cliente tira una `ServerError Exception`, la `clientPresentation` la catchea y muestra por pantalla el mensaje de error mandado por el server.

En el siguiente diagrama se muestra en naranja las clases que tiran excepciones y en violeta las que cachean excepciones para un caso donde el servidor detecte algún parámetro inválido.



## *Documentación de mecanismos de comunicación*

Protocolo: Protocolo orientado a caracteres.

Implementado sobre TCP/IP.

Se definió la siguiente estructura para los mensajes entre el cliente y el servidor

Nombre Del Campo	HEADER	CMD	LARGO	DATOS
Valores	RES/REQ	0-99	Entero	Variable
Largo	3	2	4	Variable

### Header

El largo es fijo en 3 y tenemos dos constantes:

requestHeader = "REQ": Cuando se hace un request del Client al Server (Steam).

responseHeader = "RES": Cuando se hace la respuesta desde el Server al Client.

### Comandos

LOGIN,  
LOGOUT,  
PUBLISH\_GAME,  
MODIFY\_GAME,  
DELETE\_GAME,  
BUY\_GAME,  
VIEW\_GAME,  
SEARCH\_BY\_TITLE,  
SEARCH\_BY\_RATING,  
SEARCH\_BY\_GENRE,  
WRITE\_REVIEW,  
BROWSE\_REVIEWS,  
DOWNLOAD\_COVER,  
BROWSE\_CATALOGUE,  
BROWSE\_MY\_GAMES,  
ERROR,  
EXIT,

Al tener un Enum con los comandos estos se castean solos a un número entero, correspondiente con el orden, empezando desde 0. El largo del CMD está definido en 2 según el protocolo, se transfiere en forma de bytes.

### Largo

Se define el largo del tamaño de los datos que vamos a recibir a continuación.  
Se transfiere el forma de bytes.

### Datos

A continuación la explicación de la parte de datos para cada uno de los diferentes comandos.

### Comandos de texto

#### *Pedidos Cliente*

LOGIN: solo se envía el nombre de usuario  
“username”

LOGOUT: se envía un string vacío

PUBLISH\_GAME: se envía la data en forma de  
“titulo\_del\_juego|sinopsis\_del\_juego|int\_esrb|genero\_del\_juego”  
(el envío de la portada se especifica en los datos de archivo)  
El delimitador es una constante especificada en el protocolo.  
El resto de la información del juego se inicializa en el server:  
El Id es un número consecutivo, el Publisher será el usuario que está logueado, la calificación del 1 al 10 se inicializa en 0, y las reviews se inicializan vacías.

MODIFY\_GAME: se envía la data en forma de  
“id\_del\_juego|titulo\_del\_juego|sinopsis\_del\_juego|int\_esrb|genero\_del\_juego|int\_modificar”  
El último entero es un 1 o un 0 que representa un booleano dependiendo de si se modifica la portada o no.  
Si alguno de los datos entre los delimitadores está vacío entonces significa que no se quiere modificar ese campo y queda el que ya tenía el juego. El resto de la información se mantiene igual.

DELETE\_GAME: se envía la data en forma de  
“id\_del\_juego”

BUY\_GAME: se envía la data en forma de  
“id\_del\_juego”

VIEW\_GAME: se envía la data en forma de  
“id\_del\_juego”

Con esta informacion se crea un GameView, que contiene un juego, solo con la informacion que se va a mostrar al cliente y dos booleanos (isOwned e isPublisher) que representan si el usuario logueado ya adquirio el juego y ademas si publico el juego o no.

SEARCH\_BY\_TITLE: se envía la data en forma de  
“numero\_de\_pagina|titulo\_del\_juego”

SEARCH\_BY\_RATING: se envía la data en forma de  
“numero\_de\_pagina|rating\_del\_juego”

SEARCH\_BY\_GENRE: se envía la data en forma de  
“numero\_de\_pagina|genero\_del\_juego”

WRITE\_REVIEW: se envía la data en forma de  
“id\_del\_juego|calificacion\_numerica|texto\_reseña”

BROWSE\_REVIEWS: se envía la data en forma de  
“numero\_de\_pagina|id\_del\_juego”

DOWNLOAD\_COVER: se envía la data en forma de  
“id\_del\_juego”

BROWSE\_CATALOGUE: se envía la data en forma de  
“numero\_de\_pagina”

BROWSE\_MY\_GAMES: se envía la data en forma de  
“numero\_de\_pagina”

EXIT: no se envia data

#### *Respuesta del Servidor*

LOGIN: responde con un 1 o 0 que representa un booleano si se logró loguear o no.

LOGOUT: responde con un 1 o 0 que representa un booleano si se logró salir o no.

PUBLISH\_GAME: Crea el juego con la información y lo añade a la lista de juegos del servidor. se contesta con un mensaje que puede ser “Ya existe un juego con ese título” u otros de las excepciones del servidor o "Se publicó el juego {titulo\_del\_juego} correctamente"



MODIFY\_GAME: se crea un juego modificado que luego el servidor modificará dependiendo si las propiedades están vacías o no, y se enviará la nueva portada y se elimina la anterior si se modifica.

El resto de la información se mantiene igual.

La lógica de negocio compara el juego que ya está publicado con el juego que tiene las modificaciones. Los strings de título, género y sinopsis se modifican si la información fue diferente a string vacío (""). El ESRBRating se modifica si es diferente de EmptyESRB (-1). La portada se modifica si es diferente de null.

DELETE\_GAME: se elimina el juego de la lista de juegos. Se recibe "Juego borrado exitosamente"

BUY\_GAME: Se recibe un mensaje "Juego comprado correctamente" o "No se pudo comprar el juego". No se puede comprar el juego si el usuario ya lo tiene.

VIEW\_GAME: se envía la data en forma de

"titulo\_del\_juego|sinopsis\_del\_juego|calificacion\_numerica|int\_esrb|genero\_del\_juego|  
int\_is\_owner|int\_is\_publisher". Esta información se obtiene del struct GameView.

SEARCH\_BY\_TITLE: se envía la data en forma de

"titulo\_del\_juego\_1~id\_del\_juego\_1|titulo\_del\_juego\_2~id\_del\_juego\_2|...|hasNextPage|  
hasPreviousPage"

Se utilizan dos delimiters y el largo depende de la cantidad de juegos que en su título tengan la palabra que se buscó. En lugar de mostrar todos los juegos por consola, estos tienen un límite por página, entonces los últimos ints representan booleanos para poder seleccionar por pantalla para avanzar o retroceder en el listado de juegos. La información de los juegos se obtiene de una clase GamePage que se creó con el nombre del juego que se recibió en el request. No es necesario que el título del juego aparezca entero, con que el título contenga la palabra ingresada se mostrará.

SEARCH\_BY\_RATING: igual que SEARCH\_BY\_TITLE devuelve la data en forma de

"titulo\_del\_juego\_1~id\_del\_juego\_1|titulo\_del\_juego\_2~id\_del\_juego\_2|...|hasNextPage|  
hasPreviousPage" pero se filtra por un número del 1 al 10 que es el promedio de calificaciones de las reviews y muestra todos los juegos que tienen esa calificación o mayor.

SEARCH\_BY\_GENRE: igual que SEARCH\_BY\_TITLE devuelve la data en forma de

"titulo\_del\_juego\_1~id\_del\_juego\_1|titulo\_del\_juego\_2~id\_del\_juego\_2|...|hasNextPage|  
hasPreviousPage" pero se filtra por género. Los géneros posibles están en un array constante. Estos ultimos search al ser similares los tres heredan de una clase que extiende TextCommand que se llama CreateGamePage que es la que se encarga de hacer el respond.

WRITE\_REVIEW: Siempre que se pueda publicar una review de un juego, es decir que el usuario logueado haya adquirido el juego en algún momento, el servidor devuelve :

"Clasificación por {usuario logueado} para el juego {titulo\_del\_juego} fue publicada correctamente". La review se agrega a la lista de reviews del juego. Se utiliza la calificación numérica para calcular el promedio del rating general del juego.

BROWSE\_REVIEWS: se envía la data en forma de

“autor\_de\_la\_review1~calificacion\_numerica1~texto\_reseña1|autor\_de\_la\_review2~calificacion\_numerica2~texto\_reseña2|.....|hasNextPage|hasPreviousPage”

Se usa el delimiter | para separar las reviews y el segundo delimiter ~ separa la información de cada review. Las reviews se muestran por página entonces los últimos dos booleanos representan si existe o no la siguiente página y/o la anterior.

DOWNLOAD\_COVER: se envía la data en forma de

“Id\_del\_juego”

BROWSE\_CATALOGUE: al igual que SEARCH\_BY\_TITLE, hereda del textComand, CreateGamePage, se hace una GamePage con todos los juegos y todas las ids, pero no se usa ningún filtro, se muestran todos los juegos de la lista del servidor que se envía de forma

“titulo\_del\_juego\_1~id\_del\_juego\_1|titulo\_del\_juego\_2~id\_del\_juego\_2|...|hasNextPage|hasPreviousPage”

BROWSE\_MY\_GAMES:

“titulo\_del\_juego\_1~id\_del\_juego\_1|titulo\_del\_juego\_2~id\_del\_juego\_2|...|hasNextPage|hasPreviousPage” solo con los juegos que tiene la lista del usuario actual. También hereda del textCommand CreateGamePage.

ERROR: Se utiliza para controlar los errores del servidor. No hay un request. Se utiliza para informar el error al el Client cuando es catcheado un ServerError.

### Datos de archivo

Para enviar archivos como la carátula, al protocolo anterior se agregan dos datos más

Nombre Del Campo	HEADER	CM D	LARGO	DATOS	FILESIZE	FILE
Valores	RES/REQ	0-99	Entero	Variable	Long	Variable
Largo	3	2	4	Variable	8	Variable

Los siguientes comandos utilizan estos datos:

PUBLISH\_GAME: subida de archivo.

MODIFY\_GAME: subida de archivo.

DOWNLOAD\_COVER: bajada de archivo.

Para la subida y bajada de archivos se siguen los siguientes pasos:

- Se manda el header y el command al igual que el resto de los comandos
- Si se tiene que mandar información en formato string se manda antes de mandar los archivos. Por ejemplo, en publish game se manda todo el resto de información del juego.
- Se manda el largo del archivo. El tamaño del largo del archivo está fijo en la especificación y se agregaron funciones al networkStreamhandler como ReadFileSize para facilitar el uso.
- Una vez que tenemos el largo, leemos el archivo sabiendo el largo.

Para el server

- Se guarda el archivo con un nombre nuevo (usando GUID)
- Si se modifica el juego, se le pone el nuevo path a la nueva carátula pero no se elimina la anterior hasta que no haya nadie descargandola (utilizamos un lock)

Para el cliente:

- Él puede elegir el nombre del archivo, la extensión se coloca sola según la especificación.

## *Errores encontrados*

Si un usuario decide descargar una carátula y escribe como ruta de carpeta, una carpeta que no tiene permisos. Por ejemplo, si quiere guardar la imagen en “C:\”, se tira una excepción. Si puede, por ejemplo, guardar en “C:\nombreDeCarpeta” o “C:\nombreDeCarpeta\”

Posibles soluciones:

- Validar no solo que exista la carpeta (lo cual ya hacemos) sino además conseguir los permisos de esa carpeta y chequearlos con el usuario que está logueado en el sistema. No consideramos esta opción porque pensamos que se iba un poco del alcance del obligatorio y priorizamos otras funcionalidades. Las opciones que investigamos para validar permisos nos parecen complicadas de más.
- Que el usuario ejecute el .exe como administrador.
- No dejar que el usuario elija la carpeta y utilizar una carpeta en referencia a donde está el ejecutable. Esto ya lo hicimos en la parte de “cargar datos de pruebas”, optamos por no hacerlo acá porque consideramos que mejora la usabilidad del sistema poder elegir la carpeta.

## *Posibles mejoras*

- Una vez comenzado el obligatorio, aprendimos sobre los data transfer units, lo cual hubiera sido útil aplicarlos, en vez de utilizar el dominio tanto en el servidor como en el usuario. Esto sería más útil ya que al usuario solo les llegarían/mandaría objetos con los campos necesarios y sin ninguna función. De todos modos, las Pages como GamePage, serian consideradas Data transfer units y las creamos para no pasar listas de clases de dominio o información que el servidor no quiera dar al cliente.
- Mayor separación de la clase de interfaz del cliente (ClientPresentation). Dos cosas complicaron las soluciones que buscamos para separar esta clase en distintas. La primera que al ser una aplicación de consola sencilla, y al necesitar entradas del usuario constantemente, costó separar la lógica de la interfaz. La segunda es que cada menú define a qué menú se va después, lo cual si bien permite mayor flexibilidad a la hora de darle opciones al cliente, complicó dividir el cliente en diferentes objetos sin tener que crear objetos nuevos por cada menú. De todos modos, si buscamos separar funciones, como ClientConfig o utilitarios.