Universidad ORT Uruguay

Obligatorio 1

Programación de Redes

Agustín Ferrari 240503

Francisco Rossi 219401

2021

Índice:

Descripción General:

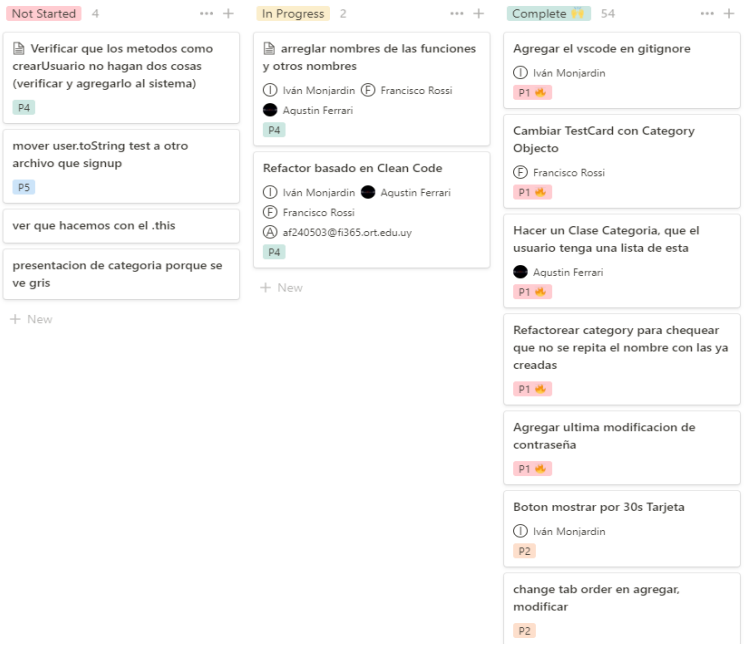
En este proyecto el equipo se propuso como objetivo, desarrollar código profesional, fácil de entender, consistente y extensible. Para llevar a cabo esto, se implementaron técnicas aprendidas durante la carrera, como, recomendaciones de Clean Code, seguir algunos patrones de diseño, entre otras.

El equipo cuenta con 2 integrantes, por lo que se decidió en conjunto, trabajar utilizando GitHub y Git Flow, creando ramas individuales para ciertas funcionalidades y/o refactors necesarios. Se planteó de esta manera para tener la posibilidad de avanzar sin tener que ser necesario la disponibilidad horaria de los otros integrantes todo el tiempo. Para que el código pueda ser consistente y ambos estemos al tanto de los avances, se priorizó el reunirse al menos 1 o 2 veces al día para discutir y sacar dudas entre el equipo.

Se establecieron algunos estándares para trabajar en el repositorio, como por ejemplo:

- El nombre de las ramas feature es "feature-NombreDeLaRama". Se utiliza camelCase y la primera letra del nombre va con mayúscula.
- Para arreglar una funcionalidad se crea una rama “fix-NombreDeLaRama”
- Si se generó un problema al mergear a master, se puede arreglar y commitear directo a master o crear una rama Fix.

Trabajamos con la herramienta Notion. Nos ayudó a manejar las tareas necesarias, y a planear una mejor solución.
Cada vez que se nos presentaba una situación o se agregaba una modificación en el foro, se agregaba en la columna de Not Started y se le asignaba una prioridad del 1 al 5, siendo 1 lo más importante. Luego cada vez que algún integrante se ponía a desarrollar, elegía una tarea, la marcaba como In progress y creaba la rama para trabajar en ellas. Finalmente se pasaba a la columna de Complete y se podía avanzar con otra tarea.



Alcance de la aplicación

La aplicación ofrece varias funcionalidades para los clientes. Se puede ver un listado de los juegos en el sistema sin la necesidad de iniciar sesión, y preguntando al cliente si quiere filtrar los mismo por nombre, género y/o rating. Reservamos el nombre de usuario que se elija para loguearse para que sea único en el sistema, con lo que se habilitan 8 acciones más a las de loguearse y mostrar el catálogo de juegos, que también se puede realizar una vez que inicie sesión.
Se tiene la opción de cerrar sesión, y poder volver a conectarse manteniendo los datos del usuario, como por ejemplo, ver la biblioteca, que es una funcionalidad extra que consideramos importante, para saber que juegos compró el usuario. También se puede comprar un juego, o publicarlo al sistema como desarrollador y dueño del mismo, subiendo todas sus características como la carátula o imagen de forma que todos los datos estén completos. Esto le genera permisos para modificar algún atributo del juego o eliminar el juego completamente si el usuario que requiere esta acción es el dueño.
Por último se puede hacer una review del juego, no es necesario comprarlo para poder realizar esto, y luego si se quiere revisar sus calificaciones antes de comprarlo, se puede ver el detalle del juego deseado, generando un promedio de todas las reviews que los usuarios realizaron sobre este juego.

Descripción y justificación de Arquitectura:

Para este obligatorio se realizó un aplicación con arquitectura cliente-servidor, todos los pedidos que realiza el cliente pasan por el servidor y reciben una respuesta acorde que es mostrada en la interfaz gráfica al usuario. Se diseñó el servidor para poder aceptar conexiones de múltiples clientes al mismo tiempo, esto implicó tener que manejar diferentes hilos que permiten escucharlos. Es por esto que se decidió crear una solución dividiendo el código del cliente y servidor en proyectos diferentes.
El primer paso, fue crear distintos proyectos, uno para el cliente, otro para el servidor y otro para lo que comparten ambos. Esto se hizo para aislar los comportamientos de cada uno.
Los proyectos se plantearon así, pensando en una arquitectura cliente-servidor, donde cada parte tiene sus propios requerimientos, intercambiando mensajes o instrucciones entre sí.

Cliente

Para la aplicación cliente se creó una aplicación de consola, que fue dividida en dos menús, uno para el cliente invitado (que no esté autenticado como un usuario en el sistema) y otro para los usuarios logueados, cubriendo así todos los requerimientos de la aplicación.
En cuanto a la estructura del cliente, se creó un paquete conteniendo los elementos del Menú, el cual se divide en Presentation y Logic. El paquete de Presentation se encarga de manejar las impresiones del menú. El paquete de Logic se encarga de manejar todos los inputs del cliente y comunicarse con el servidor. En un principio todo esto se realizó en una sola clase, lo cual perjudicó la legibilidad y mantenibilidad del código, además de generar problemas para seguir con los principios de Clean Code (por ejemplo switches de más de 10 casos).
Para resolver estos problemas se decidió implementar los patrones Strategy y Factory, para de esta forma reorganizar el comportamiento de los métodos encargados de cada opción del menú y eliminar code smells como el mencionado anteriormente. Si bien el patrón Strategy se implementa en la mayoría de los casos como una interfaz, decidimos hacerlo con una clase abstracta, para que de esta manera aprovechar el constructor de la clase padre para tener una instancia para poder acceder a los métodos de MenuHandler y no repetir este código en las clases hijas de la strategy padre.

Servidor

Al igual que en el lado del cliente, la aplicación servidora fue implementada como una consola, con la diferencia de que en este caso la interfaz es mucho más simple, ya que al correr el servidor ya comienza a esperar clientes y la única función del menú es para cerrarlo.
Al ser el servidor el que maneja la lógica de negocio del proyecto, existe un aumento de código a comparación con la parte de cliente. Para organizar las clases, se decidió separarlo en varios paquetes:

Logic: Conteniendo lógica pura del servidor, como manejo de clientes y comandos.
Presentation: Al igual que en la parte del cliente, encargado de mostrar y manejar el menu.

Domain: Clases de dominio utilizadas para almacenar los datos.
BusinessLogic: Lógica de negocio, manejo de datos de usuarios y juegos.
Utils: Excepciones custom, clases para pre-cargar datos en el catálogo de juegos.

Pudimos implementar todas las funcionalidades requeridas e incluir algunas que consideramos importantes para mejorar la usabilidad del sistema, como que el usuario pueda ver los juegos que compró (biblioteca). Las decisiones de cómo y cuáles funciones implementar fueron tomadas en conjunto por el equipo.

Mecanismos de comunicación

Protocolo

Para establecer la comunicación entre el servidor y los clientes, se usó un protocolo orientado a caracteres, implementado con Sockets sobre TCP/IP. Este cuenta con un header, el cual tiene un largo fijo de 9 caracteres, dividiéndose en:

- 3 para sí es un pedido (REQ) o una respuesta (RES).
- 2 para el comando al que se está haciendo referencia.
    - Los comandos (como puede ser Login, AddGame, etc.) están definidos en el proyecto Common, y son representados como ints, pudiendo acceder a cada uno como constante. Teniendo en cuenta que tenemos 2 caracteres para enviar en nuestro protocolo, la cantidad máxima de comandos de la que disponemos es 99, en nuestro caso usamos solamente 12.
- 4 para el largo de datos, es decir se pueden enviar mensajes de hasta 9999 caracteres.

Además se especificó que el largo máximo de los paquetes, sería de 32KB.

Respuestas

Para las respuestas del servidor a los clientes, además de tener que especificar en el header que se trata de un tipo RES, se decidió crear mensajes predefinidos como constantes en el proyecto Common. De esta forma se logró reducir notablemente la cantidad de errores de comunicación al momento de interpretar los mensajes, ya que en lugar de hardcodear todos los mensajes de ambos lados (cliente y servidor), lo que puede inducir a errores causados por typos por ejemplo, se crearon constantes en el paquete en común para no tener que manejar directamente los mensajes ni del lado del cliente ni del servidor. Esto también es beneficioso en caso de tener que cambiar uno de los mensajes respuesta, ya que solo se necesita cambiarlo en la definición del protocolo sin riesgo de introducir nuevos errores.

Imagenes

En cuanto al envío de imágenes, también se usó el Socket, mientras que para almacenar estos archivos se usó FileStream, creando un handler e interfaces para poder realizar la inversión de dependencias. Esto se ubica en el paquete común, ya que ambas partes de la comunicación necesitan enviar y recibir imágenes.
Se decidió que el formato de las mismas sea únicamente en PNG. En caso de intentar subir un archivo de otro formato, se le notifica al usuario y se cancela la acción.

Para los pedidos de imagenes, se decidió que estas se envíen al final de los datos, en tres partes:

- La primera para el largo del nombre de la imagen y el largo de la imagen, siendo de 4 y 8 bytes respectivamente. En caso de no llegar a tener 4 bytes de largo de nombre, se rellena con 0s a la izquierda, lo mismo para el largo del archivo. Haciendo esto nos aseguramos de que el largo de los datos de la imagen es siempre 12 y a que corresponde cada byte para poder leerlo posteriormente. Todos estos datos se encuentran guardados en constantes en la clase Specification de Common, permitiendo cambiarlos sin mayores problemas y haciendo más sólido el protocolo, ya que es desde el único lugar al que se acceden a estos datos.
- La segunda es para enviar el nombre del archivo, para poder incluirlo en el nombre que se almacena en disco.
- La última es para el envío de la imagen, para esto se debe haber leído previamente el largo y el nombre inicial del archivo. Una vez recibido el archivo en su totalidad, se procede a escribirlo en el path especificado en el app, para esto, se le concatena el nombre del juego adelante seguido de un guión bajo, para asegurarnos de que el nombre sea único.

  En caso de no poder leer la imagen, se notifica al usuario y se pide que revise los permisos del archivo antes de subirlo.

Control de concurrencia

Mecanismos de mutua exclusión

Al permitir la conexión de múltiples clientes que potencialmente puedan querer acceder a los mismos recursos como listas de usuarios, juegos, etc. Esto puede traer errores como lectura sucia o actualización perdida. Para solucionar este problema se decidió implementar un mecanismo de mutua exclusión basado en locks, bloqueando la lectura y escritura de las listas en caso de que otro thread quiera acceder a las mismas. De esta manera, sólo un thread puede acceder a las listas a la vez y nos aseguramos de que no ocurran estos problemas.
Las dos listas principales del sistema son la de juegos y usuarios, las mismas se encuentran en business logic dentro de sus controladoras respectivas. Por cada una de estas listas se utiliza un lock, es decir, se puede leer/escribir un juego, aunque el lock de los usuarios esté activado.

Singleton

Otro de los problemas que pueden ocurrir es que se creen múltiples instancias de las controladoras, que tienen la lista de usuarios y juegos, o otras clases similares. Para resolver este problema, se decidió implementar el patrón singleton con locks en las clases que se vean afectadas y así evitar que existan varias instancias pudiendo provocar que se creen varias listas de juegos en paralelo y no una única que contenga todos los juegos del sistema, por ejemplo.

Descripción y justificación de diseño:

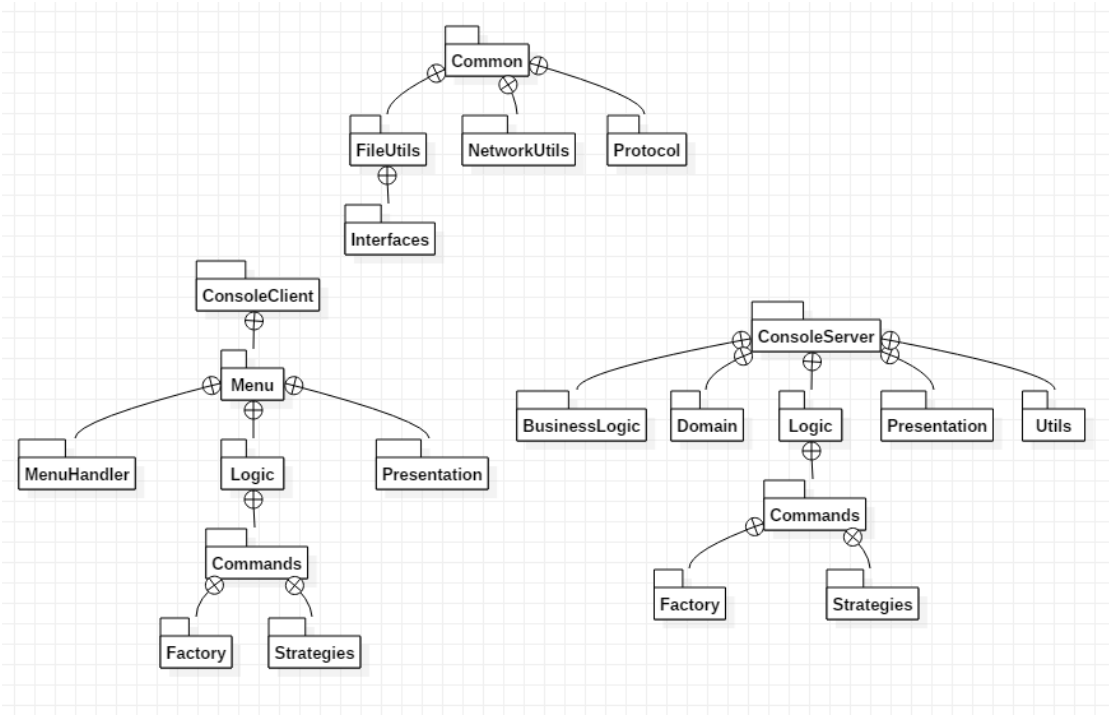
Paquetes

Se buscó que el acoplamiento fuera bajo y la cohesión alta. Agregamos varios paquetes a la solución, pero dos de los más importantes fueron Logic tanto del lado del servidor como del lado del cliente. El primero fue una decisión de diseño para poder manejar todas las request que el usuario solicitara hacer. Donde se evitó crear un switch que manejara las distintas opciones, debido a que el tamaño sería demasiado extenso y tampoco seguiría las recomendaciones de Clean Code, y el segundo para enviar todas las solicitudes de cada cliente, indicando por medio de la consola que instrucciones requería.
También se separó la parte de BusinessLogic, del paquete Domain, y Utils. Ya que el mismo se encargará además de ser el experto en manejar la listas de usuario en el sistema y de los juegos, de modificar todos los datos que involucra el sistema.

Para resolver el problema del switch extenso, se ideó una solución con paquetes para que el sistema sea escalable. En el paquete Logic\Commands, se crearon dos paquetes, Factory y Strategies. Se decidió utilizar el patrón Factory, encargado del manejo de las strategies. Factory es el encargado de recurrir a la strategy correcta según el caso de la solicitud, donde el comando de acción involucra una implementación concreta de la misma. Estas decisiones de diseño fueron implementadas tanto en el cliente como en el servidor. Se decidió implementarlo así, para en un futuro si se decide crear nuevas acciones, el sistema pueda seguir su diseño inicial, y simplemente agregar pocas líneas de código para agregar una nueva opción en el menú, agregando una nueva strategy.

A medida que el código fue creciendo, en función de las pruebas y los principios de Clean Code, se fueron haciendo refactors para mejorar la solución, como por ejemplo evitar los magic numbers. También se recurrió a diseñar una solución en base a los patrones y principios de diseño. Por ejemplo, el paquete Presentation en el servidor, separando su interfaz gráfica del resto de la lógica. Se recurrió al principio de inversión de dependencia, para que las dependencias se den entre interfaces preferentemente, en vez de clases concretas.

Para el manejo de errores, decidimos crear excepciones personalizadas que fueron utilizadas en todas las acciones del sistema. Se encuentran en el paquete CustomExceptions dentro del paquete Utils. Se planteó de esta manera para que a la hora de capturarlas fuera más sencillo y seguir las recomendaciones de Clean Code.



[1]

En el paquete Domain, se encuentran los objetos concretos que maneja el sistema, Game, Review y User, se detallará más adelante sobre el mismo.

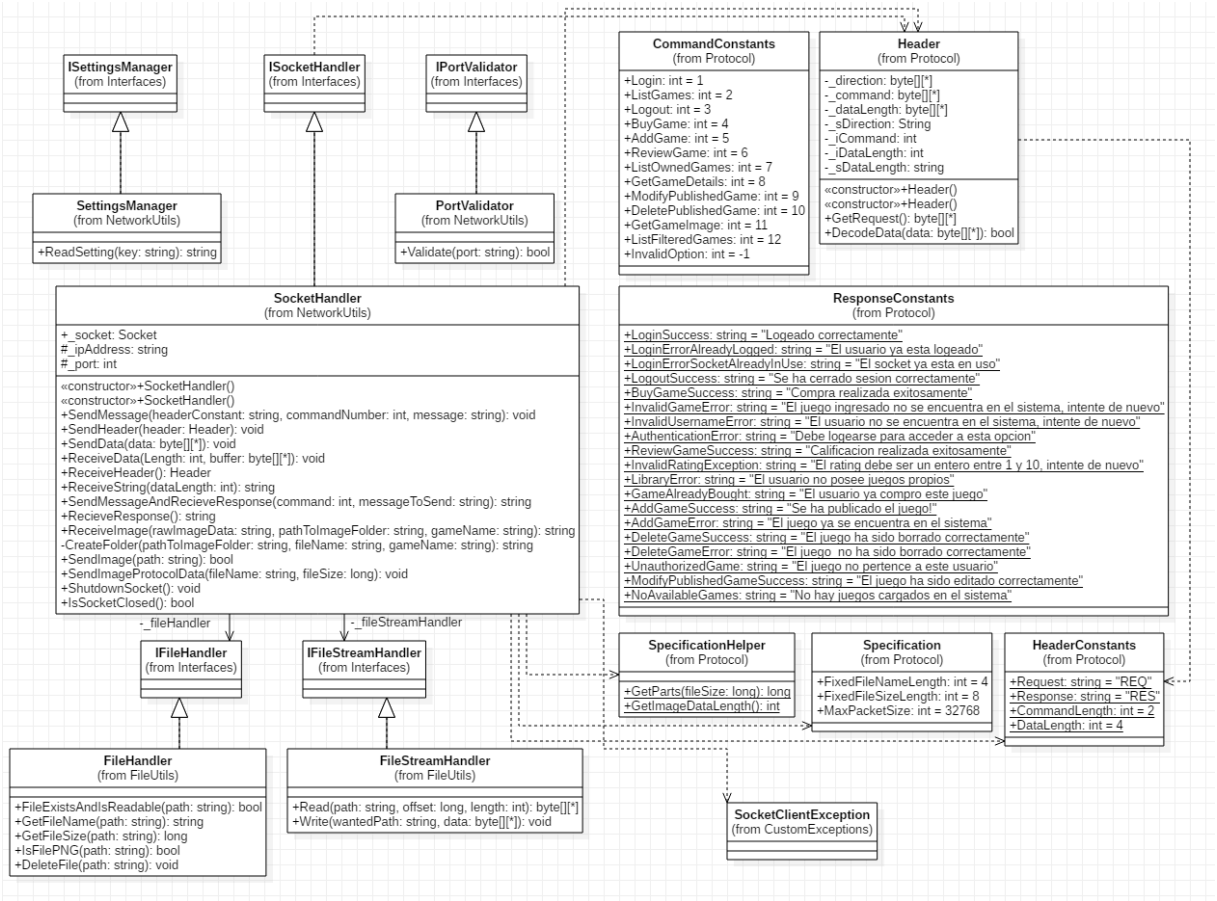
Por último en el proyecto Common, contamos con 3 paquetes, FileUtils, NetworkUtils y Protocol. El primero, es el encargado del manejo de archivos, obtiene un rol fundamental en la carátula del juego. NetworkUtils se basa en el Protocolo y el FileUtils que definimos, para realizar la comunicación entre cliente y servidor. Tiene funcionalidades como las de mandar/recibir header y mensajes para que el sistema funcione correctamente. Finalmente, Protocol, es el que tiene el protocolo de red que definimos para que la comunicación se base en un intercambio de caracteres entre ambos involucrados (cliente, servidor)

[1] Diagrama encontrado en Diagrams/Package.SVG

Diagramas de clases:

Para entender más específicamente cómo diseñamos cada paquete, podemos observar los siguientes Diagramas de Clases, comprendidos en diferentes proyectos:

Proyecto Common [2]



FileUtils

En este paquete contamos con dos clases concretas:

- FileHandler : Se asegura que el archivo a leer exista y también obtiene el nombre, tamaño a partir de un path.
- FileStreamHandler : Lee información, cargándola en un buffer de bytes y también es responsable de escribir información dado cierto path del sistema operativo.

Ambas implementan interfaces, que luego son ellas las que exponen los métodos disponibles.

NetworkUtils

Cuenta con 3 clases concretas:

- PortValidator : Comprueba que el puerto del Appconfig se encuentre en un rango y formato correcto .
- SettingsManager : Lee el puerto y IP configurada en el Appconfig.

[2] Diagrama encontrado en Diagrams/Common.SVG

- SocketHandler : Se encarga de manejar los sockets, tanto del cliente como del server. Heredan de ella ClientSocketHandler y ServerSocketHandler.

Las 3 clases implementan interfaces, que luego son ellas las que exponen los métodos disponibles.

Protocol

En este paquete hay 6 clases concretas.

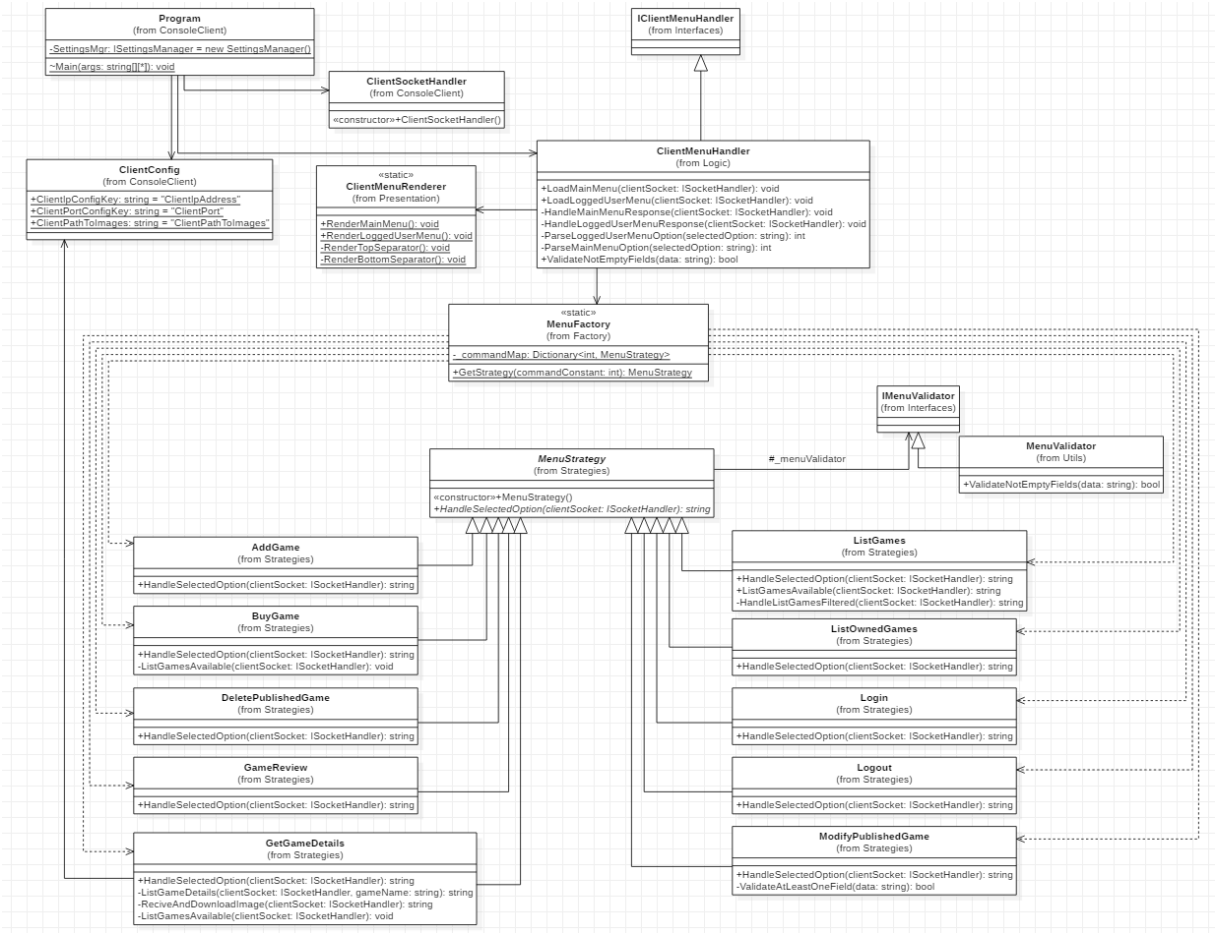
- CommandConstants : Define todas las opciones que puede ingresar el cliente en el menú como constantes, para indicar cómo manejar la acción a lo largo del sistema.
- Header : Indica que tipo de información se va a mandar, tiene un dirección que indica si es un REQ o RES, un comando o acción, y el largo de los datos a enviar.
- HeaderConstants : Define ciertas constantes, para establecer un formato del Header. El comando tiene que ser de largo 2, si no se le agrega un 0 adelante y el largo de la información puede ser máximo de 4 bytes.
- ResponseConstants : Luego de realizada una acción concreta, el servidor siempre responde al cliente con respuestas predefinidas según la acción, dependiendo si pudo completar la tarea.
- Specification : Indica cuanto es el máximo de bytes de de un paquete, como también el largo del nombre de un archivo y el tamaño de un archivo.
- SpecificationHelper : Dado el tamaño de un archivo, indica en cuantas paquetes puede el protocolo mandar la información y también indica el el largo que ocupa el nombre y el tamaño de un archivo.

Utils.CustomExceptions

Definimos 1 excepción personalizada en el proyecto Common.

- SocketClientException : Indica si se perdió la conexión con el socket del cliente.

Proyecto Client [3]



Menu.Logic.Commands.Factory

En este paquete existe solo una clase:

- MenuFactory : Devuelve una strategy adecuada, según la acción que requiera el usuario desde el menú. Las strategies están asociadas a un comando en específico para poder mapearlas con un diccionario. Esto evita tener un switch para cada acción y sigue los principios de OCP.

Menu.Logic.Commands.Strategies

Cuenta con 11 clases concretas

- AddGame : Estrategia que maneja la publicación de un juego al sistema. No se puede omitir ningún campo requerido para la creación del mismo.
- BuyGame : Resuelve la acción de comprar un juego del sistema, ingresando el nombre del juego deseado, a partir de la lista de juegos disponibles.
- DeletePublishedGame : Elimina un juego, el usuario que requiera esta acción, deberá ser el mismo que publicó este juego en el sistema.

[3] Diagrama encontrado en Diagrams/Client.SVG

- GameReview : Para que el cliente pueda ingresar una calificación al juego, esta estrategia resuelve esta acción.
- GetGameDetails : Obtiene los detalles de algún juego del sistema, ingresando el nombre del mismo. El cliente decide si quiere descargar la imagen o no ingresando el número 1 por consola.
- ListGamesLoggedUser : Separamos en dos strategies el listado de juegos en sistema. Cuando el usuario está logueado y cuando no lo está. Ambos pueden filtrar por nombre, género y calificación, sin tener que ingresar todos los filtros para realizar esta acción.
- ListGamesMainMenu : Realiza una acción similar a la clase mencionada anterior, la diferencia es que cargan menú diferentes luego de traer la lista de juegos. Debido a que la respuesta del servidor para estas acciones son la propia lista de juego y no una respuesta constante, decidimos implementarlo de esta manera, con dos strategies separadas.
- ListOwnedGames : Esta acción no era un requerimiento de la letra, pero pensamos que era importante poder saber que juegos había adquirido el usuario.
- Login : Los usuarios se pueden loguear al sistema, en esta instancia el único requerimiento para loguearse, es que el nombre que ingrese, no se encuentre en el sistema activo en este momento. Se manejó un diccionario de usuarios logueados, que se asocia al socket de la conexión y el nombre ingresado.
- Logout : Permite que el usuario pueda salir del sistema.
- ModifyPublishedGame : Resuelve la acción de modificar un juego, se deben ingresar el nombre del juego publicado por el usuario a modificar y al menos uno de los campos que se quiera cambiar.

Estas strategies, heredan de una clase abstracta MenuStrategy, que se encarga de poder acceder a las funciones ClientMenuHandler, como validar campos, o cargar distintos menús en las strategies. Todas las strategies implementan HandleSelectedOption, una función declarada en esta clase abstracta, que resuelve la acción requerida dependiendo la solicitud del cliente.

Menu.Logic

En este paquete se encuentra ClientMenuHandler, que implementa la interfaz IClientMenuHandler y es como expone sus métodos. Esta clase es la encargada de cargar los menú y también de manejar los comandos que ingresa el usuario, utilizando la Factory y strategies mencionadas antes.

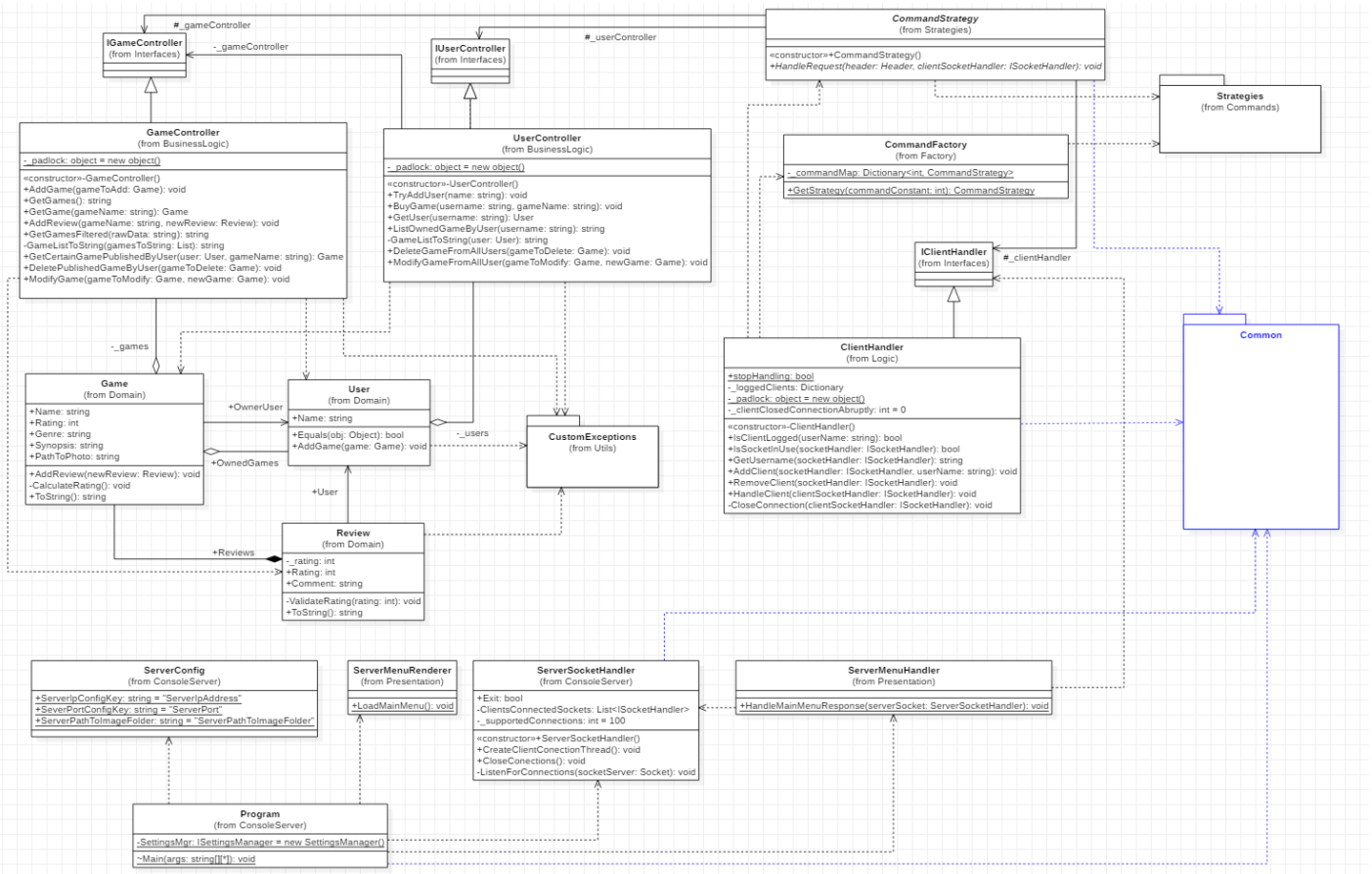
Menu.Presentation

Nuestro diseño tiene una interfaz gráfica basada en una consola. ClientMenuRenderer es la clase que se encarga de imprimir los dos menú disponibles, MainMenu y LoggedUserMenu.

ClientConfig, ClientSocketHandler y Program son clases dentro del Proyecto ConsoleClient.

- ClientConfig : Se utiliza para poder utilizar correctamente el App.config y no tener que poner el puerto, IP e path a guardar imagen en varios lugares, solo se debería modificar en el App.config.
- ClientSocketHandler: Hereda de SocketHandler y una vez que un cliente nuevo se quiere conectar, genera una instancia y conecta el socket.
- Program : Una vez que valida el puerto, con el IP y puerto de App.config, conecta el socket y carga el menú principal para que comience a funcionar la aplicación cliente.

Proyecto Server [4]



[4] Diagrama encontrado en Diagrams/Server.SVG

BusinessLogic

Cuenta con 2 clases concretas:

- GameController : Es la controladora que tiene la lista de juegos del sistema, y realiza las operaciones correspondientes sobre ellos. Agregar, borrar, modificar y obtener juegos según la strategy lo requiera.
- UserController : El sistema diferencia los usuarios logueados y activos, con los usuarios que ingresaron al sistema. Esto se hace para que se pueda salir del sistema y volver a ingresar, pudiendo acceder a sus juegos adquiridos.

Para ambas controladoras se implementó el patrón Singleton, para que solo exista una instancia para todos los clientes. Con este patrón y la implementación de mutua exclusión, nos aseguramos que no se generen errores si varios clientes se conectan al servidor y solicitan distintas acciones sobre los juegos del sistema.
Exponen sus métodos por medio de una interfaz cada controladora, IGameController y IUserController.

Domain

Este paquete se crea para contener los objetos utilizados en el sistema:

- Game : Es el objeto juego del sistema, que tiene ciertas properties como Name, Rating, una lista de Reviews y un OwnerUser siendo este, el cliente que publicó el juego.
- Review : Los juegos tienen una lista de Reviews, estas tienen Rating, Comment y un Usuario, para identificar quién la creó.
- User : Un objeto que identifica al usuario logueado, contiene una lista de los juegos que el cliente compró.

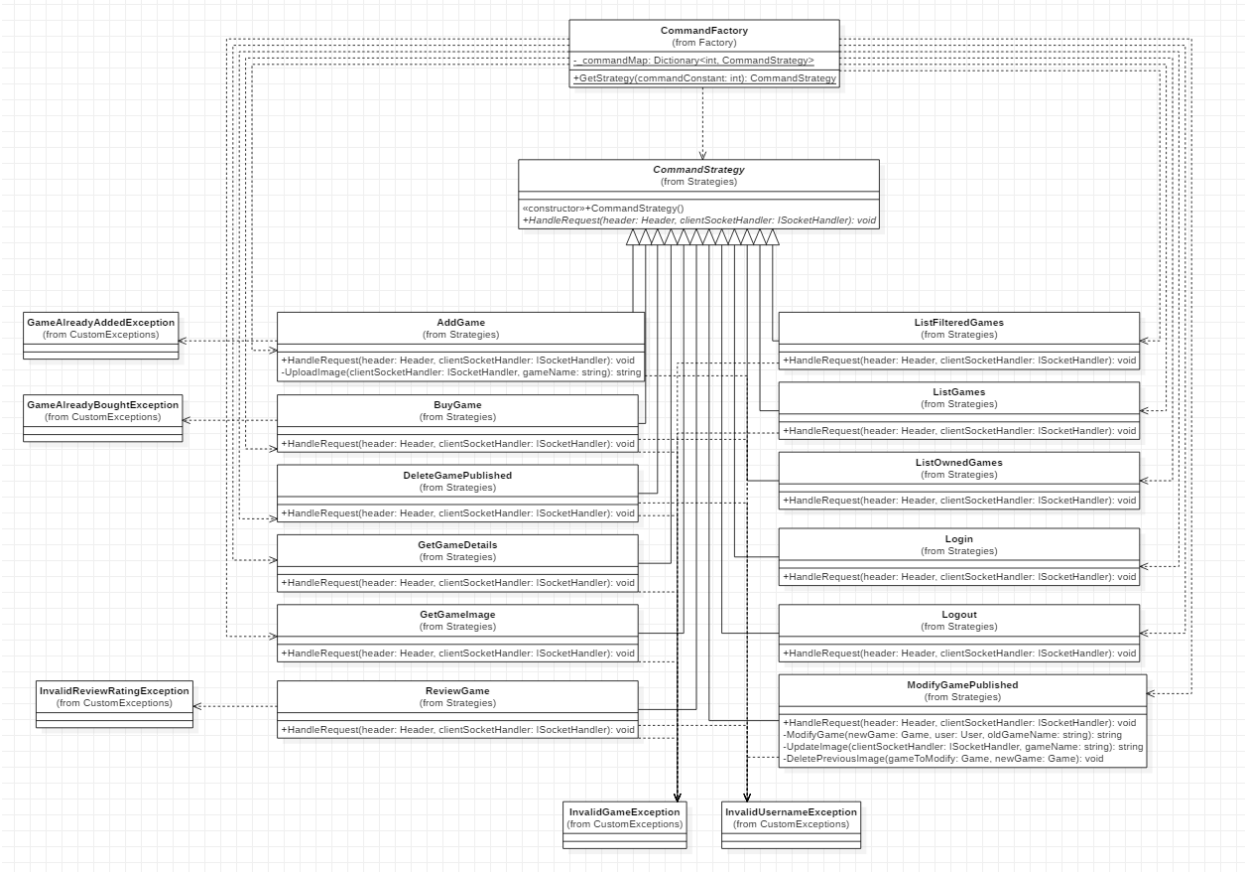
Logic.Commands.Factory

Del mismo modo que en el proyecto del cliente implementamos el patrón Factory, para esto, creamos una Factory CommandFactory, que por medio de un comando que recibe del proyecto cliente, maneja por medio de una strategy la acción a tomar. Este diseño nos soluciona tener que utilizar un switch y manejar cada acción que el cliente enviará por medio del socket con un comando.

Logic.Commands.Strategies

Este paquete cuenta con 11 clases concretas y CommandStrategy, una clase abstracta que obtiene las instancias de las controladoras y del ClientHandler.
Las strategies resuelven las acciones que se mandan los clientes por medios de comandos y devuelven un mensaje para saber cómo resultó la acción solicitada para que la aplicación cliente tome decisiones según como resultó en el servidor. Por ejemplo cuando se ingresa un

juego a comprar que no se encuentra en el sistema, el server manda una respuesta constante y el cliente si recibe ese mensaje lo muestra por pantalla e indica que no se pudo realizar la compra.
Las strategies son las mismas que las del cliente pero con distintas funcionalidades, debido a que se usan los mismos comandos a lo largo del sistema (CommandConstants). Estas trabajan con los datos necesarios para llamar a las controladores según corresponda, y si los métodos de las controladoras tiran excepciones, las catchea y le manda un mensaje al cliente.
[5]



Logic

Cuenta con la clase ClientHandler. Esta clase expone sus métodos por medio de una interfaz IClientHandler y maneja un diccionario de usuarios logueado en el sistema. La clave es el socket y el valor el nombre de usuario que es único. Se implementa el patrón Singleton para que este diccionario sea único a lo largo de la vida del sistema y utilizamos locks para bloquear el acceso y permitir la mutua exclusión con varios clientes.
Se encarga de manejar el pedido del cliente, obteniendo una strategy por medio de una factory y el comando recibido. Si ocurre algún error o el cliente cierra la consola, cierra la conexión y lo elimina del diccionario de usuarios activos.

[5] Diagrama encontrado en Diagrams/ServerStrategy.SVG

Presentation

ServerMenuHandler, es el responsable de manejar la interfaz gráfica de la aplicación servidor. La única opción que implementamos, es si se quiere cerrar el servidor, por lo cuál el cliente no podrá enviar ningún pedido posterior a esto.
ServerMenuRenderer muestra las diferentes acciones que permite el servidor por consola, en este caso sólo “Salir del sistema”.

Utils.CustomExceptions

Definimos excepciones personalizadas para distintos errores que pueden llegar a ocurrir durante la ejecución de acciones solicitadas por el cliente:

- GameAlreadyAddedException : El juego ya se encuentra en el sistema, no se puede agregar dos veces, chequea que el nombre sea único antes de agregarse.
- GameAlreadyBoughtException : Este usuario ya tiene este juego en su lista de juegos comprados, no se puede comprar dos veces.
- InvalidGameException : El juego buscado, no se encuentra en el sistema.
- InvalidReviewRatingException : Verifica que el rating del juego esté en un formato adecuado antes de cargarlo en el sistema.
- InvalidUsernameException : Catcheamos este error, por prevención. Verifica que el usuario esté en el sistema.

Utils

CatalogueLoader es una clase auxiliar para precargar juegos, todos bajo un usuario ficticio Juan. Esto nos sirvió para probar varias funcionalidades durante el desarrollo del sistema. Es por esto que decidimos entregar esta clase junto con el resto del proyecto. Se decidió crear los juegos sin imágenes, ya que en caso de cambiar de computador, se perderán las mismas, por lo cual no tiene sentido a menos que se descarguen las imágenes en el path donde elija el usuario. De todas maneras con la función de modificar juegos, se le puede agregar una carátula a los que ya están cargados como pruebas.

Program, ServerConfig y ServerSocketHandler se encuentran bajo el proyecto ConsoleServer.

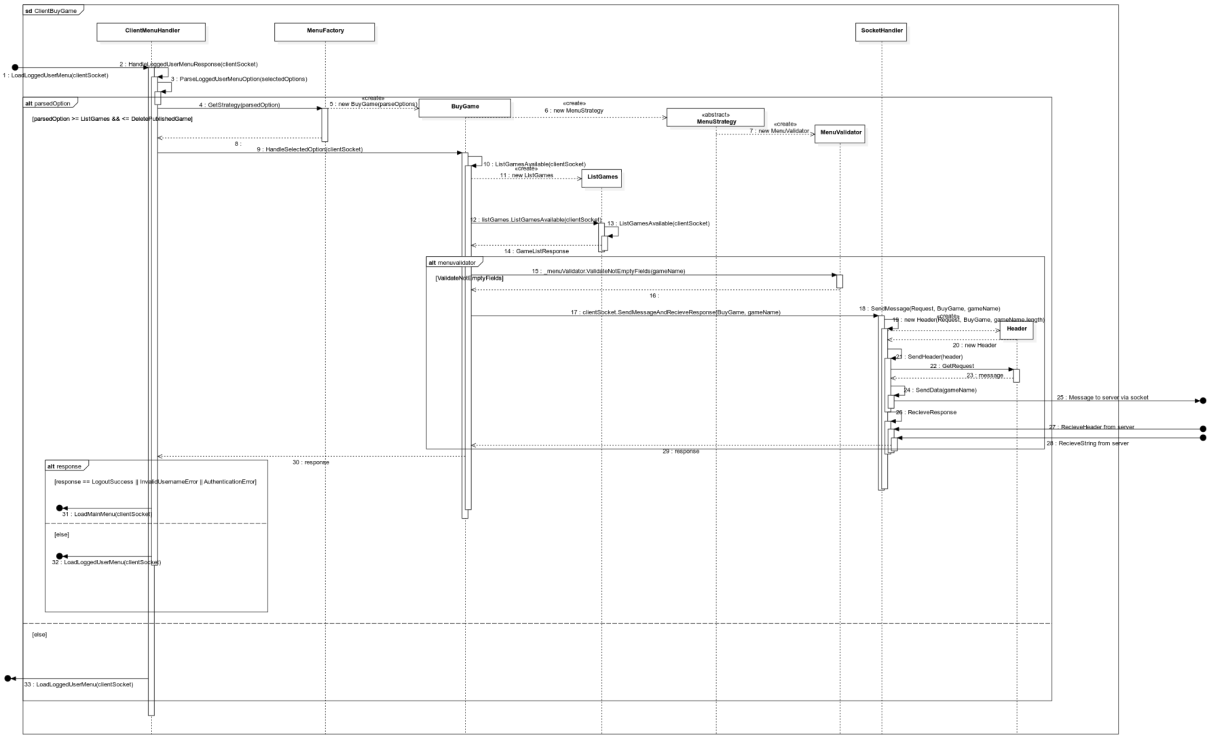
Program obtiene puertos e IP configurados para el proyecto por medio del App.config al igual que el proyecto del cliente. Para efectos prácticos, la configuración es la misma para ambos proyectos, para poder correr en una misma máquina la aplicación cliente como la del servidor. Esta clase instancia un ServerSocketHandler, encargado de las conexiones con clientes y luego se encarga de manejar la entrada en la consola del servidor, si se ingresa el comando para salir de la misma.

ServerConfig es una clase auxiliar de App.config para poder utilizar sus claves, y solo tener que modificar ese archivo para que la aplicación funcione correctamente.

ServerSocketHandler tiene definido un número arbitrario de 100, que indica la cantidad de conexiones, o clientes que acepta simultáneamente. Está "escuchando" de cierta manera si ingresan nuevos pedidos de conexión, mientras el server esté activo. Cuando ocurre esto, crea un nuevo thread que además lo inicia, permitiendo el uso de la aplicación y manejo de comandos en paralelo para todos los clientes conectados.
Si se corta la conexión, por medio de la consola del servidor, se encarga de cerrar todas las conexiones de los sockets que estén activas.

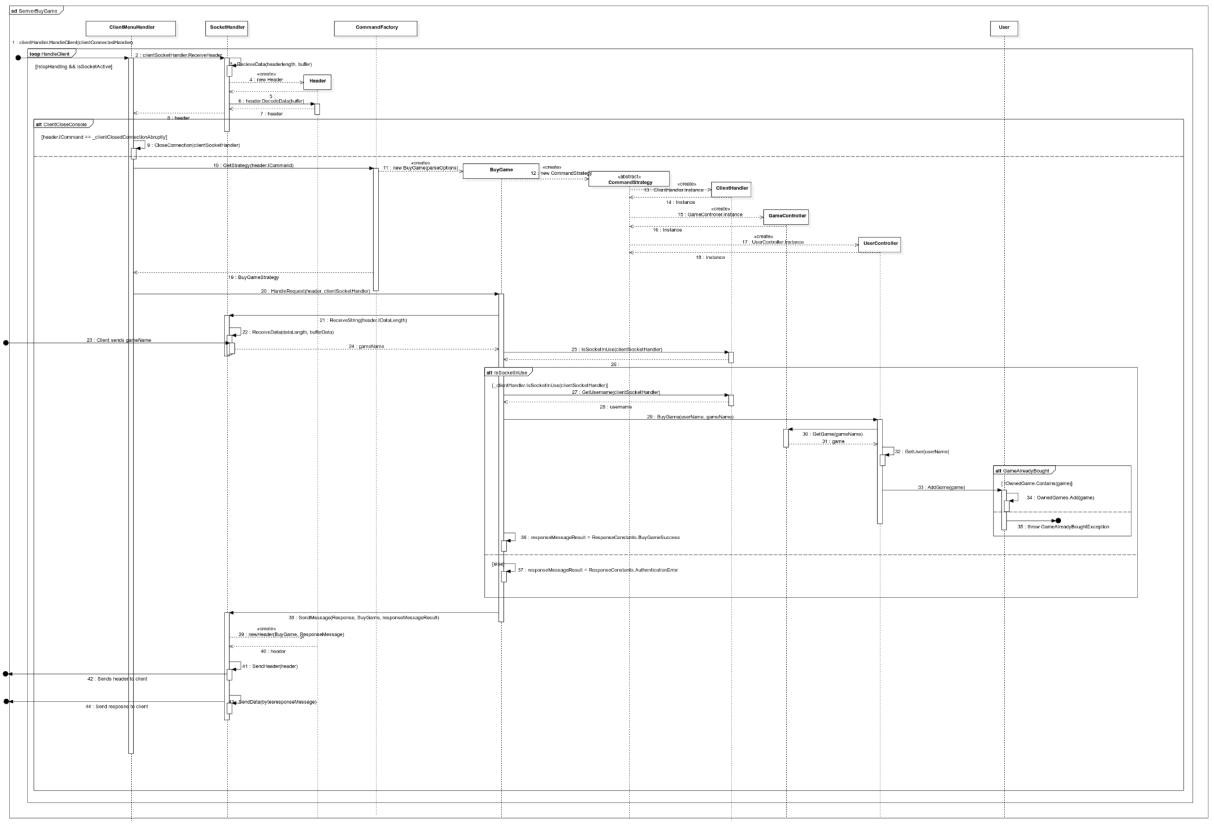
Diagramas de Interacción

En este diagrama, se quiso reflejar las clases involucradas en la acción de comprar un juego del sistema para un usuario previamente logueado. La acción comienza cargando la interfaz gráfica para que el usuario seleccione la opción de comprar un juego y termina cuando del lado del servidor, se recibe una respuesta de la acción solicitada.
Esta acción decidimos separarla en dos diagramas de secuencia, una que refleje la solicitud del cliente por consola y como la Factory crea la strategy adecuada para comprar un juego. En el diagrama se omite la funcionalidad interna de mandar información por medio de sockets, para que no agregue complejidad innecesaria que no aporta a la acción.
La segunda parte o parte del servidor del diagrama es la acción de recibir un comando de comprar un juego y comunicarse con las clases, controladoras para agregar el juego al cliente si es que ya no lo tiene y el juego existe, devolviendo una respuesta por medio de sockets.



6

[6] Diagrama encontrado en Diagrams/BuyGameClient.SVG



7

[7] Diagrama encontrado en Diagrams/BuyGameServer.SVG

Configuración

Para la configuración personal de los usuarios, tanto como para el server como para el cliente, se decidió crear un archivo App.Config en formato XML donde se pueden configurar los parámetros de configuración con los que se ejecutan ambas aplicaciones.
Estos son:

Server:

ServerIpAddress: La IP donde corre el server.

ServerPort: El puerto que escucha el server.

ServerPathToImageFolder: El directorio donde se guardaran las imágenes de todos los juegos.

Cliente:

ServerIpAddress: La IP del server a donde se conecta el cliente.

ServerPort: El puerto del server al que se conecta el cliente.

ClientPathToImageFolder: El directorio donde se guardaran las imágenes que recibe el cliente el servidor.

Estándares de codificación

Algunas de las decisiones de diseño ya fueron comentadas anteriormente, sin embargo queríamos resaltar otros puntos relevantes.
Al inicio del proyecto entendimos que era necesario establecer en grupo estándares de codificación para poder llegar a una solución clara y con buena calidad de código.
Algunos estándares establecidos fueron:

- El código y commits se escribe en inglés.
- La interfaz y mensajes a los usuarios se escriben en español.
- Los atributos privados de las clases empiezan con '_' Ej: _nombreAtributo.
- Los métodos, clases y atributos estáticos (sin importar su visibilidad) usan PascalCase.
- En el resto de casos se utiliza camelCase.
- No utilizar el "this." para referirse a atributos al menos que sea:
    - Para evitar errores de compilación
    - Para ayudar a la comprensión del código
    - Para referirnos a una property adentro de una misma clase