Programación de redes

Obligatorio 1

Gastón Landeira **- 238473**
Nicolás Rosa **- 225609**

Universidad ORT Uruguay

Profesor: Luis Barrague

Gastón Landeira - 238473
Nicolás Rosa - 225609

Índice

Notas:

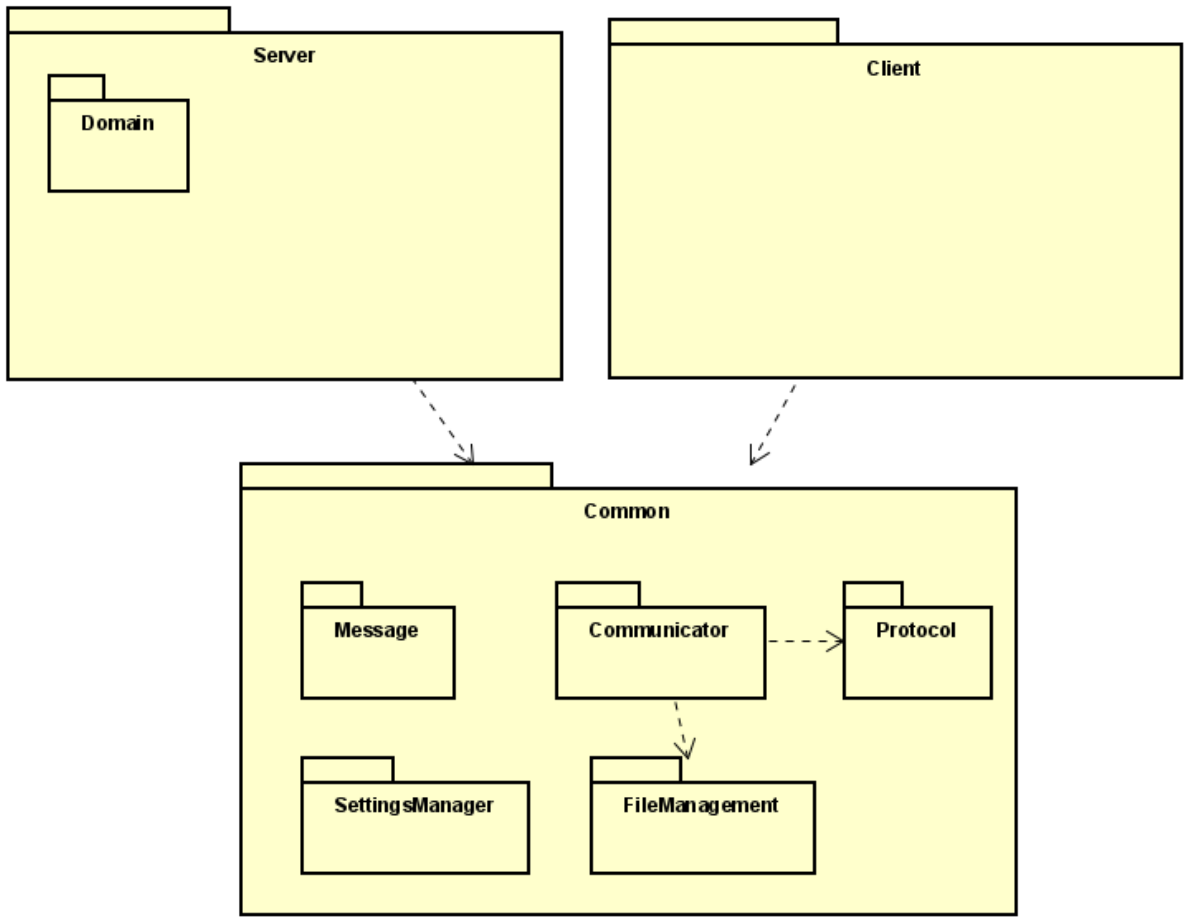
Todo el proyecto se realizó en Github, se puede encontrar en el siguiente link:
https://github.com/nicoRosa18/Obligatorio-1-Programacion-de-redes

Se adjunta una carpeta llamada Diagramas con los archivos de los diagramas completos para una mejor visualización. Estos no se agregan aquí por el tamaño de los mismos. Los archivos .svg se pueden abrir en el navegador.

Descripción de la arquitectura.

Dividimos el sistema en 3 subsistemas/proyectos: Client, Server y Common. El primero tiene el propósito de brindar al usuario las funciones del servidor. Se limita a actuar de interfaz empleando la implementación de la conexión cliente-servidor. Esto se consigue no acoplando lógica de dominio en él más que validaciones básicas para evitar comunicación innecesaria. Continuando con los subsistemas se encuentra el Server, este implementa todas las funcionalidades solicitadas y las expone hacia los clientes que se le conecten. Dentro de él se pueden encontrar dos grandes partes: una orientada a la lógica y requisitos funcionales encerrada en su propio paquete llamado Domain, mientras que otra orientada a la conexión y manejo de conexiones de clientes. Por último se creó Common como paquete auxiliar a ambas partes. Common cuenta con clases que son de utilidad para la comunicación en general. Aquí se encuentran tanto el protocolo de comunicación como el manejo de archivos y los mensajes del sistema.

Diagrama de paquetes:



Gastón Landeira - 238473
Nicolás Rosa - 225609

Documentación de diseño detallada de cada componente y justificación del diseño.

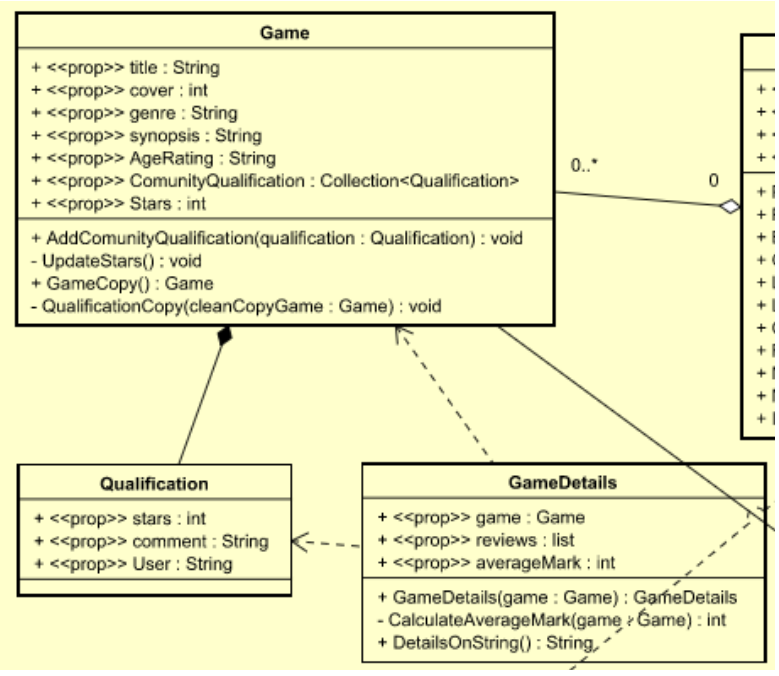
Server:

Paquete Domain

Este paquete se encarga de toda la lógica del sistema así como de almacenar los datos. La decisión de no separar estas dos grandes funcionalidades fue porque no se requirió persistencia, habilitando a que nuestra lógica de dominio se acoplara con el dominio en sí, decisión que se cuestionará más adelante con el problema de mutua exclusión. A continuación se hará una explicación clase por clase de las funcionalidades de las mismas.

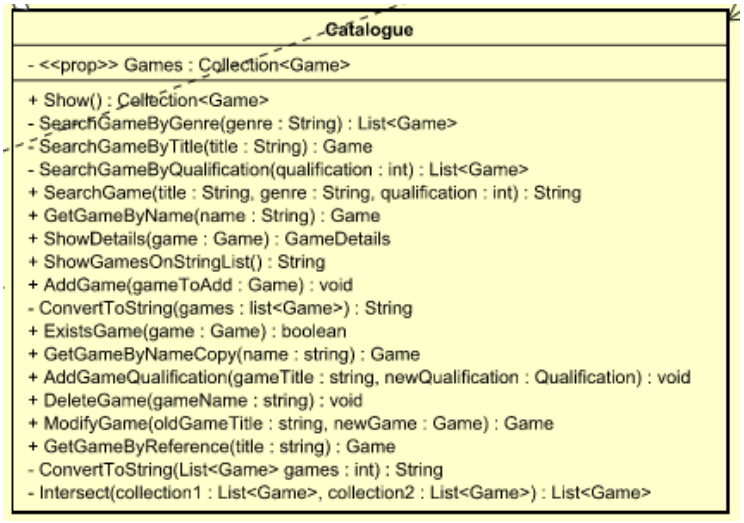
Comenzaremos en orden de menor a mayor dependencia por la clase Qualification. Esta fue creada para encapsular todos los campos de una calificación o review en un solo objeto. Cuenta con la cantidad de estrellas, un breve comentario y el nombre del usuario que la realiza, no una referencia al usuario porque no se permitirá esa navegabilidad.

La clase que se encarga de Qualification es la clase Game, la cual es responsable de almacenar todos los datos requeridos para un juego así como también almacenar estas reviews. Esta clase se vio perjudicada por la no previsión de la mutua exclusión e implementa un método que copia sus datos y los pasa como copia, no como referencia.



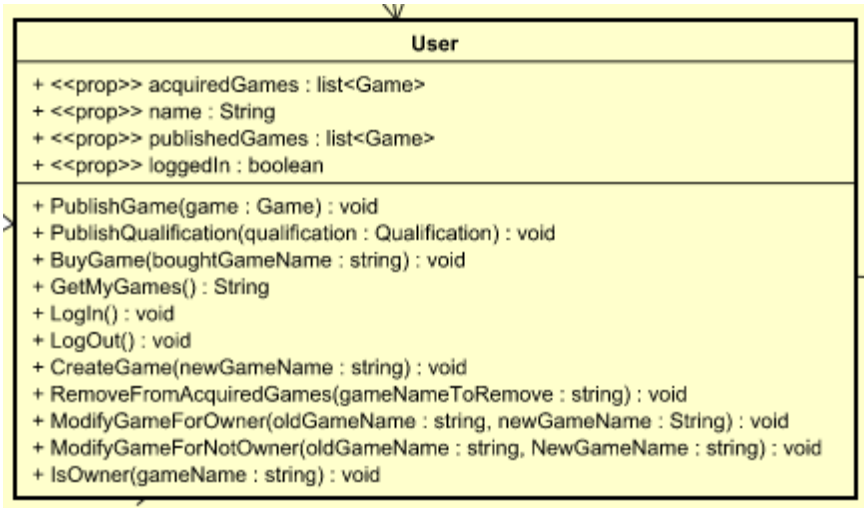
Con el requisito de enviar los detalles de un juego junto a su calificación y sus reviews se creó la clase GameDetails. Una clase de corta vida que solo cuenta con métodos previos para el cálculo de uno de sus atributos. Al comienzo del proyecto se planteó algún tipo de extensibilidad de esta clase para que pueda ser utilizada en el envío de otro formato de datos pero el protocolo ya por sí solo plantea el formateo a bytes de los datos. Con el tiempo la idea fue quedando obsoleta, solo se le permitió quedar por su practicidad y por su lógica al momento de su creación. Esta clase es completamente refactoreable a permanecer en Game. En síntesis, GameDetails calcula a partir de un juego y sus reviews la calificación promedio del mismo y presenta toda la información requerida de manera detallada y prolija, esto último a través del método DetailsOnString().

La lista de todos los juegos se encuentra en la clase Catalogue. La clase surge de la necesidad de ubicar la lógica de los juegos, y qué mejor clase que la encargada de almacenarlos. En específico es



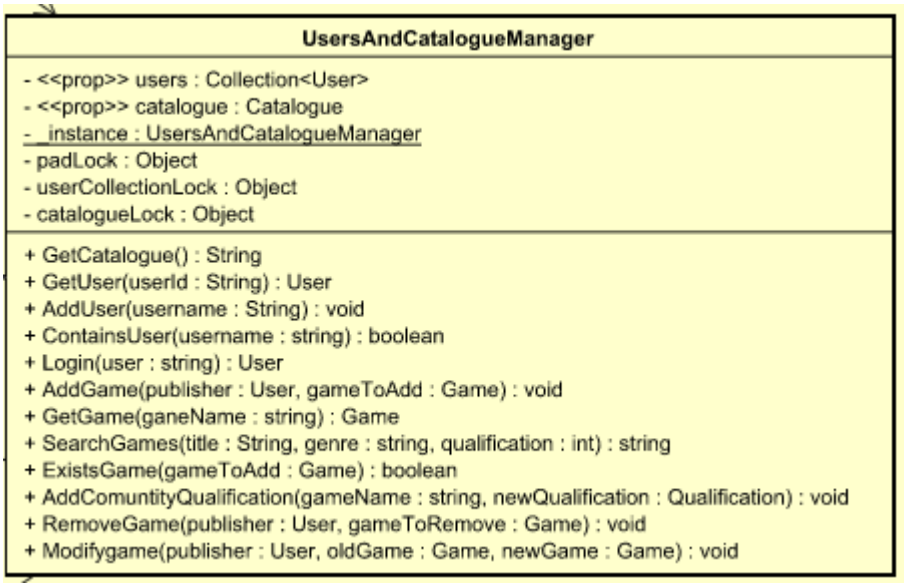
responsable por la búsqueda de los juegos por los distintos campos (título, género y/o estrellas). Se destaca el y/o porque se permite buscar cada una por separado o una combinación entre género y estrellas. La búsqueda de un juego por título devolverá el juego sin importar los campos de género o estrellas, ya que el título es el identificador del juego. También es mencionable que se encarga de gestionar la lista de los juegos por si ocurre una modificación o eliminación. En el caso de la modificación fue necesario utilizar el juego por referencia, es por esto que existe el método GetGameByReference(). Excluyendo este caso, todos los demás métodos utilizan la copia del objeto. Además, por la ley de Demeter (que se siguió durante la codificación del proyecto), se extiende la funcionalidad del juego de agregar una calificación o de mostrar, permitiendo una clara visibilidad de la responsabilidad de las clases y de las dependencias.

En paralelo al catálogo de juegos se ubica la clase User. Esta contiene los datos del usuario así como también se encarga de almacenar una pseudo referencia por un lado a los juegos comprados y por otro lado a los juegos publicados por este. Pseudo referencia porque guarda en una lista de strings nada más que los id de los juegos, que en este caso es su título. También se encarga tanto de los métodos modificar y remover como de si es el publicador del juego. Además, con la intención de que solo se permita un usuario por cliente, cuenta con una variable booleana que indica el uso del mismo. Esta pequeña decisión tiene un gran impacto en el sistema porque libera de la encapsulación, causada por la mutua exclusión, al usuario en sí. Permitiendo su paso por referencia a la lógica del dominio al no ser un recurso compartido.

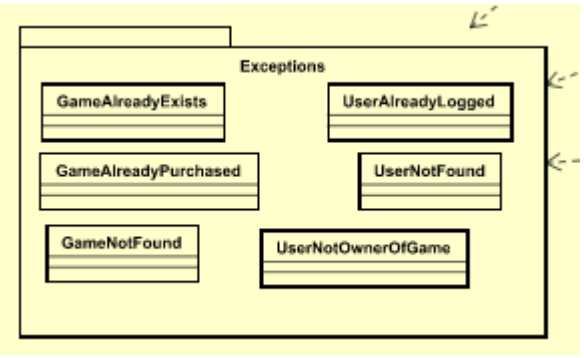


Lo que sí es compartido y tiene que ser controlado por mutua exclusión es la lista de usuarios y también el catálogo de juegos. Ambos tienen que formar parte de una clase compartida por todas las sesiones conectadas. Es por esto que se creó la clase UserAndCatalogueManager. Una clase a la cual le implementamos el patrón Singleton por la razón anterior.
Para acceder a esta clase y controlar su creación se utiliza un lock. Y utiliza otros dos locks más para controlar el acceso al catálogo y a la lista de usuarios, protegiendo la mutua exclusión. Fue por el problema este mismo problema que se decidió que todo acceso de la sesión a la lógica pasase por esta clase, y que todos los elementos devueltos por ella sean copias y no referencias (menos el caso

del usuario explicado antes). Una mejor solución hubiese sido la creación del catálogo y la lista de usuario sin lógica alguna en el paquete de dominio (pasando en un acceso de datos la lista de objetos por copia), y otro paquete llamado lógica de dominio con los métodos que utilizan al dominio mediante el acceso a los datos. De esta manera se hubiese logrado que cada paquete cuente con su propia responsabilidad, desligando y haciendo la aplicación más extensible.
Se tomó la decisión de solo mantener los tres locks mencionados para mantener la responsabilidad de esto encapsulada en UserAndCatalogueManager. Es por esto (y por la ley de Demeter) que se pueden mapear linealmente todos los métodos de la clase a los requisitos del obligatorio. Todo pasa por esta clase pero, se obtiene un mayor control a costa de perder eficiencia. Ya que por ejemplo se tiene que bloquear el catálogo para buscar un juego cuando lo óptimo sería que la lista fuese copiada y que el lock solo se realice para copiar la lista y, proseguir buscando el juego en la lista copiada.



Como punto aparte se manejó la creación de excepciones que manejamos más tarde en la sesión, estas se encuentran en el paquete Exceptions dentro del namespace ServerExceptions. Todas se crearon con el objetivo de avisar a la sesión que debía comunicarse con el cliente por información errónea ingresada.



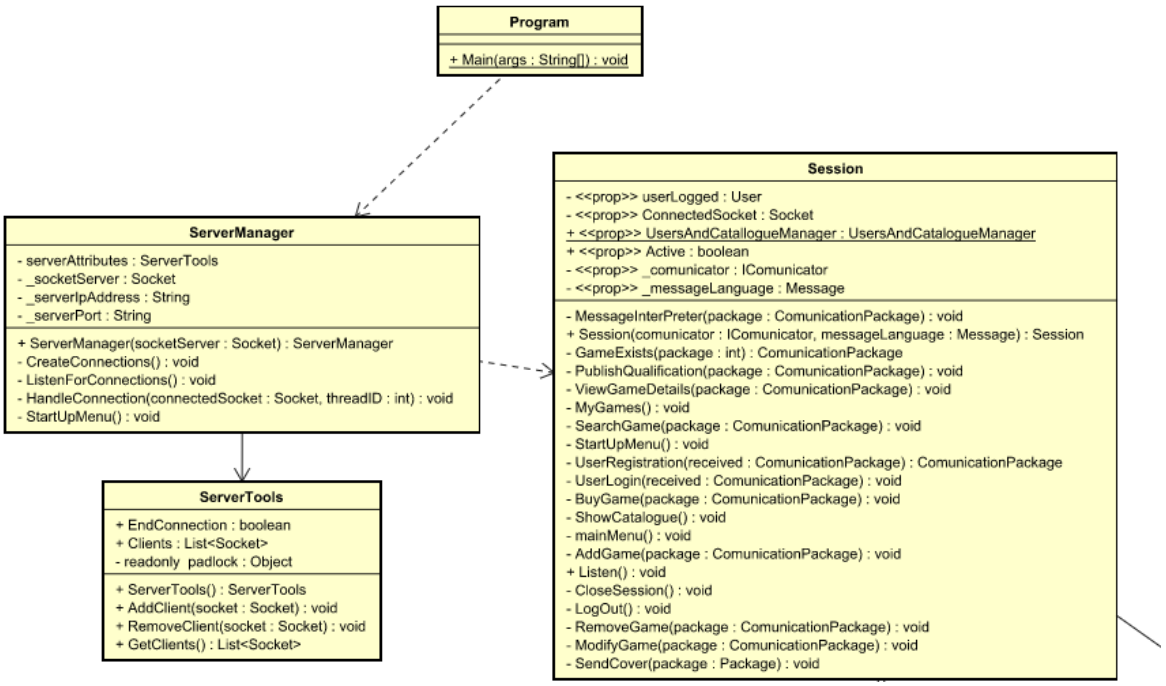
Clases de Server

Por fuera de Domain tenemos 4 clases: Program, ServerTools, ServerManager y Session.
Comencemos a describirlas en el orden que se ejecutan.

Primero tenemos la clase Program que tiene el método main, su única responsabilidad es inicializar el sistema y crear un objeto de tipo ServerManager.
Continuando con ServerManager, esta es la clase encargada de crear el socket de escucha, recibir nuevas conexiones y asignarles un hilo, para así poder cumplir con el requisito de que la aplicación sea Multithreading y pueda manejar varias conexiones en simultáneo. También se encarga de cerrar el servidor y las conexiones establecidas. Cuando el método listenForConnections() recibe una conexión de parte de un cliente, se le asigna un nuevo hilo y en ese hilo se crea un objeto de tipo Session. Mientras la sesión esté activa o el servidor se mantenga conectado, se queda iterando en el método Listen del objeto recién creado. La persona que corra el servidor tiene la posibilidad de cerrarlo. Si decide darle uso a esta funcionalidad inmediatamente se cierran todas las conexiones establecidas y se termina el programa. Aquí es donde la clase ServerTools es de ayuda. Esta clase solo se compone de un booleano que indica el cierre del servidor y una lista de sockets conectados al servidor. Al finalizar el server, el booleano del paquete de ServerTools que fue instanciado en el thread principal de ServerManager y pasado por referencia a cada thread se vuelve verdadero, provocando que todas los sockets de la lista de sockets que se encontraban en el paquete se cierren. En principio esto hubiese provocado un error porque se intenta cerrar un socket que está esperando una conexión (el del server) pero, por una jugada de este recibe una conexión y después se cierra satisfactoriamente. La jugada consiste en provocar un fake socket al cerrar el servidor, solucionando el problema. Se destaca el uso de los lock en la lista de sockets de ServerTools para permitir el paso de la lista sin errores (solución de un problema de mutua exclusión).

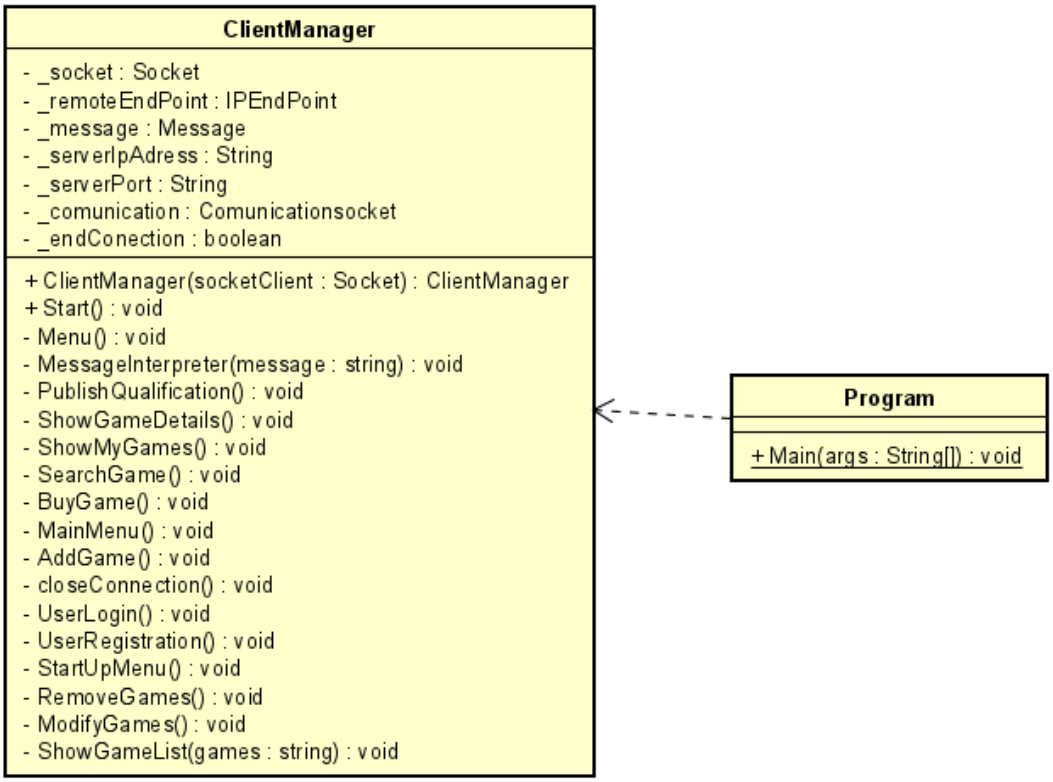
Llamamos Session a la clase que actúa como menú de un cliente en el servidor. Esto es porque le asignamos un usuario a ella, asignándole un estado a la clase del servidor. Esto con la cualidad mencionada de que tiene un principio y un final, genera una idea de sesión.
La clase Session se encarga de exponer las funcionalidades del servidor al cliente, así como también recibir las peticiones del cliente y generar una respuesta para enviarla. Esta clase tiene una property de tipo UserLogged (el mismo mencionado al comienzo del párrafo), que sirve para evitar enviar un token de usuario cada vez que se realiza una petición desde el cliente hacia el servidor. La property _comunicator del tipo IComunicator nos sirve para comunicarnos con el cliente, tanto para recibir como para enviar mensajes. Session usa La property UsersAndCatalogueManaguer para todas las acciones que le indique el cliente. Por último quedan las prop Active y _messageLanguage, la primera nos indica si la sesión aún se encuentra activa, o si fue cerrada desde alguno de los extremos, y messageLanguage nos proporciona los mensajes del servidor que son de utilidad para enviarselos al usuario.
Como se dijo antes, hay tantas Session como conexiones de clientes y un solo UserAndCatalogueManager. Es por esto que siempre se tiene que tener en cuenta que los objetos que se observan allí son copias de los elementos reales(menos User) porque si fuesen los mismos objetos (por referencia) cada vez que se intenta acceder al mismo juego se caería el programa. De esta manera se aíslan responsabilidades.

Especificando en el código, se decidió tener un switch al cual se entra en cada comienzo de acción por parte del cliente. Dependiendo del comando, se realiza el procedimiento. Así se pueden dividir fácilmente las acciones aunque la clase haya quedado de una extensión más larga de la deseada. En otra oportunidad se podría haber separado el switch en otra clase y los métodos en otra más, y haberlos instanciados en Session.
Además, al ser la última línea antes del cliente se decidió controlar las excepciones aquí. La mayoría hacen triggerear otro curso de acción donde se manda otro mensaje al cliente pero, en los casos de las recepciones de archivos lo que podría salir mal es bastante amplio, por lo que se optó por controlar cualquier caso de excepción.



Client:

Pasando al subsistema del cliente: Para este subsistema decidimos implementar un subsistema “semi inteligente”, que conozca las acciones que puede realizar el servidor y que datos se necesitan. Elegimos esta decisión de diseño para no sobrecargar de comunicaciones innecesarias al servidor, de modo tal que si por ejemplo queremos agregar un juego, el cliente es el encargado de recoger todos los datos necesarios para esta acción, cuando los obtiene todos se comunica con el servidor enviando el comando con la acción requerida y los datos para esa acción.
También agregamos varias validaciones básicas, como lo son chequear si se ingresa un campo vacío, o si es el tipo de valor esperado (un int en caso de un número).
Cada vez que se inicia este subsistema, partiendo desde la clase Program, se crea un objeto del tipo ClientManager, el constructor de esta clase es la encargada de crear los elementos necesarios para la conexión con el servidor, como también seter a los atributos del objeto mismo los atributos que utiliza para la comunicación (_comunicator) y el idioma de los mensajes del sistema. La decisión de quitar los mensajes del cliente y en su lugar usar un objeto que los contenga fue tomada para favorecer la extensión del idioma como la calidad de código, manteniéndolo más limpio de esta manera.
Luego de que en Main de Program se crea el objeto, se invoca al método Start() del mismo. Este se encarga de inicializar la conexión y llamar al método menú.Este último se encarga de pedir el startup menu al servidor y mostrarlo por consola cuando lo recibe. Además, en este método, se encuentra el bucle principal de la aplicación cliente, mientras nosotros no cerramos la conexión (al ingresar exit), o se cierre remotamente, el sistema va a seguir en el while , leyendo la consola, e invocando al método MessageInterpreter con el string que acaba de ser ingresado en la consola.
Este método, análogo al que encontramos en el servidor, este, cuando nosotros ingresamos el número de la funcionalidad que queremos utilizar, se encarga de redirigirlos al método que gestiona la misma. Ese metodo conoce los datos que necesita el cliente para completar correctamente la funcionalidad, por lo que reúne todos los datos, los valida, y los envía al cliente mediante el método SendMessage del atributo _ccomunication , este método recibe como parámetros un int,el comando de la funcionalidad, que lo obtenemos desde la clase Command Constants, que se encuentra en el package Common , y un string que son los datos recopilados en por el método. Como ejemplo si queremos registrar un usuario, el método RegisterUser() , va a pedir el nombre de usuario, y va a comunicarse con el servidor a través de _comunicator, enviando el parámetro CommandConstats.RegisterUser y el nombre de usuario el cual queremos registrar. Luego de enviar los datos, utilizamos el método ReceiveMessage() de _communication para conocer la respuesta del servidor, este método nos retorna un string al cual lo mostramos en consola con Console.WriteLine(). Pasando al tema de la complejidad de extender el sistema con nuevas funcionalidades, consideramos que este no sería un gran inconveniente, ya que basta con agregar la constante de la funcionalidad al switch que se encuentra dentro de MessageInterpreter y crear un método que se encargue de reunir o pedir lo que sea necesario para llevar a cabo dicha funcionalidad.

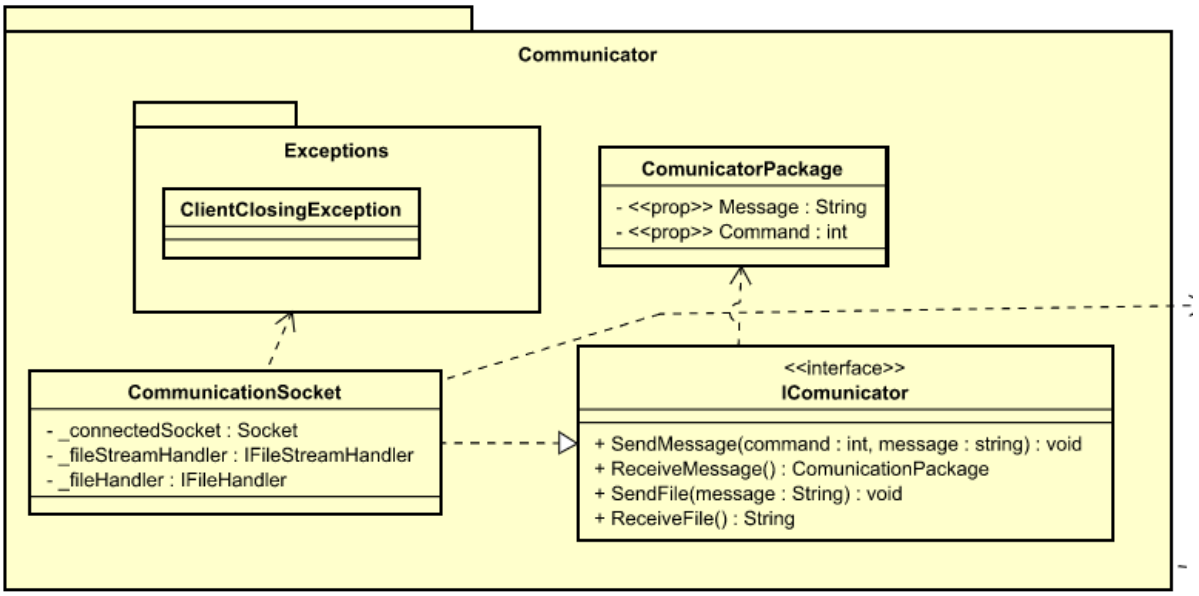


Common:

El proyecto Common consta de clases auxiliares que comparten tanto el cliente como el servidor. Cuenta de paquetes dedicados a la comunicación como Communicator o Protocol, un paquete dedicado a la escritura y lectura de archivos llamado FileManagement, otro paquete dedicado a los string de mensajes que se muestran por consola al cliente llamado Message y, por último, un paquete dedicado a la lectura de los archivos app.config nombrado SettingsManager. Se explicará cada uno en detalle a continuación.

Paquete Communicator

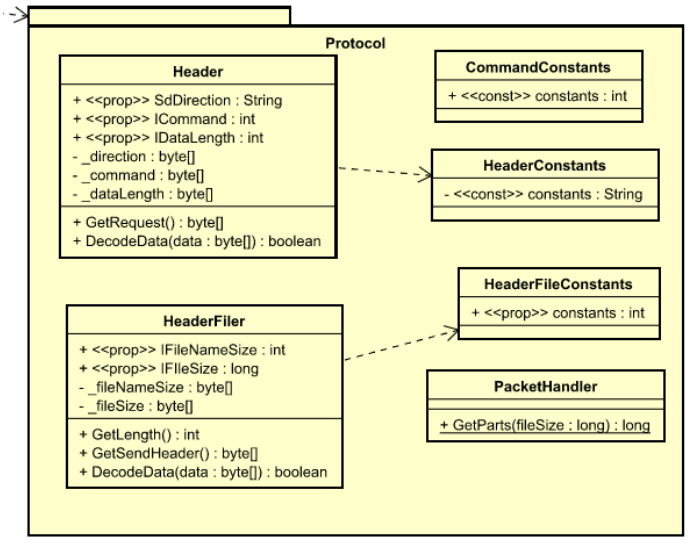
Cuenta con una interfaz, su implementación y una clase que actúa como la pieza de portación de datos. El cliente y el servidor ambos usan la interfaz, permitiendo la extensibilidad de la comunicación por si se desea hacer de otra manera. La implementación de la interfaz llamada CommunicationSocket vincula tanto el manejo de archivos como el protocolo utilizado con el manejo de datos a través de sockets ya inicializados en Client o Server. Desde afuera la clase es muy simple y sencilla de usar, escondiendo mucha complejidad en su interior. Cuatro métodos básicos que son enviar, recibir mensajes o recibir, enviar archivos. Si se entra en el interior de la clase se puede observar como utiliza el header de los protocolos para enviar o recibir datos en bytes, los cuales convierte o revierte a string una vez finalizado el proceso. Primero se envía/recibe el protocolo, y acto siguiente se recibe/envía la información del mensaje en sí.



Se menciona que al recibir un archivo con el mismo nombre, este es eliminado y reemplazado por el nuevo archivo, no se intenta seguir escribiendo información dentro de él.
En caso de que no se halle la imagen en el path guardado al momento de enviar el archivo en Session, se provee una imagen a modo de default. A esta creación se la intentó aislar lo más posible de Session para respetar el principio Single Responsability pero, al ser un caso tan específico,se tuvo que usar el método de creación que se encuentra en FileDefaultCover del paquete FileManagement. Para futuras versiones se creará otro paquete en Common que maneje esta situación.
Además, con el objetivo de no interferir entre sistemas operativos por sus diferentes configuraciones de paths, a los archivos guardados en el servidor y cliente se les debe especificar el path deseado de guardado en los App.Config de los cuales se hablará más adelante.
Por último, con la incorporación de la modificación y borrado de juegos, se cambia o borra satisfactoriamente el string del path de la carátula del juego pero no se elimina el archivo mismo del proyecto. Permanece por si no se elimina con intención para recuperar más tarde. Si se desea su eliminación total, se debe hacer manualmente desde el explorador de archivos.

Paquete Protocol

Las clases importantes a observar aquí son Header, HeaderFile y CommandConstants, las demás son auxiliares.
Header define como se enviarán y recibirán los mensajes, o sea el protocolo que se utilizara en el mismo. Codifica y decodifica sus datos. Nosotros elegimos un protocolo orientado a caracteres Implementado sobre TCP/IP. La regla (que se aplica tanto en Header como HeaderFile) es que los valores irán a la derecha, y si sobran números de espacio se rellenará con ceros a la izquierda. Los valores en sí del Header se encuentran en la clase HeaderConstants, para su fácil cambio de



ser necesario. Los valores son de la siguiente manera:

Nombre del campo	Direction	Commands	Data length	Data
Posible Valor	Res/Req	00-99	0000-9999	variable
Largo en el header	3	2	4	se manda inmediatamente después.

En nuestro caso, como se puede observar en CommandsConstants se tienen 24 números reservados para comandos. Igualmente se pueden utilizar hasta 99 comandos. La idea es que el cliente mande un número de comando, y si la solicitud al servidor se realiza correctamente, se devuelva el mismo número de comando. En otro caso se especifica el error mediante otro número de comando distinto y el mensaje.

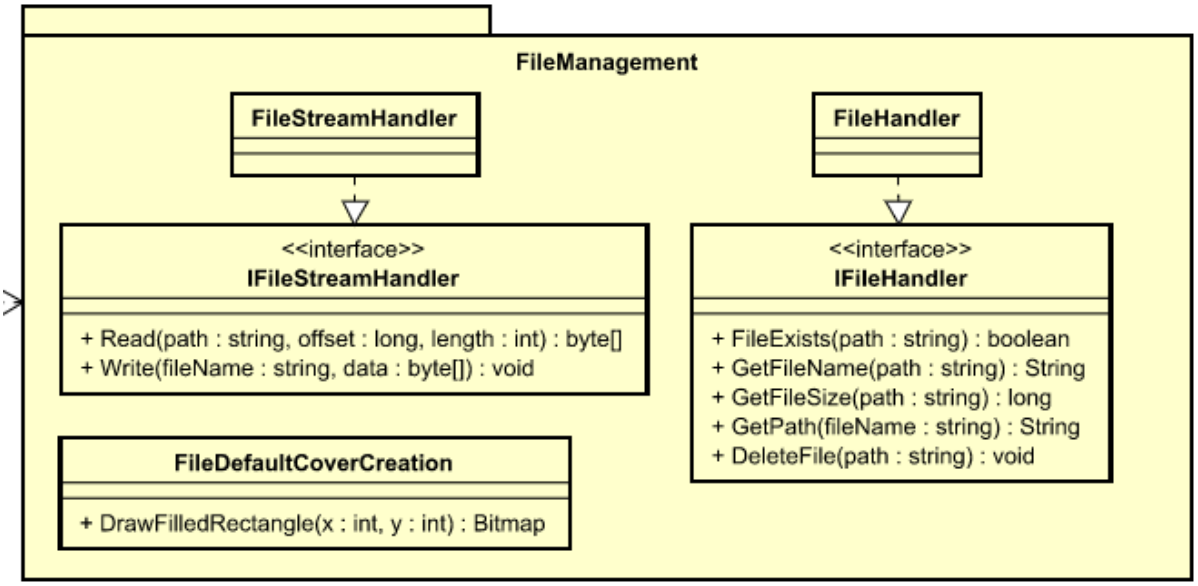
En la misma línea de pensamiento, HeaderFile se especializa en enviar datos más grandes que en este caso utilizamos para imágenes, pero se podría usar para cualquier tipo de datos.
Los valores son de la siguiente manera:

Nombre del campo	FileNameSizeLength	FileSizeLength	MaxPacketSize
Posible Valor	0000-9999	00000000-99999999	0-32768
Largo en el header	4	8	Paquete que se envía de a partes después del header

La diferencia con Header esencial es la cantidad de datos que maneja HeaderFile. Es tanto así que se debe separar en paquetes a enviar/recibir. El método de separación de paquetes de datos se encuentra en PacketHandler.
El concepto es primero enviar el HeaderFile, acto siguiente el nombre del file, y por último los datos del file.

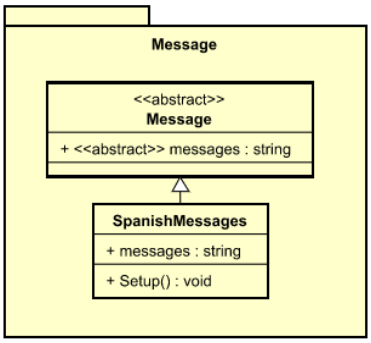
Paquete FileManagement

Paquete dedicado exclusivamente a los archivos. Cuenta con dos interfaces y sus implementaciones. La interfaz IFileStreamHandler con su implementación FileStramHandler, y la interfaz IFileHandler con su implementación FileHandler.
El primero se encarga de la lectura y escritura pura de los archivos. El segundo provee de métodos que permiten el manejo de archivos.



La idea es que estas clases se encuentren ocultas y apartadas de la solución en sí. Sus métodos son útiles, pero no dejan de ser una herramienta para el manejo de archivos, solo con sus métodos deberíamos ser capaces de manejarnos. Además, la implementación de las interfaces permiten la extensibilidad del código sin impactar en la CommunicationSocket que la utiliza.
Una posibilidad que se planteó durante la codificación es la de solo permitir archivos .jpg (imágenes) pero no se llevó a cabo. Concluimos que el formato de los archivos no interesaba en lo más mínimo a la aplicación y, si en un futuro se desea restringir esto, se implementara en este paquete.
El paquete también incluye una clase llamada FileDefaultCover implementada más tarde al codificar. Esta clase se encarga de contener el método para diseñar la imagen por defecto que enviará el servidor al cliente por si no se encuentra la imagen del juego en cuestión. Aunque sea la única clase del paquete que no está oculta al Server, se decidió depositarla aquí porque trabaja con archivos a bajo nivel, y dejarla en Session hubiese sido demasiado acoplamiento.

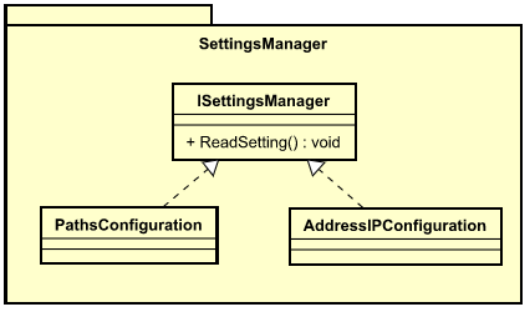
Paquete Message



Para mejorar la calidad del código se decidió sacar todos estos mensajes de la clase Session, y dejarlo como constantes en la clase de tipo Message. Además, para que el idioma sea extensible sin tener que modificar el código, se hizo que los atributos de Message (los mensajes) sean sobreescribibles en clases hijas, así, podemos tener clases hijas que implementen lenguajes específicos y cambiar el idioma de los mensajes del servidor sin tener que modificar otras clases, gracias al polimorfismo. La solución elegida para los mensajes del sistema se adapta perfectamente al principio Solid *Liskov Substitution Principle* ya que en el caso de que la clase Message no sea abstracta y decidamos que contenga el idioma default, por ejemplo , mensajes en inglés, podemos reemplazar la clase por cualquiera de las subclases que hereden de ella.

Paquete SettingsManager

Este paquete cuenta con una interfaz ISettingsManager y sus implementaciones AddressIPConfiguration y PathsConfiguration. Básicamente es la configuración para poder leer de los documentos App.Config que se encuentran tanto dentro del Server como del Client. Por lo tanto si se llega a dar un release de la app, el código no se toca en ningún momento, solo el App.Config para cambiar la dirección IP y puerto del servidor de donde se esté levantando la aplicación.
Además, se deberá cambiar la dirección de los path en donde se desean guardar los archivos de las carátulas de los juegos. En el caso de App.Config del Server se puede cambiar si se desea el nombre de la imagen por defecto generada para las carátulas de los juegos cuando no se encuentra la guardada.
Las dos clases que implementan la interfaz cuentan con el mismo código, esto es fácilmente cambiable a una sola clase. Sin embargo por falta de tiempo se decidió dejar de esta manera aunque implique la repetición de código.
Cabe destacar que existen dos App.Config en la aplicación para cada proyecto. Se podría haber hecho uno en común en Common pero, si se llegan a separar los proyectos en diferentes computadoras, sería necesario crear uno individual para cada uno. A efectos de este proyecto, no haría diferencia.



Documentación de mecanismos de comunicación de los componentes de su solución.

Diagrama de secuencia de inicio del cliente:

Aca podemos apreciar de mejor manera lo comentado anteriormente sobre clientManager, este se encarga de crear todos los elementos necesarios para la conexión con el servidor, y luego el método start establece la conexión, en caso de que no haya en éxito en esta operación , el cliente procede a cerrarse.

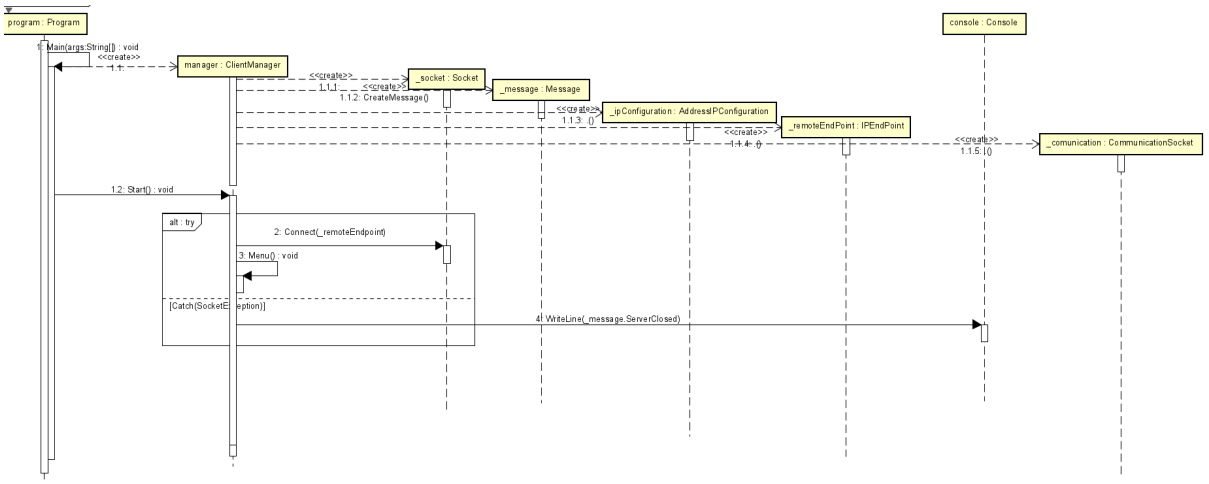


Diagrama de secuencia del método Register user en Cliente.

Se escogió este caso de uso (registro de un usuario), para demostrar cómo es la interacción entre el cliente y el servidor, todas las demás funcionalidades son análogas a ésta, con mayor o menor complejidad. Vemos que primero pide al cliente el nombre de usuario que quiere registrar, chequea que no sea nulo o vacío, este chequeo básico nos ayuda a no crear comunicación innecesaria hacia el servidor, ganando en recursos y eficiencia. Si pasa esta verificación, se utiliza el método SendMessage de _comunicator para enviarle al servidor la constante con la acción a realizar y los datos correspondientes, en este caso el nombre de usuario. Una vez recibida la respuesta del servidor la muestra por pantalla y vuelve al menú de inicio.

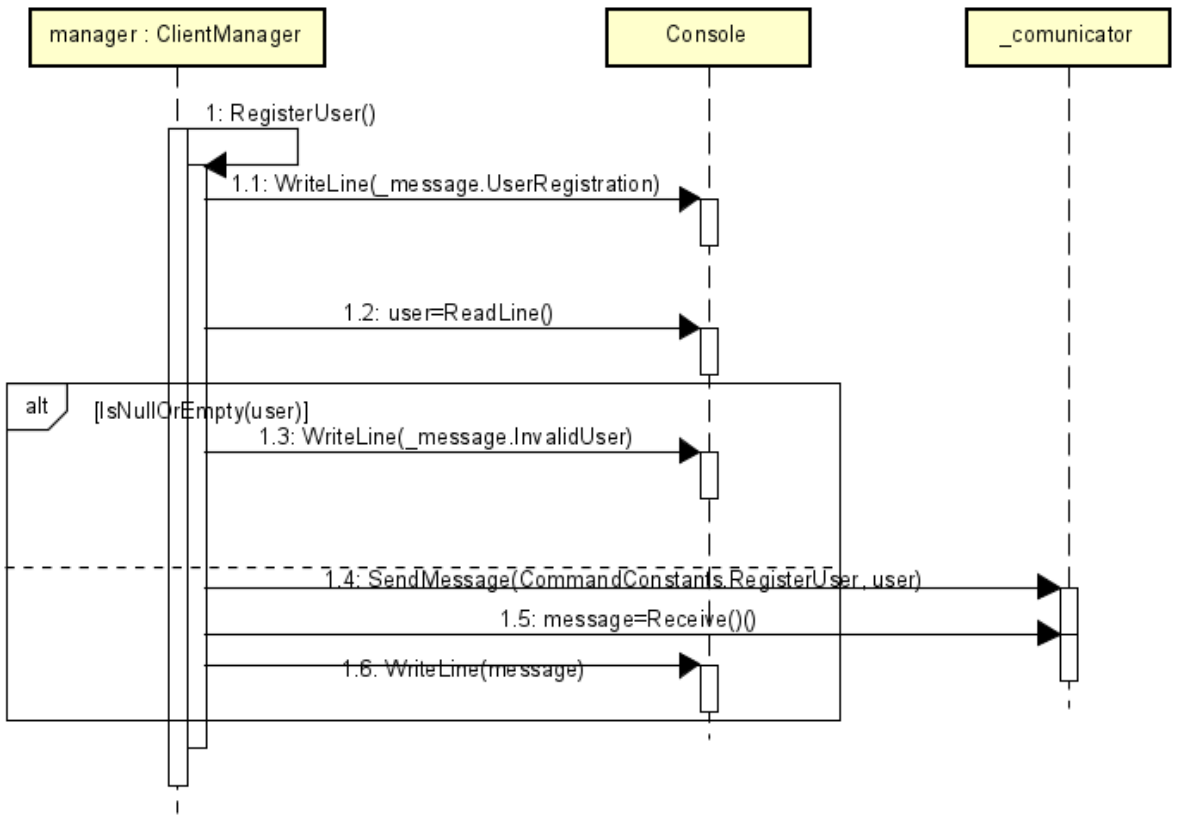
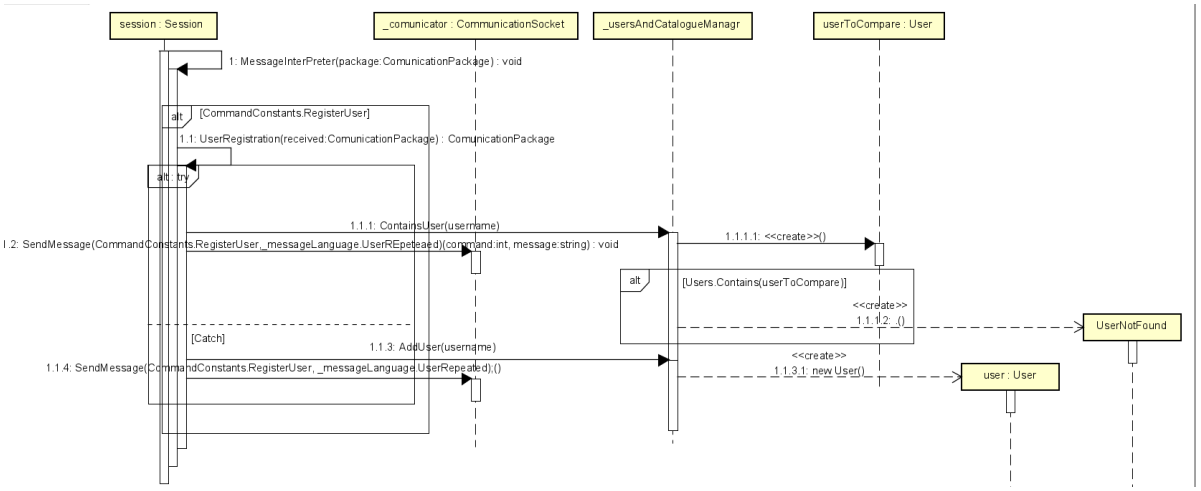
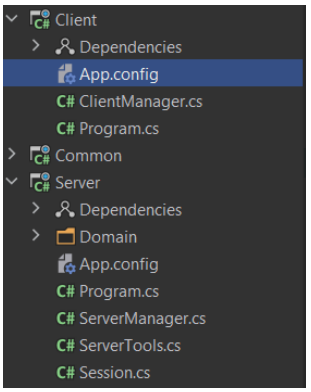


Diagrama de secuencia del servidor para el flujo de registro de usuario.

Continuando con la acción solicitada por el cliente, creamos el diagrama de secuencia en el servidor para el caso de uso de registro de usuario. Decidimos no agregar toda la parte de la iniciación del servidor y la aceptación de una nueva conexión del cliente para no agregar complejidad al diagrama. El servidor , en sesión, queda escuchando nuevos mensajes de parte del cliente, una vez recibe un mensaje, se lo pasa al método MessageInterpreter, la cual , a través del comando recibido en el paquete , llama al método correspondiente. Una vez dentro del método UserRegistration, se verifica si el nombre de usuario ya existe, de serlo así, se le envía al cliente un mensaje de respuesta indicando. Si no existe se lo agrega a la lista de usuarios que está en la clase UsersAndCatalogueManager. Luego de agregado se le informa al cliente del resultado de la acción.



Guía de Uso:



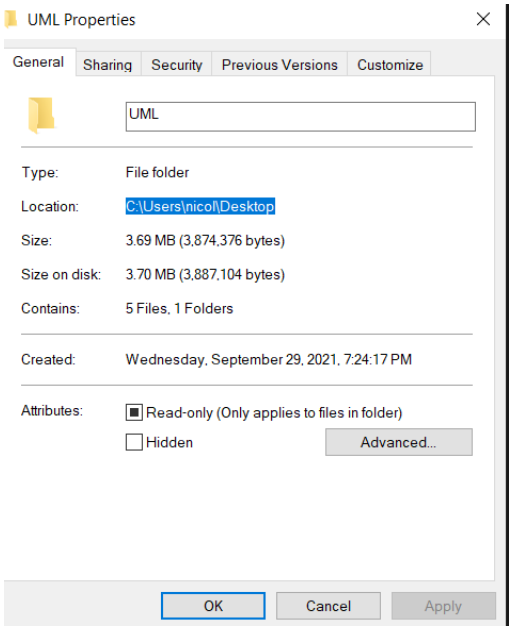
Lo primero que debemos hacer es localizar los archivos App.config tanto en el package Client, como en Server.

Buscamos en los archivos dentro del tag App.Config la key ServerIpAdress.

Cambiamos el valor de la key mencionada en ambos archivos por la ip de nuestra PC.

En windows podemos conocer la ip accediendo a la command Prompt e ingresando el comando ipconfig.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <appSettings>
        <add key="ServerIpAddress" value="192.168.1.5"/>
        <add key="ServerPort" value="30000"/>
        <add key="DefaultPath" value="defaultImage.png"/>
        <add key="CoversPath"  value="C:\Users\nicol\Desktop\coversServer\"/>
    </appSettings>
</configuration>
```



Por último debemos cambiar el valor de la key “CoversPath” para establecer dónde queremos que se almacenen las carátulas de los juegos. Para esto vamos a cualquier elemento que tengamos en el escritorio, le damos click derecho y propiedades.
Se nos abrirá una pequeña pestaña con información acerca del archivo y la ruta en la que se encuentra, la copiamos y reemplazamos en el valor de la key. Una vez pegada agregamos ***\coversServer*** **y** ***\coversClient*** respectivamente.

Una vez realizados los cambios, lo guardamos y simplemente ejecutamos primero el servidor, y luego el cliente. Luego de esto se guiará intuitivamente gracias a los mensajes desplegados en la consola.

Si se desea salir del sistema sin cerrar la consola se tiene la posibilidad de escribir exit en el cliente dentro del menú principal, o si se desean cerrar todas las conecciones, exit en la consola del servidor.