Universidad ORT Uruguay
Facultad de Ingeniería

Programación de redes

Sistema Vapor

Nicolás Gibbs - 227347
Germán Konopka - 238880
Martin Robatto - 240935

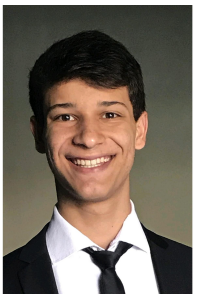
Tutores: Luis Barragué y Roberto Assandri

2021

Autoría

Nro. Estudiante	240935
Nombre:	Martín
Apellido:	Robatto
Grupo / Turno:	M6A

Nro. Estudiante	238880
Nombre:	German
Apellido:	Konopka
Grupo / Turno:	M6A



Nro. Estudiante	227347
Nombre:	Nicolás
Apellido:	Gibbs
Grupo / Turno:	M6A

Índice

Descripción General

Nuestro trabajo consistió en la creación de un sistema basado en C# que consta de dos aplicaciones, cliente y servidor, que presentan funciones tales como gestión completa de juegos, ver catálogo y adquirir juegos, publicar reseñas, entre otras. La aplicación cliente se encarga de las interacciones de los usuarios con el sistema, mientras el servidor es quien atiende las peticiones de los clientes y administra los recursos del sistema relacionados a los servicios brindados.

Cabe destacar que en esta oportunidad no se cuenta con persistencia de datos mediante el uso de una base de datos, sino que se recurre al almacenamiento en memoria para los datos ingresados al sistema. En otras palabras, se busca comunicar que una vez cerrada la aplicación servidor, los datos ingresados se eliminaran.

En resumen, se trata de una arquitectura cliente- servidor, donde el cliente realiza las peticiones que le correspondan y el servidor atiende las peticiones para los recursos que administra.

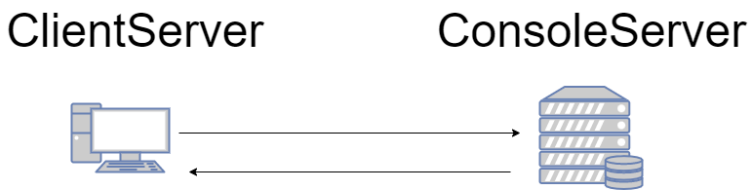
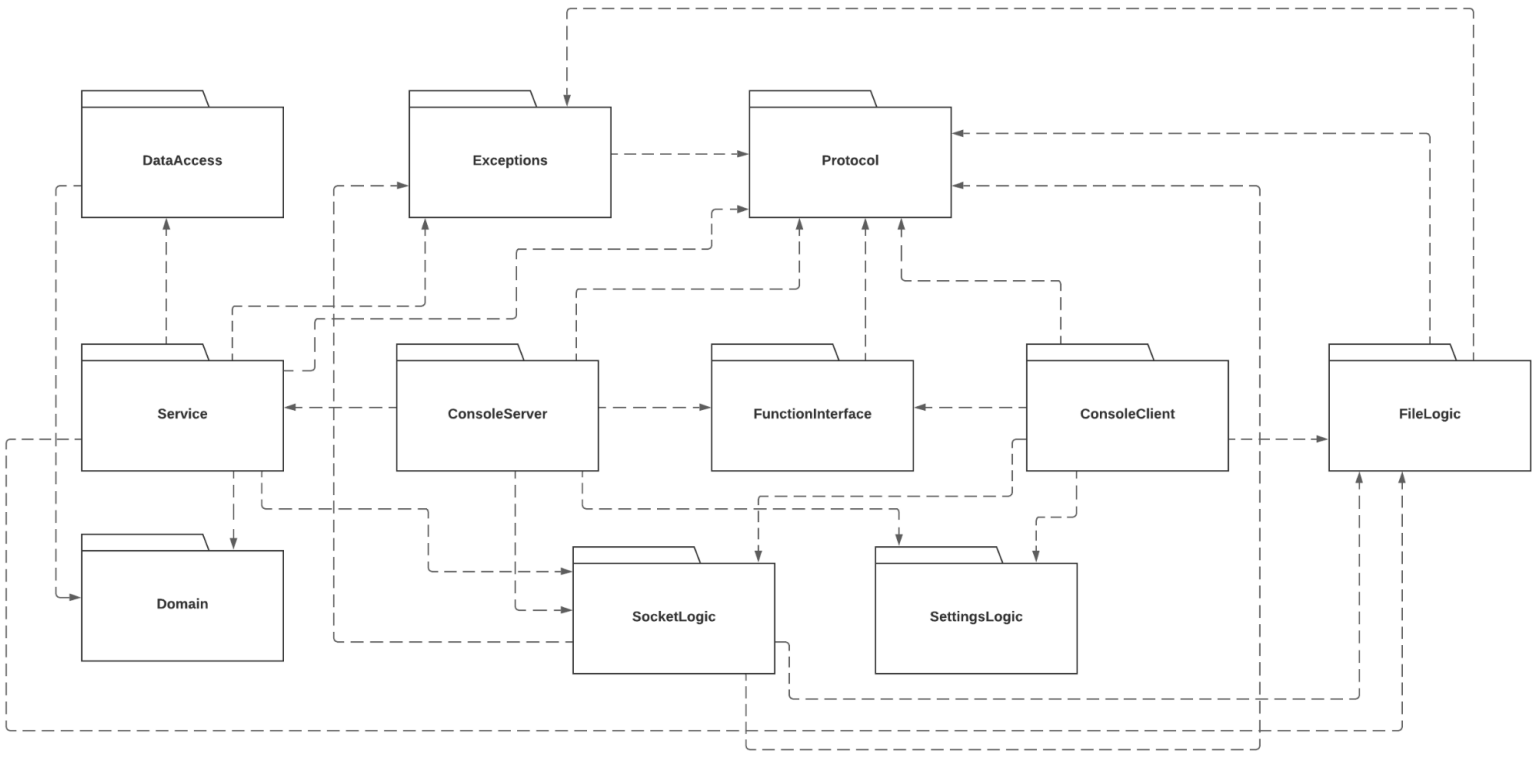


Diagrama de Paquetes

A continuación, se muestra la distribución general de los módulos que componen al sistema. Representa la vista general de la estructura del sistema y de toda la lógica para la interacción cliente-servidor. Cada uno de ellos representa un archivo .dll que estará en el directorio del sistema.



Breve descripción de responsabilidades

Nuestra solución está compuesta por varios proyectos que contienen un conjunto de clases organizadas según las responsabilidades que conlleven. Se describen todos los paquetes según su responsabilidad y las funcionalidades que presentan.

Domain: Responsable de contener las entidades de dominio que el sistema necesita como base.

DataAccess: Contiene las clases responsables de almacenar los datos del sistema. En vistas de una solución más simple, se optó por crear un repositorio por cada entidad de dominio y no utilizar un único repositorio genérico. Se destaca la utilización del patrón "Singleton" aplicado a todos los repositorios existentes, el cual se explica más adelante en el documento.

Exceptions: Contiene clases que representan excepciones personalizadas. En caso de encontrar algún error en la validación de los modelos de entrada, se lanza su correspondiente excepción personalizada que contiene un mensaje detallando lo ocurrido, con el fin de mejorar el feedback hacia el cliente a la hora de lanzar una excepción.

Service: Contiene implementaciones de lógica de negocio para los recursos expuestos. Se utilizan para procesar las consultas que le son delegadas. Dentro, cada funcionalidad tiene sus propias validaciones, así como métodos encargados de colaborar con otras entidades, para brindar el servicio correspondiente.

FunctionInterface: Contiene las interfaces que definen el comportamiento que las funcionalidades del cliente y servidor deben implementar. Se generó principalmente para no violar el principio de Open/Close, en caso de que se necesite implementar nuevas funcionalidades en un futuro.

Protocol: Aquí es donde se define el protocolo necesario para que las aplicaciones cliente y servidor intercambien información mutuamente. Contiene el formato para los "headers", las constantes para las funciones, es decir los comandos para cada funcionalidad del sistema, las constantes para el "header" y los status code creados. La clase "DataPacket" es una fabricación pura que representa el contenido de un paquete en el intercambio de información, a diferencia de la clase "ResponseData" que se desarrolló como contenedor temporal para almacenar las respuestas del servidor.

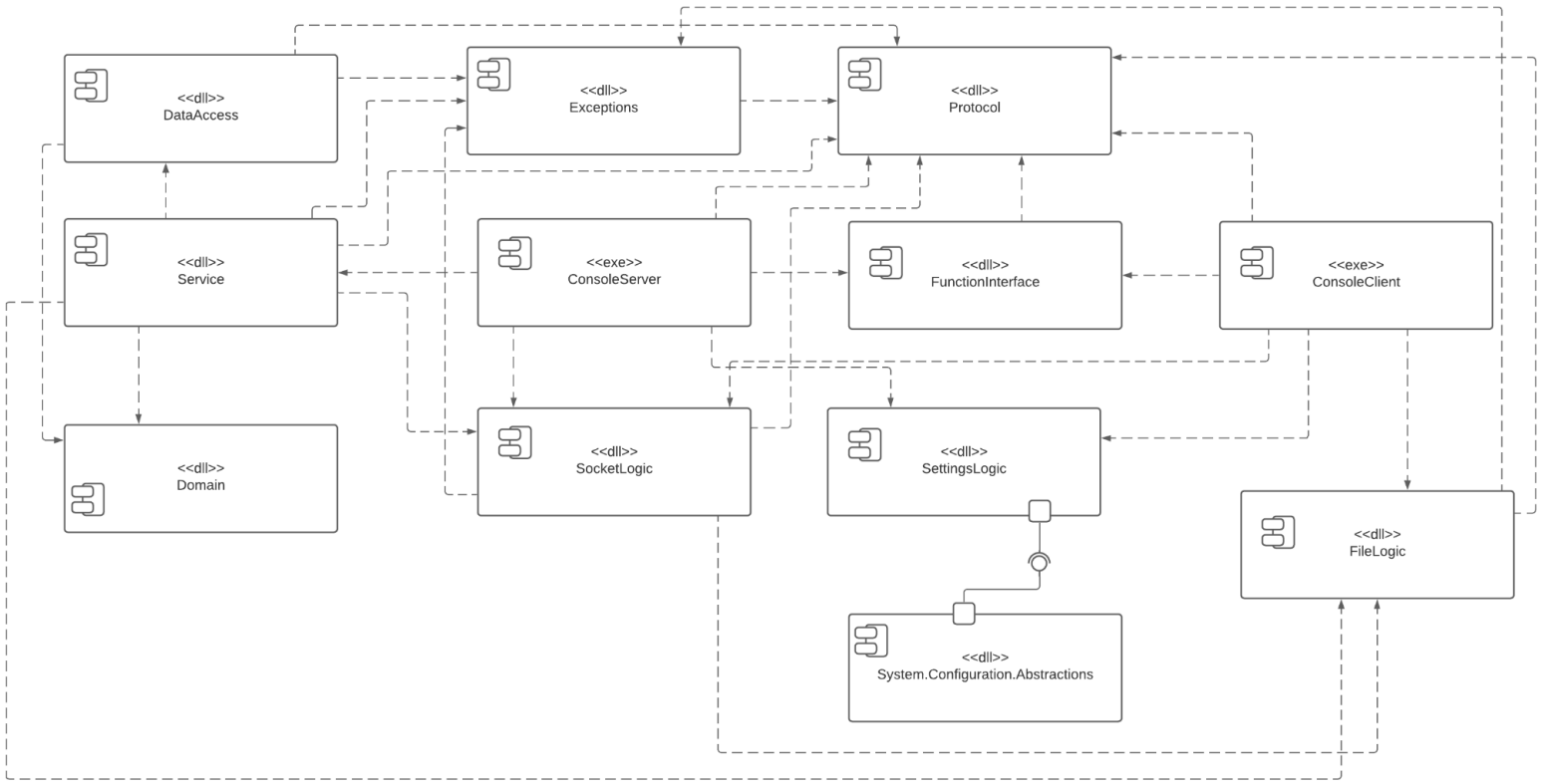
SettingsLogic: Paquete que contiene la lógica dirigida a leer el archivo "appsettings". Cuenta con la interfaz "ISettingsManager" que se encarga de definir los métodos implementados por la clase SettingsManager.

SocketLogic: Paquete que define los métodos de envío y recepción de bytes entre diferentes sockets. Es el nivel más bajo al cual llegaremos en nuestro sistema.

FileLogic: Paquete que define los métodos de lectura y escritura de archivos. A su vez, contiene un manager para obtener información y realizar validaciones de archivos.

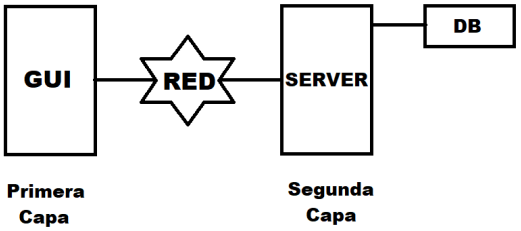
ConsoleClient: Contiene lo relacionado con la aplicación cliente. Tiene el display para imprimir en consola, el administrador de cliente ("ClientHandler") y por último el program con el método main.
ConsoleServer: Contiene lo relacionado con la aplicación servidor. Tiene el display para imprimir en consola, el administrador del servidor("ServerHandler") y por último el program con el método main.

Diagrama de Implementación (Componentes)



Justificación del Diseño

A modo de introducción, comenzando por la arquitectura de alto nivel, el proyecto se basa en un modelo de dos capas: GUI en consola y servidor con la lógica y base de datos en memoria.



Nuevamente cabe destacar que, en esta oportunidad, no se cuenta con persistencia de datos mediante el uso de una base de datos, sino que se recurre al almacenamiento en memoria para los datos ingresados al sistema.

Al momento de desarrollo, la capa de Server se encuentra igualmente dividida en subcapas, siendo estas: Functions, Service y DataAccess. Este enfoque, que consiste en segmentar la capa según sus responsabilidades, cumple con el principio de responsabilidad única (SRP de SOLID) y ayuda a mantener organizado el código para que se pueda encontrar fácilmente dónde se implementa una función determinada.

La comunicación entre el cliente y el servidor se realiza mediante Sockets. Al momento de inicializar el servidor se crea un thread, utilizando la clase Thread de la librería System.Threading, encargado de ejecutar el método ListenForConnections() que emplea la clase TCPListener dado que facilita su uso para comenzar a escuchar conexiones con los sockets de clientes TCPClient. Las clases TCPListener y TCPClient provienen de la librería System.Net.Sockets y son “Wrapper” de TCP que internamente implementan la clase Socket.

Ver Diagrama de Clase: ConsoleClient
Ver Diagrama de Clase: ConsoleServer

El protocolo generado para nuestro sistema es utilizado por ambas partes en el intercambio de información. Se envían headers que almacenan la información correspondiente al tipo de envío, indicando si se trata de una petición o respuesta, seguido por el comando específico de la funcionalidad deseada y el largo de la información que se enviará luego del header.

Ver Diagrama de Clase: Protocol

El formato general de la trama se representa en la clase DataPacket y contiene:

Nombre Del Campo	HEADER	CMD	LARGO	DATOS	STATUS CODE
Valores	RES/REQ	0-99	Entero	Variable	200/201/400/404/500
Largo	3	2	4	Variable	3

El receptor de la información crea un buffer donde reserva el espacio necesario en bytes para almacenar la información y se espera por su llegada en la correspondiente segmentación de paquetes.
Se utilizó la técnica Data Transfer Object (DTO) que consiste en limitar la información que se recibe y devuelve utilizando modelos en formato de cadena de caracteres, separando el contenido con el carácter "#", o en ocasiones se utiliza también "&", que tienen un sub-set de determinada entidad del dominio. Estos modelos son usados principalmente para omitir ciertas properties al momento de intercambiar información con la finalidad de reducir tamaño de las entidades y esconder información, simplificando el problema para los clientes.

Una de las características esenciales que identifica a la arquitectura cliente-servidor es que permite la conexión de varios clientes simultáneamente con el servidor. Para lograr esta concurrencia se utilizan Threads, donde cada cliente conectado tiene su hilo para realizar operaciones, lo que genera un problema de lectura sucia cuando múltiples clientes intentan acceder al mismo recurso (listas de memoria compartida) mientras se actualiza su contenido. Para ello, se soluciona implementando un control de acceso a los recursos (listas de memoria compartida) utilizando locks que los bloqueen cada vez que un cliente quiera realizar operaciones que modifiquen su contenido.

A su vez, se implementó el patrón creacional Singleton para los repositorios en DataAccess y las lógicas en Service, como mecanismo para asegurar que estas clases tienen una única instancia disponible, controlando el acceso al recurso compartido, y se les proporcione un único punto de acceso global para dicha instancia.

Ver Diagrama de Clase: DataAccess

Se implementó el patrón creacional Factory Method para desarrollar las funciones disponibles en el proyecto. Para lograrlo, se proporcionan las interfaces IClientFunction e IServerFunction, contenidas en el paquete FunctionInterface, que exponen el método común Execute() para ser implementado de forma diferente por función. El método fábrica está contenido en las clases FunctionDictionary para cliente y servidor, el cual retorna un diccionario con clave de ID de Función (especificadas como constantes en la clase FunctionConstants) y valor de Instancia de Clase que implementa la Interface para una función particular.

El código que utiliza el método fábrica no encuentra diferencias entre las funciones devueltas y las trata como la Interface IClientFunction o IServerFunction. El cliente sabe que todas las funciones deben tener el método Execute() implementado, pero no necesita saber cómo funciona exactamente.

Al poder incorporar nuevos tipos de funciones en el programa sin descomponer el código existente se cumple el principio de abierto/cerrado.

Ver Diagrama de Clase: FunctionInterface

Se conoce que Factory Method puede funcionar conjuntamente con el patrón de comportamiento Template Method. Optamos por utilizar Template Method que defina un esqueleto de un algoritmo en el método Execute() de la superclase FunctionTemplate, pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura. Los pasos pueden ser abstractos, o ser virtual contando con una implementación por defecto.

Para el cliente, los pasos definidos en el método Execute() son:
BuildRequest() → SendRequest() → ReceiveResponse() → ProcessResponse()

Para el servidor, los pasos definidos en el método Execute() son:
ReceiveRequest() → ProcessRequest() → BuildResponse() → SendResponse()

Los pasos relacionados con Build y Process son declarados como abstractos, forzando las subclases a proporcionar sus propias implementaciones para estos métodos. Ahora, los pasos relacionados con Send y Receive están implementados dentro de la superclase para deshacerse de código duplicado, puesto que no tiene sentido proporcionar implementaciones en la totalidad de subclases, y son declarados como virtual, para poder sobreescribirse en funciones particulares como las encargadas de transferir archivos.

Por ejemplo, para añadir un nuevo tipo de función a la aplicación, sólo se debe crear una nueva subclase de FunctionTemplate, implementar los métodos Build() y Process() y finalmente agregarla al diccionario de FunctionDictionary con clave de ID de Función (especificadas como constantes en la clase FunctionConstants) y valor de Instancia de Clase.

Ver Diagrama de Clase: ConsoleClient
Ver Diagrama de Clase: ConsoleServer

Llegados a este punto, queremos aclarar que se efectuó un supuesto para el requerimiento funcional: CRF4. Búsqueda de juegos. Se deberá poder buscar un juego por categoría, título y rating.
Optamos por implementar el requisito en tres funcionalidades independientes de búsqueda, por lo que el usuario no podrá filtrar con múltiples criterios. Creemos fuertemente que al ser de esta manera, se genera un mayor entendimiento acerca de la búsqueda de juegos que el sistema ofrece.

Para verificar la correctitud de los modelos de entrada, se implementaron clases validadoras en el paquete Service. Entre las validaciones implementadas, la principal y comúnmente utilizada trata de que los datos de entrada no estén vacíos.

Ver Diagrama de Clase: Service

En caso de encontrar un defecto en los modelos de entrada, se implementaron un conjunto de excepciones personalizadas, que son lanzadas según el contexto particular, para tener mayor conocimiento de lo ocurrido. Los casos contemplados hasta la fecha son:

- Not Found: No se encuentra una entidad solicitada.
- Invalid Input: La información entrante contiene datos erróneos.
- Already Exists: La entidad entrante ya existe en el sistema.
- No Readable File: Un archivo no se puede leer.

Estas excepciones personalizadas heredan de una excepción generalizada igualmente personalizada que se encarga de asociar el mensaje de las hijas con un código de estado. Esta excepción generalizada (AppException) se utiliza para centralizar todas las excepciones personalizadas y es atrapada por el filtro de excepciones, generando un bloque try-catch más reducido.

Ver Diagrama de Clase: Exceptions

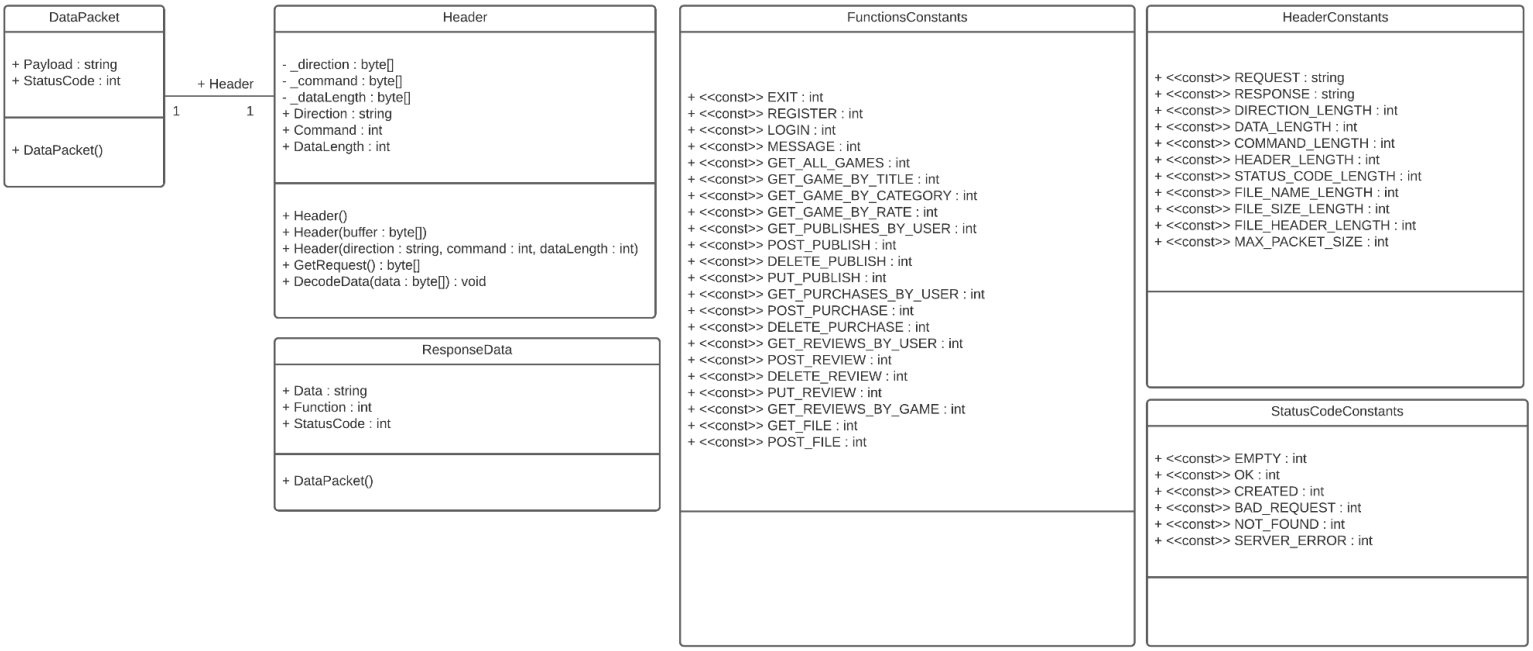
Se utiliza un conjunto reducido de códigos de estado, comúnmente conocidos por los desarrolladores, permitiendo simplificar el significado de retorno.

- 200 - Ok:
  Se utiliza como respuesta a solicitudes GET correctas.
- 201 - Created:
  Se utiliza como respuesta a solicitudes POST correctas, indicando que la petición ha sido completada y ha resultado en la creación de un nuevo recurso.
- 400 - Bad Request:
  Se utiliza como respuesta a solicitudes erróneas (error de cliente), cuando la solicitud contiene sintaxis errónea, indicando que no procederá la solicitud en el servidor.
- 404 - Not Found:
  Se utiliza como respuesta a solicitudes erróneas (error de cliente), cuando no se encuentra el recurso solicitado.
- 500 - Internal Server Error:
  Se utiliza como respuesta a solicitudes erróneas (error de servidor), cuando ocurre un problema interno.

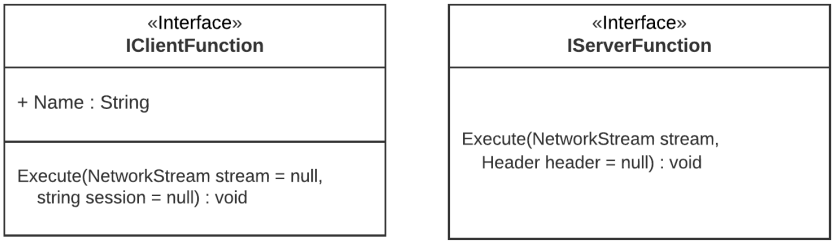
Cabe destacar que estos códigos se retornan acompañados de un mensaje significativo de lo ocurrido.

Anexo: Diagramas de Clases

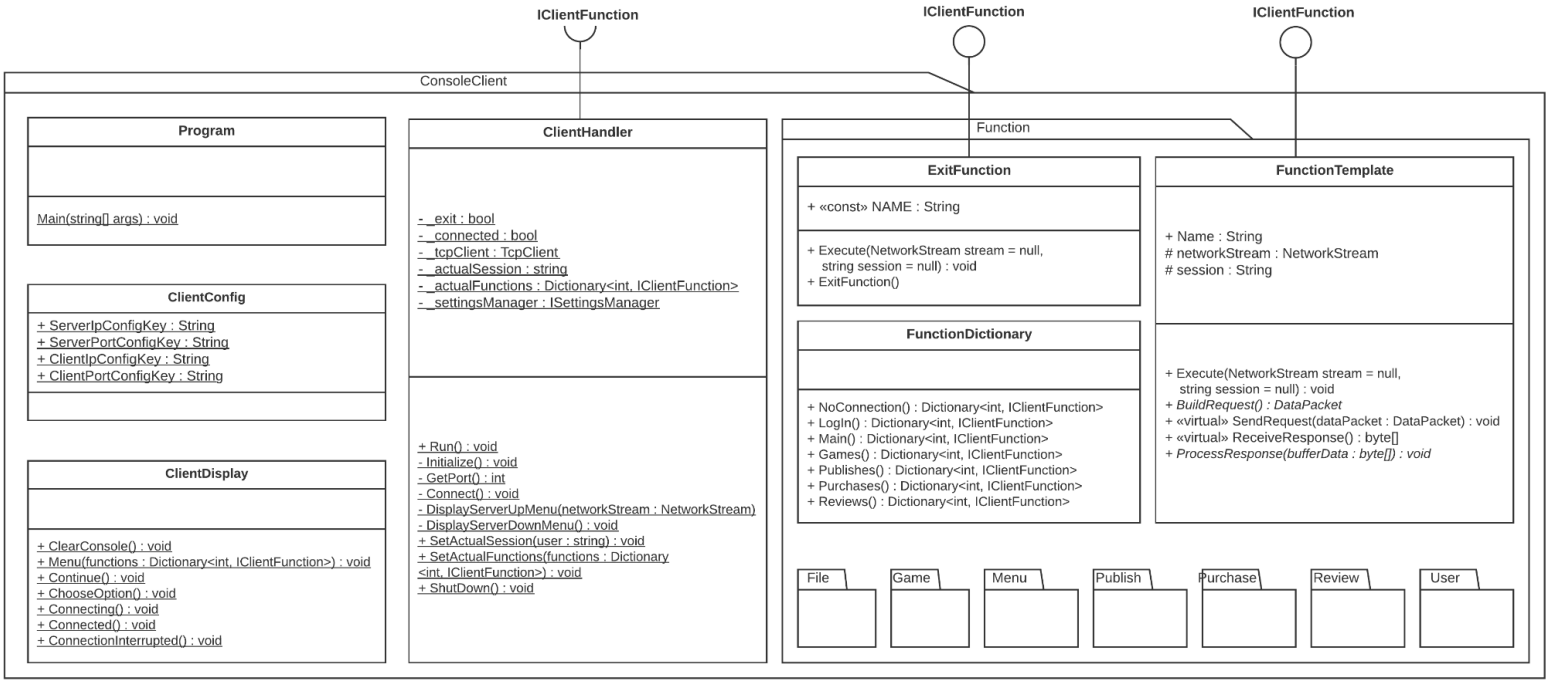
Protocol



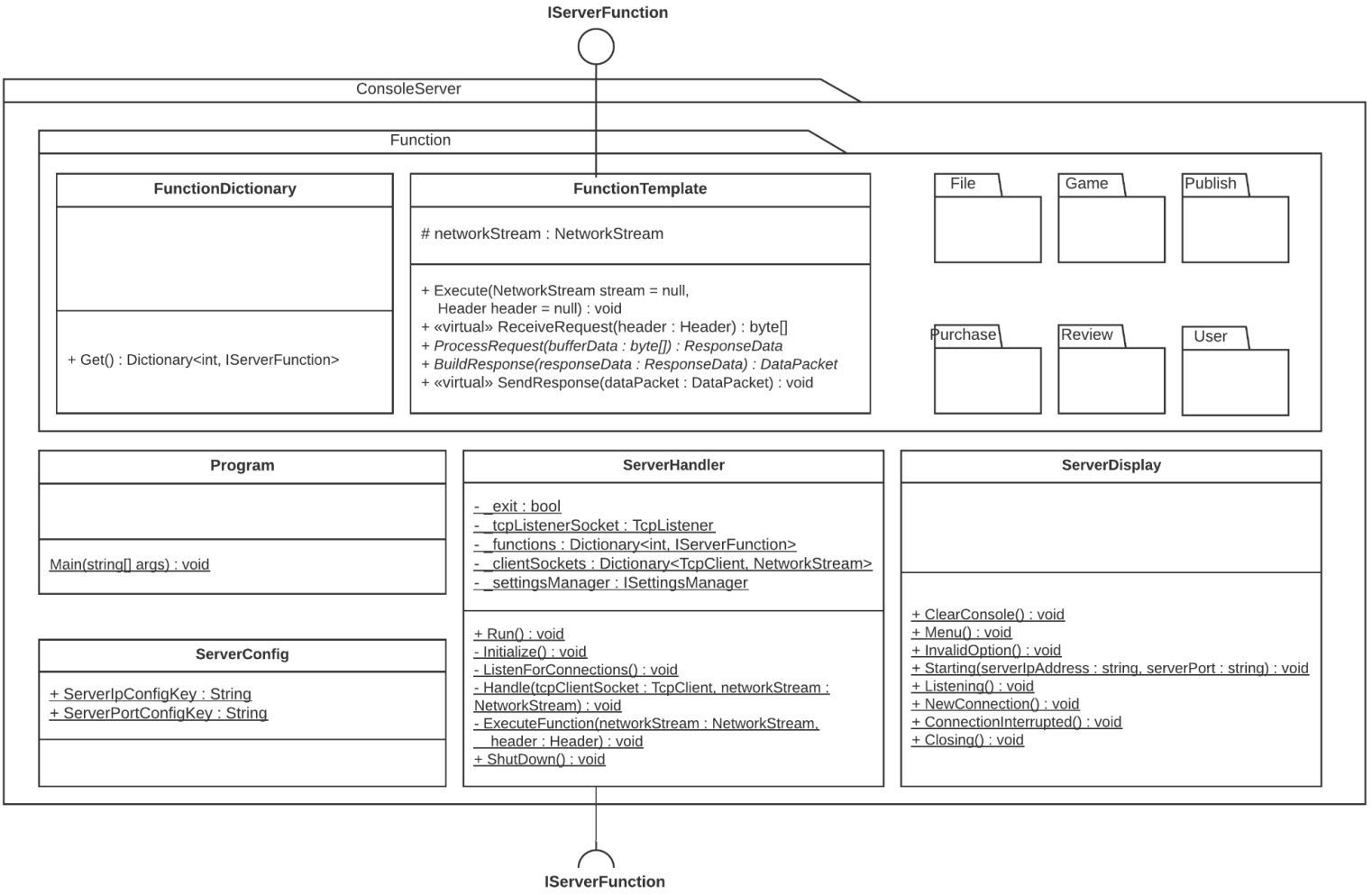
FunctionInterface



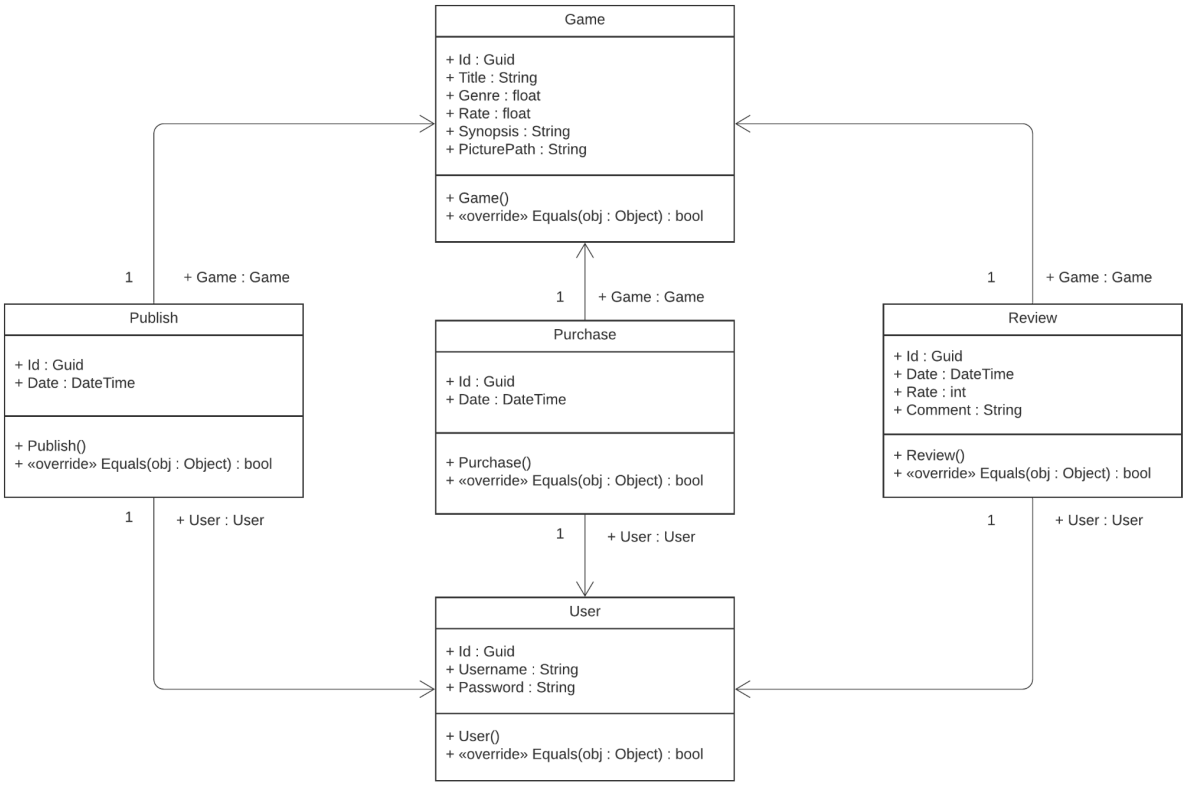
ConsoleClient



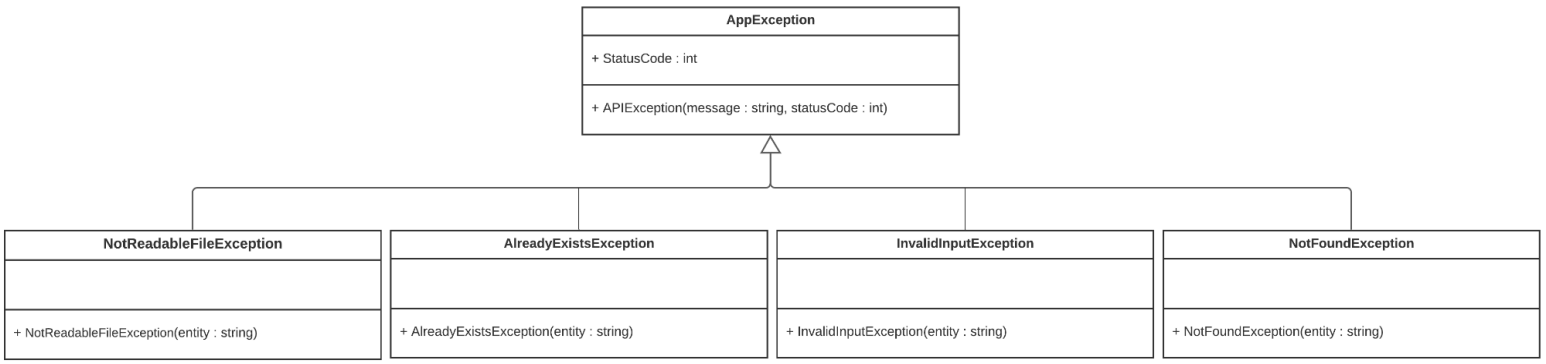
ConsoleServer



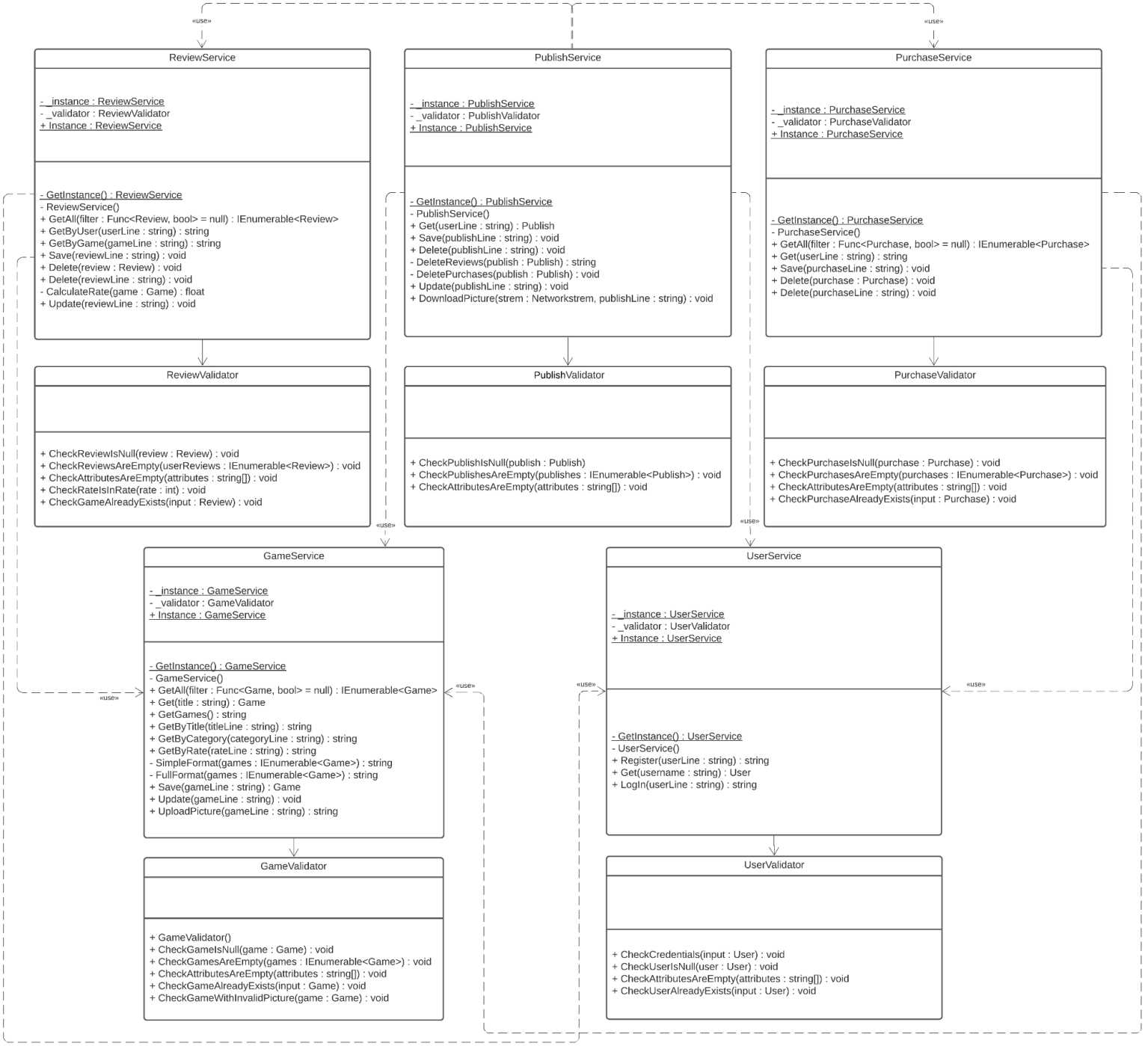
Domain



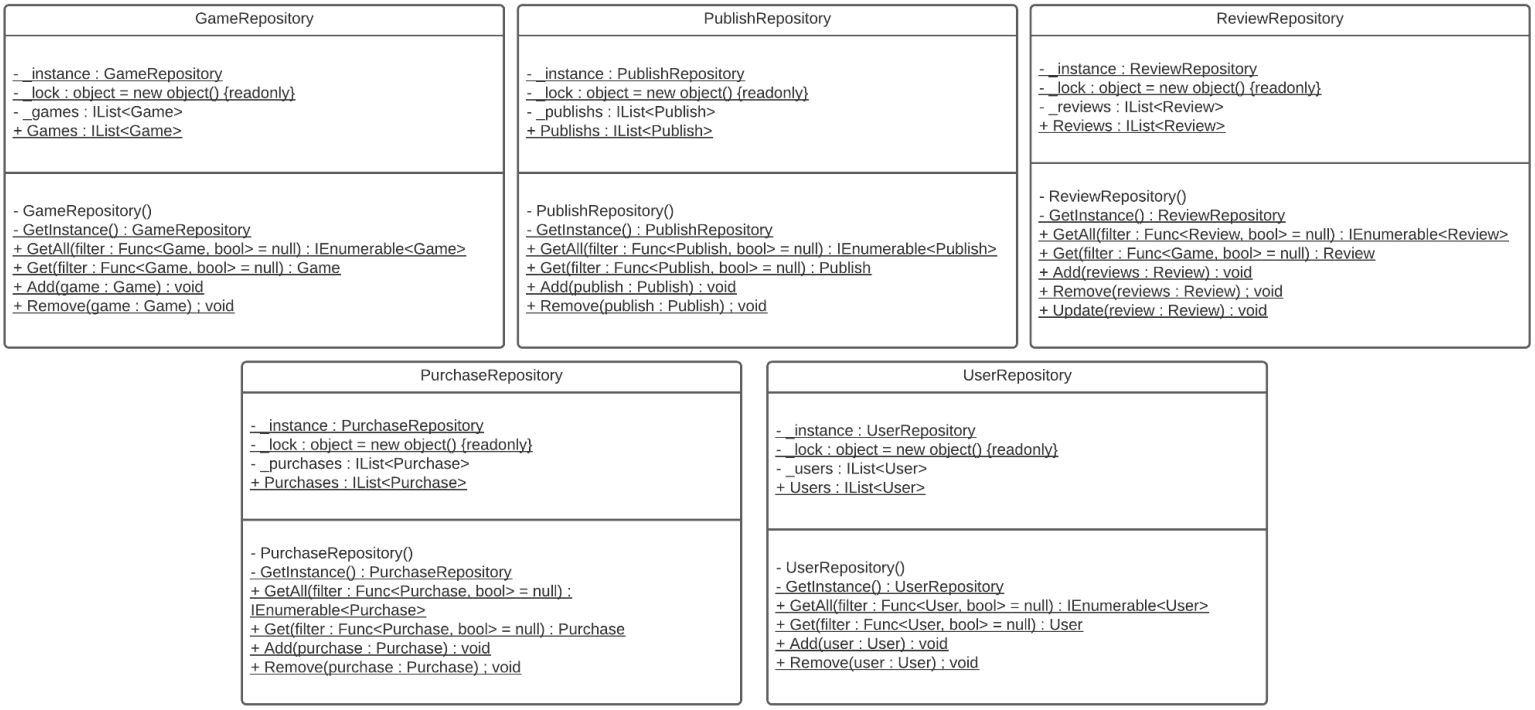
Exceptions



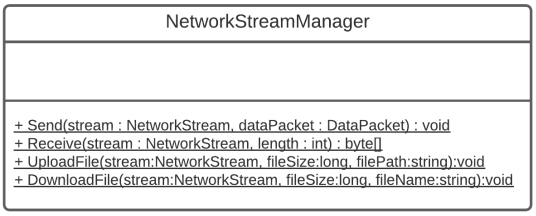
Service


«use»
«use»
ReviewService
- _instance : ReviewService
- _validator : ReviewValidator
+ Instance : ReviewService
- GetInstance() : ReviewService
- ReviewService()
+ GetAll(filter : Func<Review, bool> = null) : IEnumerable<Review>
+ GetByUser(userLine : string) : string
+ GetByGame(gameLine : string) : string
+ Save(reviewLine : string) : void
+ Delete(review : Review) : void
+ Delete(reviewLine : string) : void
- CalculateRate(game : Game) : float
+ Update(reviewLine : string) : void
PublishService
- _instance : PublishService
- _validator : PublishValidator
+ Instance : PublishService
- GetInstance() : PublishService
- PublishService()
+ Get(userLine : string) : Publish
+ Save(publishLine : string) : void
+ Delete(publishLine : string) : void
- DeleteReviews(publish : Publish) : string
- DeletePurchases(publish : Publish) : void
+ Update(publishLine : string) : void
+ DownloadPicture(strem : Networkstrem, publishLine : string) : void
PurchaseService
- _instance : PurchaseService
- _validator : PurchaseValidator
+ Instance : PurchaseService
- GetInstance() : PurchaseService
- PurchaseService()
+ GetAll(filter : Func<Purchase, bool> = null) : IEnumerable<Purchase>
+ Get(userLine : string) : string
+ Save(purchaseLine : string) : void
+ Delete(purchase : Purchase) : void
+ Delete(purchaseLine : string) : void
ReviewValidator
+ CheckReviewIsNull(review : Review) : void
+ CheckReviewsAreEmpty(userReviews : IEnumerable<Review>) : void
+ CheckAttributesAreEmpty(attributes : string[]) : void
+ CheckRateIsInRate(rate : int) : void
+ CheckGameAlreadyExists(input : Review) : void
PublishValidator
+ CheckPublishIsNull(publish : Publish)
+ CheckPublishesAreEmpty(publishes : IEnumerable<Publish>) : void
+ CheckAttributesAreEmpty(attributes : string[]) : void
PurchaseValidator
+ CheckPurchaseIsNull(purchase : Purchase) : void
+ CheckPurchasesAreEmpty(purchases : IEnumerable<Purchase>) : void
+ CheckAttributesAreEmpty(attributes : string[]) : void
+ CheckPurchaseAlreadyExists(input : Purchase) : void
«use»
«use»
«use»
«use»
«use»
«use»
GameService
- _instance : GameService
- _validator : GameValidator
+ Instance : GameService
- GetInstance() : GameService
- GameService()
+ GetAll(filter : Func<Game, bool> = null) : IEnumerable<Game>
+ Get(title : string) : Game
+ GetGames() : string
+ GetByTitle(titleLine : string) : string
+ GetByCategory(categoryLine : string) : string
+ GetByRate(rateLine : string) : string
- SimpleFormat(games : IEnumerable<Game>) : string
- FullFormat(games : IEnumerable<Game>) : string
+ Save(gameLine : string) : Game
+ Update(gameLine : string) : void
+ UploadPicture(gameLine : string) : string
UserService
- _instance : UserService
- _validator : UserValidator
+ Instance : UserService
- GetInstance() : UserService
- UserService()
+ Register(userLine : string) : string
+ Get(username : string) : User
+ LogIn(userLine : string) : string
GameValidator
+ GameValidator()
+ CheckGameIsNull(game : Game) : void
+ CheckGamesAreEmpty(games : IEnumerable<Game>) : void
+ CheckAttributesAreEmpty(attributes : string[]) : void
+ CheckGameAlreadyExists(input : Game) : void
+ CheckGameWithInvalidPicture(game : Game) : void
UserValidator
+ CheckCredentials(input : User) : void
+ CheckUserIsNull(user : User) : void
+ CheckAttributesAreEmpty(attributes : string[]) : void
+ CheckUserAlreadyExists(input : User) : void

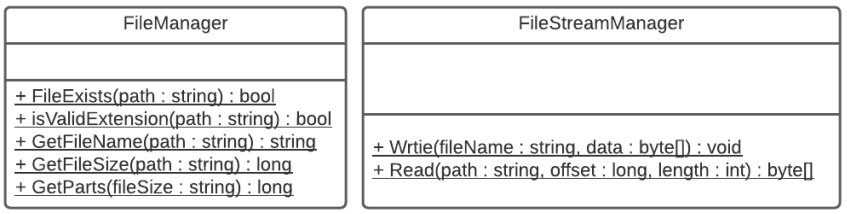
DataAccess


GameRepository
- _instance : GameRepository
- _lock : object = new object() {readonly}
- _games : IList<Game>
+ Games : IList<Game>
- GameRepository()
- GetInstance() : GameRepository
+ GetAll(filter : Func<Game, bool> = null) : IEnumerable<Game>
+ Get(filter : Func<Game, bool> = null) : Game
+ Add(game : Game) : void
+ Remove(game : Game) ; void
PublishRepository
- _instance : PublishRepository
- _lock : object = new object() {readonly}
- _publishs : IList<Publish>
+ Publishs : IList<Publish>
- PublishRepository()
- GetInstance() : PublishRepository
+ GetAll(filter : Func<Publish, bool> = null) : IEnumerable<Publish>
+ Get(filter : Func<Publish, bool> = null) : Publish
+ Add(publish : Publish) : void
+ Remove(publish : Publish) ; void
ReviewRepository
- _instance : ReviewRepository
- _lock : object = new object() {readonly}
- _reviews : IList<Review>
+ Reviews : IList<Review>
- ReviewRepository()
- GetInstance() : ReviewRepository
+ GetAll(filter : Func<Review, bool> = null) : IEnumerable<Review>
+ Get(filter : Func<Game, bool> = null) : Review
+ Add(reviews : Review) : void
+ Remove(reviews : Review) : void
+ Update(review : Review) : void
PurchaseRepository
- _instance : PurchaseRepository
- _lock : object = new object() {readonly}
- _purchases : IList<Purchase>
+ Purchases : IList<Purchase>
- PurchaseRepository()
- GetInstance() : PurchaseRepository
+ GetAll(filter : Func<Purchase, bool> = null) :
IEnumerable<Purchase>
+ Get(filter : Func<Purchase, bool> = null) : Purchase
+ Add(purchase : Purchase) : void
+ Remove(purchase : Purchase) ; void
UserRepository
- _instance : UserRepository
- _lock : object = new object() {readonly}
- _users : IList<User>
+ Users : IList<User>
- UserRepository()
- GetInstance() : UserRepository
+ GetAll(filter : Func<User, bool> = null) : IEnumerable<User>
+ Get(filter : Func<User, bool> = null) : User
+ Add(user : User) : void
+ Remove(user : User) ; void

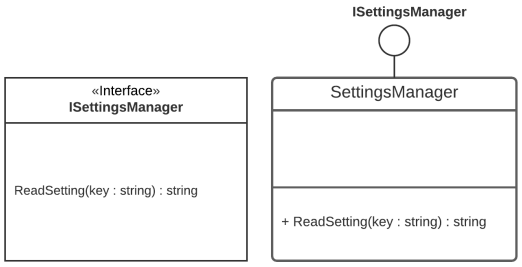
SocketLogic



FileLogic



SettingsLogic



Anexo: Listado de Funciones

ID	Funcion	Resource	Description	Responses
0	EXIT	User	Cierra la conexión	200
1	REGISTER	User	Crea un usuario	201, 400, 500
2	LOGIN	User	Logeo de usuario	200, 404, 500
5	GET_ALL_GAMES	Game	Retorna todos los juegos	200, 404, 500
6	GET_GAME_BY_TITLE	Game	Retorna el juego correspondiente al nombre ingresado	200, 404, 500
7	GET_GAME_BY_CATEGORY	Game	Retorna los juegos con igual categoría a la ingresada	200, 404, 500
8	GET_GAME_BY_RATE	Game	Retorna los juegos con igual calificación a la ingresada	200, 404, 500
10	GET_PUBLISHES_BY_USER	Publish	Retorna todas las publicaciones del usuario ingresado	200, 404, 500
11	POST_PUBLISH	Publish	Crea una publicación	201, 400, 500
12	DELETE_PUBLISH	Publish	Elimina la publicación correspondiente al nombre ingresado	200, 404, 500
13	PUT_PUBLISH	Publish	Modifica la publicación correspondiente al nombre del juego ingresado	200, 400, 404, 500
20	GET_PURCHASES_BY_USER	Purchase	Retorna todas las compras del usuario ingresado	200, 404, 500
21	POST_PURCHASE	Purchase	Crea una compra	201, 400, 500
22	DELETE_PURCHASE	Purchase	Elimina la compra correspondiente al nombre ingresado	200, 404, 500
30	GET_REVIEWS_BY_USER	Review	Retorna todas las reseñas del usuario ingresado	200, 404, 500
31	POST_REVIEW	Review	Crea una reseña	201, 400, 500
32	DELETE_REVIEW	Review	Elimina la reseña correspondiente al nombre ingresado	200, 404, 500
33	PUT_REVIEW	Review	Modifica la reseña correspondiente al nombre del juego ingresado	200, 400, 404, 500
36	GET_REVIEWS_BY_GAME	Review	Retorna todas las reseñas del juego ingresado	200, 404, 500
40	GET_FILE	Game	Retorna una imagen de un juego	200, 404, 500
41	POST_FILE	Publish	Carga una imagen en una publicación	201, 400, 500

Referencias bibliográficas

Apuntes e información de las prestaciones correspondientes a Programación de Redes, en Universidad ORT Uruguay.

Apuntes e información de las prestaciones correspondientes a Diseño de Aplicaciones 1 y 2, en Universidad ORT Uruguay.

Patron Singleton

Patron Factory Method

Patron Template Method