



Índice

ÍNDICE	1
REQUERIMIENTOS DE SOFTWARE	4
SISTEMAS OPERATIVOS.....	4
HERRAMIENTAS Y BIBLIOTECAS.....	4
Desarrollo.....	4
Interfases Gráficas	4
<i>Glade 3.4.5</i>	4
<i>Gtkmm 2.4</i>	4
<i>LibGlademm 2.4</i>	4
Otras Bibliotecas	4
<i>TinyXML</i>	4
<i>Trivial C++ Logger</i>	4
Compilación y Sistemas de Construcción.....	5
¿Cómo Compilar?	5
Pruebas	5
DESCRIPCIÓN GENERAL.....	6
ARQUITECTURA.....	6
MÓDULOS	6
Editor de Escenarios	6
Servidor.....	6
Cliente	6
Biblioteca Común	6
Biblioteca de Juego	6
Biblioteca de Interfaz de Usuario	6
EDITOR DE ESCENARIOS.....	7
DESCRIPCIÓN GENERAL	7
CLASES	7
Editor::Presentation.....	7
<i>StageEditorGui</i>	7
<i>StageEditorHandler</i>	7
DIAGRAMAS UML.....	8
Arquitectura	8
Modelo.....	9
DESCRIPCIÓN DE ARCHIVOS Y PROTOCOLOS	9
Representación en xml de un Escenario	9
SERVIDOR.....	10
DESCRIPCIÓN GENERAL	10
CLASES	10
Server::Game.....	10
<i>GameServer</i>	10
Server::Game::Levels	10
<i>LevelManager</i>	10
<i>Level</i>	11
Server::Game::Sessions.....	11
<i>GameSessionManager</i>	11
<i>GameSession</i>	12
DIAGRAMAS UML.....	13
Servidor.....	13
Comunicación con el Cliente	14
Comandos	15
Manejo de Sesiones y Niveles	16



DESCRIPCIÓN DE ARCHIVOS Y PROTOCOLOS	17
Niveles	17
<i>Estructura de Carpetas.....</i>	17
<i>Convención de Nombres.....</i>	17
Protocolo de Comunicación	17
<i>Envío de un Mensaje.....</i>	17
<i>Comando Login</i>	18
<i>Comando de Unión a Sesión.....</i>	18
<i>Comando de Devolución de Sesiones Disponibles</i>	18
<i>Comando de Envío de Niveles</i>	18
<i>Comando de Comienzo de un Nivel.....</i>	19
<i>Comando de envío de Feedback.....</i>	19
CLIENTE	20
DESCRIPCIÓN GENERAL	20
CLASES	20
Client::Game::Facade.....	20
<i>CommandFacade.....</i>	20
Client::Game::Model.....	21
<i>GameClient</i>	21
DIAGRAMAS UML.....	22
Modelo.....	22
Comandos	23
Controladores de Vista.....	23
DESCRIPCIÓN DE ARCHIVOS Y PROTOCOLOS	24
Niveles	24
<i>Estructura de Carpetas.....</i>	24
<i>Convención de Nombres.....</i>	24
Protocolo de Comunicación	24
<i>Comando para Login en el Servidor.....</i>	24
<i>Comando para Envíos de Mensajes de Chat.....</i>	24
<i>Comando de Solicitud de Sesiones Disponibles.....</i>	24
<i>Comando de Creación de Sesiones Solitarias.....</i>	24
<i>Comando de Creación de Sesiones Multijugador.....</i>	25
<i>Comando de Unión a Sesión.....</i>	25
<i>Comando de Solicitud de un Nivel.....</i>	25
<i>Comando de Notificación de Finalización de un Nivel.....</i>	25
<i>Comando de envío de Feedback.....</i>	25
BIBLIOTECA COMÚN	26
DESCRIPCIÓN GENERAL	26
CLASES	26
Logger.....	26
Common::Protocol.....	26
<i>ProtocolSender.....</i>	26
<i>ProtocolReceiver.....</i>	26
Common::Sockets	27
<i>Data.....</i>	27
<i>TcpSocket</i>	27
Common::Threads	29
<i>Thread.....</i>	29
Common::Utils	29
<i>Math.....</i>	29
<i>StringUtils.....</i>	30
<i>FileUtils.....</i>	30
DIAGRAMAS UML.....	31
DESCRIPCIÓN DE ARCHIVOS Y PROTOCOLOS	31
Protocolo General de Intercambio	31
<i>¿Por qué decimos que este protocolo es general?</i>	32
<i>¿Por qué se envía la longitud de los datos como un vector de caracteres?</i>	32



BIBLIOTECA DE JUEGO	33
DESCRIPCIÓN GENERAL	33
CLASES	33
Game::Core	33
<i>Particle</i>	33
<i>Spring</i>	35
<i>PhysicsEngine</i>	35
Game::Core::Affectors	36
<i>ForceAffector</i>	36
Game::Design	36
<i>ElementContainer</i>	36
<i>ElementFactory</i>	36
Game::Design::Elements	36
<i>Connection</i>	36
<i>Element</i>	37
Game::Design::Connect	39
<i>Connect</i>	39
Game::Design::Move	39
<i>MovePoint</i>	39
Game::Logic	40
<i>DesignToolbox</i>	40
<i>GameEngine</i>	40
Game::Serialization	41
Game::serialization::service	41
<i>StageService</i>	41
<i>StageServiceImpl</i>	41
<i>ServiceLocator</i>	41
Game::Serialization::Persistence	41
<i>StageXmlRepository</i>	41
<i>StageXmlRepositoryTinyXml</i>	41
<i>RepositoryFactory</i>	41
DIAGRAMAS UML	42
Core	42
Logic	43
Design	43
BIBLIOTECA DE INTERFAZ DE USUARIO	45
DESCRIPCIÓN GENERAL	45
CLASES	45
UI::Controllers	45
<i>ConfigurableRenderAreaController</i>	45
<i>RenderAreaInputHandler</i>	45
UI::Views	46
<i>Renderer</i>	46
<i>GameRenderArea</i>	46
DIAGRAMAS UML	48
PROGRAMAS INTERMEDIOS Y DE PRUEBA	49
Pruebas de Conectividad y Protocolo	49
Pruebas Gráficas	49
Pruebas Algorítmicas	49
CÓDIGO FUENTE	50



Requerimientos de Software

Sistemas Operativos

Originalmente, este software fue diseñado y pensado para correr bajo Linux. De todas maneras, se ha puesto empeño en lograr que sea multiplataforma. Si bien no se ha llegado a probar en sistemas operativos no pertenecientes a la familia de Linux, creemos que es factible que la aplicación se pueda compilar y ejecutar sin inconvenientes¹.

Herramientas y Bibliotecas

Hemos utilizado diversas herramientas y bibliotecas con el objetivo de resolver distintas problemáticas.

DESARROLLO

Para el desarrollo decidimos utilizar *Eclipse* como entorno integrado de desarrollo.

INTERFASES GRÁFICAS

Para el diseño de interfases gráficas se utilizaron un conjunto de bibliotecas multiplataforma que mencionaremos a continuación.

Glade 3.4.5

Es una herramienta de desarrollo visual de interfases gráficas. Su utilización fue realmente muy beneficiosa puesto que nos permitió acelerar los tiempos de diseño sin necesidad de escribir código.

Gtkmm 2.4

Es la versión orientada a objetos de la popular biblioteca GTK+ para la creación de entornos gráficos.

LibGlademm 2.4

Esta biblioteca fue utilizada con el fin de leer los archivos XML provenientes de Glade. Estos archivos contienen el diseño de la interfaz gráfica y esta biblioteca es la que nos permite obtener objetos de Gtkmm a partir de lo definido en dichos archivos.

OTRAS BIBLIOTECAS

TinyXML

Dado que el formato de intercambio entre el cliente y el servidor es XML fue necesario la lectura y escritura de este tipo de archivos. TinyXML es una biblioteca sencilla que nos proveyó las herramientas necesarias para el manejo de archivos XML. Vale aclarar que esta librería fue incluida dentro del código fuente.

Trivial C++ Logger

Otro requerimiento del enunciado es el hecho de permitir optar por una ejecución en modo de *debug*. Para ello utilizamos este *logger* que da la posibilidad de imprimir por pantalla o a un archivo de *log* aquellos eventos críticos.

¹ Deberán efectuarse pequeñas modificaciones en el código para utilizar la librería de Sockets de Windows.



COMPILACIÓN Y SISTEMAS DE CONSTRUCCIÓN

Una de las problemáticas iniciales con las que el grupo debió lidiar fue el hecho de la colaboración a distancia. Es decir, desarrollar el presente trabajo sin necesidad de estar constantemente reunidos. Para ello se debió optar por utilizar un estándar en lo que respecta a la construcción de la aplicación.

Los sistemas de construcción² resultaron ser muy útiles a la hora de generar los ejecutables de la aplicación, logrando que la construcción sea relativamente portable, al menos entre ambientes Unix.

La herramienta que se eligió fue CMake en su versión 2.6.2. Su uso nos simplificó considerablemente los problemas típicos de compilación de una aplicación con cierta envergadura.

¿CÓMO COMPILAR?

Para poder compilar la aplicación se deberá crear una carpeta *build* (por ejemplo) en el mismo nivel donde se encuentra la carpeta *src* (la cual contiene el código fuente). Luego se deberán ejecutar las siguientes líneas.

```
build $> cmake ../  
build $> make
```

Esto generará una carpeta *bin* dentro de la carpeta *build* con los ejecutables y otros archivos necesarios para la ejecución (por ejemplo, los archivos de *glade*).

Para habilitar o deshabilitar el *logger*, se deberá modificar el archivo presente en *src*, llamado CMakeLists.txt. En la sección de *add_definitions* observaremos un *flag* que podrá ser *-DTLOG* o bien *-DTFLOG=file*. En caso de no desear *loguear*, no se debe incluir ninguno de dichos *flags*. En caso de querer *loguear* a salida estándar se deberá utilizar el primer *flag* mencionado, mientras que para *loguear* a un archivo en particular se deberá utilizar el segundo.

PRUEBAS

Inicialmente se conversó acerca de la posibilidad de utilizar algún *framework* de pruebas como por ejemplo *cppUnit*. Si bien las ventajas son innumerables, creímos conveniente por una cuestión de tiempo, el hecho de crear pequeños programas de prueba.

² En la jerga informática son más conocidos como *Build Systems*.



Descripción General

Arquitectura

La arquitectura será, claramente, cliente-servidor. Ahora bien, en lo que a cada módulo respecta se ha optado por utilizar arquitecturas y diseños flexibles que nos permitan efectuar modificaciones, agregados y mejoras de forma sencilla y rápida.

Módulos

Si bien en las siguientes secciones se hablará en detalle de cada módulo en particular, aquí daremos una brevísimas descripción de ellos. El proyecto se divide principalmente en cinco módulos, a saber:

EDITOR DE ESCENARIOS

El editor de escenarios nos permite crear nuevos niveles para que sean cargados en el servidor y, de esta manera, extender y hacer más desafiante al juego.

SERVIDOR

Es el servidor del juego al cual se conectarán los clientes que deseen jugar a *The Astounding Machine*.

CLIENTE

Los clientes son los usuarios del juego. Este módulo es quien recibe los niveles del servidor, permite jugar y conversar con un rival. Todo esto a través de una interfaz gráfica amigable.

BIBLIOTECA COMÚN

Esta biblioteca agrupa todas aquellas clases que son utilizadas por el resto de los módulos.

BIBLIOTECA DE JUEGO

Esta biblioteca contiene aquellas clases que brindan los servicios asociados a la simulación física. Es además donde se encuentra el modelo principal de la aplicación, incluye los elementos constitutivos de un nivel y las clases asociadas a su manejo.

BIBLIOTECA DE INTERFAZ DE USUARIO

Esta biblioteca expone los servicios de manejo genérico de interfaz con el usuario. Contiene las clases asociadas al *rendering* de elementos, así como también los controladores asociados a las acciones del usuario.



Editor de Escenarios

Descripción General

El Editor de Escenarios será utilizado para la creación de los distintos niveles que estarán disponibles en el Servidor. Al momento de la instalación ambos se instalan juntos dado su estrecha relación.

Clases

A continuación se detallarán las clases y métodos principales de este módulo por espacio de nombres (*namespace*).

EDITOR::PRESENTATION

StageEditorGui

Representa a la vista del patrón MVC. Es una interfaz que agrupa a todos los métodos que pueden ser llamados por el controller. No contiene ninguna referencia a ninguna librería gráfica en particular.

StageEditorHandler

Es el Controller del patrón MVC. Es la única clase de presentation que tiene acceso al módulo *serialization* utilizando al *BusinessDelegate*, que es la representación del model del patrón MVC.

```
/**
 * Envía la orden de persistir un escenario.
 */
void saveStage();

/**
 * Persiste un escenario con un nombre dado.
 * @return true Si el escenario se persistió con éxito y false si el
 *         escenario no se persistió.
 */
bool saveStage(const std::string& stageName);

/**
 * Envía la orden de persistir un escenario pidiendo un nombre por
 * pantalla.
 */
void saveAsStage();

/**
 * Recupera un escenario
 */
void loadStage();

/**
 * Inicializa la vista para un nuevo escenario
 */
void newStage();

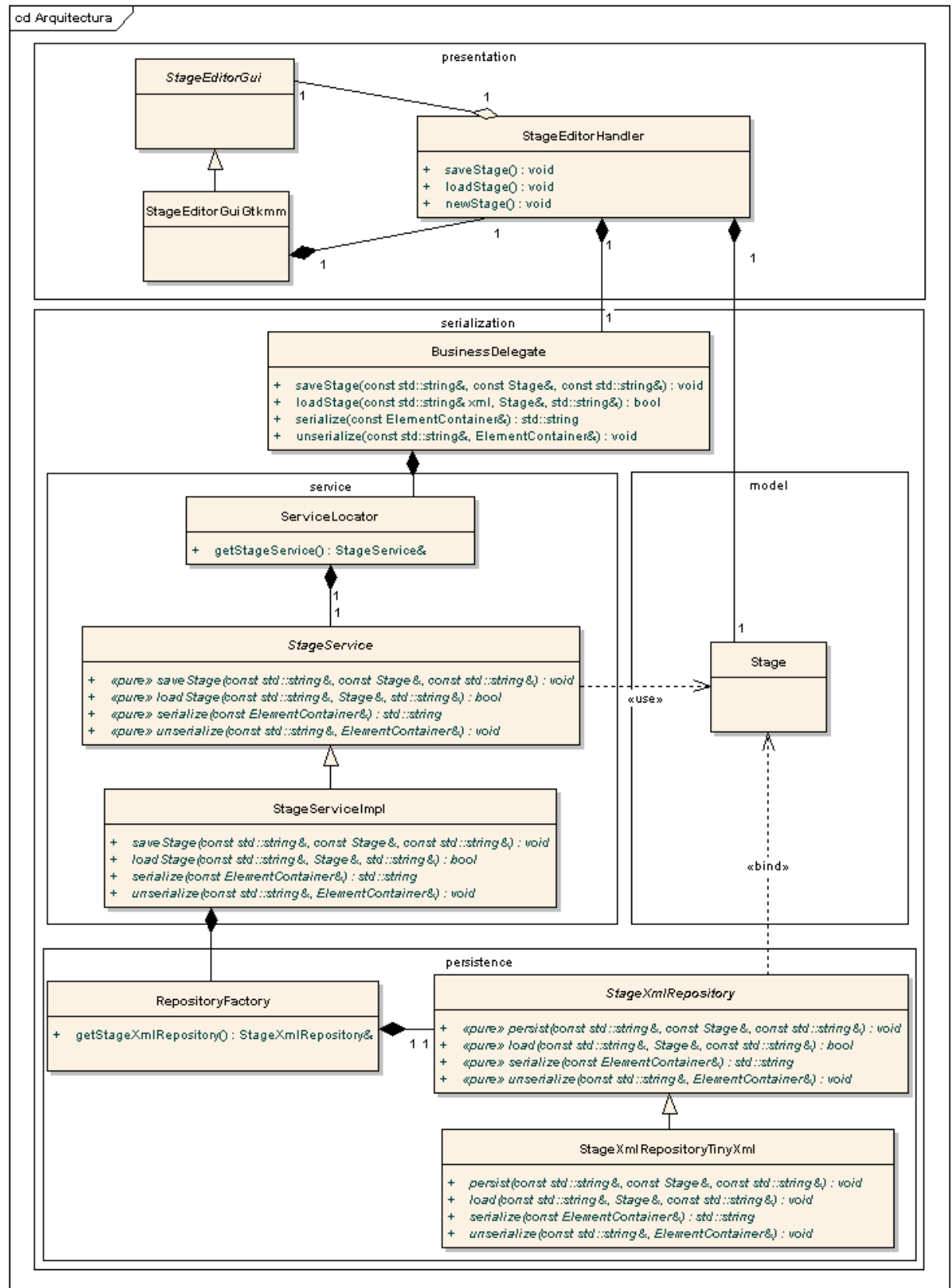
/**
 * @return El contenedor de los elementos del escenario
 */
game::design::ElementContainer& getElementContainer();

/**
 * @return La fábrica para los elementos del escenario
 */
game::serialization::model::ElementFactoryImpl& getElementFactory();
```

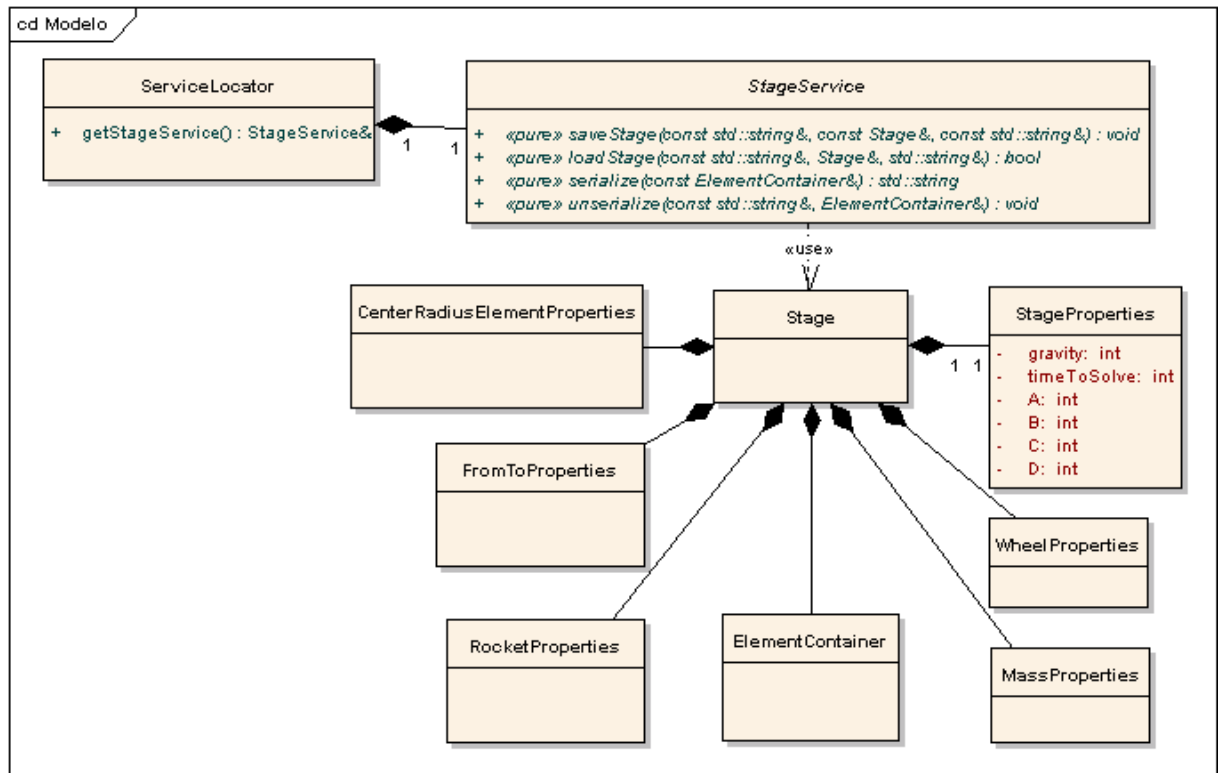


Diagramas UML

ARQUITECTURA



MODELO



Descripción de Archivos y Protocolos

En el editor de escenarios se efectuará la persistencia del nivel creado. El mismo deberá respetar las convenciones utilizadas en el servidor. Es decir, que se creará un archivo descriptor de nivel llamado *level.xml* y otro que será la imagen de fondo del mismo, llamado *background.img*.

REPRESENTACION EN XML DE UN ESCENARIO

```

<?xml version="1.0" ?>
<Stage gravity="" maxtime="" a="" b="" c="" d="">
  <MassProperties mass="" maxradius="" minradius="" number="" price="" />
  <MetalBarProperties maxlength="" minlength="" number="" price="" />
  <RopeProperties maxlength="" minlength="" number="" price="" />
  <WheelProperties torque="" decaytime="" maxradius="" minradius="" number="" price="" />
  <MetalPlatformProperties maxlength="" minlength="" number="" price="" />
  <CanvasTapeProperties maxlength="" minlength="" number="" price="" />
  <RocketProperties decaytime="" strength="" number="" price="" />
  <FixedPointProperties maxradius="" minradius="" number="" price="" />
  <Ball posx="" posy="" radiusposx="" radiusposy="" radius="" maxradius="" minradius="" />
  <ExitAreaCollection>
    <ExitArea posx="" posy="" radiusposx="" radiusposy="" radius="" maxradius="" minradius="" />
    ...
  </ExitAreaCollection>
  <FixedPointCollection>
    <FixedPoint posx="" posy="" radiusposx="" radiusposy="" radius="" maxradius="" minradius="" />
    ...
  </FixedPointCollection>
  <ObstacleCollection>
    <Obstacle startposx="" startposy="" endposx="" endposy="" maxlength="" minlength="" />
  </ObstacleCollection>
</Stage>
  
```

Los tags coinciden con los nombres de las clases que representan a las entidades, los atributos de cada tag coinciden con los nombres de los atributos de las clases.



Servidor

Descripción General

El servidor es quien, como la palabra lo indica, *sirve* a los jugadores conectados. Es decir, será el encargado de proveer el ambiente de juego³ como así también de proporcionar un medio para la comunicación entre los distintos jugadores. Asimismo, tendrá la vital funcionalidad de mantener sesiones de juego, permitiendo de esta manera que existan distintos grupos de jugadores ya sea en un juego multiusuario o bien, en un juego monousuario.

Por otro lado, el servidor deberá enviar señales y notificaciones a los jugadores respecto de eventos que sucedan. Por ejemplo, finalización de un nivel, unión de un jugador dentro de una sesión y otros que veremos luego.

Clases

A continuación se detallarán las clases y métodos principales de este módulo por espacio de nombres (*namespace*).

SERVER::GAME

GameServer

Esta clase encapsula la conectividad relacionada al juego.

```
/**
 * Inicia la ejecución del Game Server en la IP y puerto parametrizados.
 * @param ip La IP donde se iniciara el Server.
 * @param port El puerto donde se iniciara el Server.
 * @throws SocketException en caso de error con el socket.
 */
virtual void startServer(const std::string& ip,
                        unsigned short port = GameServer::DEFAULT_PORT);

/**
 * Inicia la ejecución del Game Server en la IP local y el puerto
 * parametrizado.
 * @param port El puerto donde se iniciara el Server.
 * @throws SocketException en caso de error con el socket.
 */
virtual void startServer(unsigned short port = GameServer::DEFAULT_PORT);

/**
 * Detiene el funcionamiento del Game Server y libera los recursos
 * asociados.
 */
virtual void shutdownServer();
```

SERVER::GAME::LEVELS

LevelManager

Esta clase efectúa el manejo de niveles dentro de una sesión.

³ Llamamos ambiente de juego a la descripción del nivel junto con la imagen de fondo asociada al mismo.



```
/**
 * Prepara el siguiente nivel para ser devuelto.
 */
void prepareNextLevel();

/**
 * Devuelve el nivel actual.
 * @return El nivel actual.
 */
Level getCurrentLevel() const;

/**
 * Marca el nivel actual como enviado.
 */
void markCurrentLevelAsSent();

/**
 * Devuelve la cantidad de veces que fue enviado el nivel actual.
 * @return La cantidad de veces que fue enviado el nivel actual.
 */
int getLevelTimesSent() const;
```

Level

Esta clase representa al nivel en sí, contiene el tamaño y la ubicación tanto del descriptor del nivel como de la imagen de fondo.

```
/**
 * Retorna el número de nivel.
 * @return El numero de nivel.
 */
int getLevelNumber() const;

/**
 * Retorna la longitud del archivo de nivel.
 * @return La longitud del archivo de nivel.
 */
long getLevelFileSize() const;

/**
 * Retorna el path al archivo de nivel.
 * @return El path al archivo de nivel.
 */
const std::string& getLevelFilePath() const;

/**
 * Retorna la longitud de la imagen de fondo del nivel.
 * @return La longitud de la imagen de fondo del nivel.
 */
long getLevelBackgroundSize() const;

/**
 * Retorna el path a la imagen de fondo del nivel.
 * @return El path a la imagen de fondo del nivel.
 */
const std::string& getLevelBackgroundPath() const;
```

SERVER::GAME::SESSIONS

GameSessionManager

Esta clase efectúa el manejo de todas las sesiones existentes en el servidor.



```
/**
 * Crea una nueva Sesión de Juego para un jugador y la retorna.
 * @return La Sesión de Juego creada.
 */
GameSession* createSinglePlayerGameSession();

/**
 * Crea una nueva Sesión de Juego para más de un jugador y la retorna.
 * @return La Sesión de Juego creada.
 */
GameSession* createMultiPlayerGameSession();

/**
 * Retorna una Sesión de Juego dado su Id.
 * @param gameSessionId El Id de la Sesión de Juego.
 * @return La Sesión de Juego. NULL si no existe.
 */
GameSession* getSessionById(int gameSessionId);

/**
 * Retorna una lista con todas las Sesiones de Juego existentes.
 * @return Todas las Sesiones de Juego existentes.
 */
const std::list<GameSession*>& getGameSessions();

/**
 * Agrega un Cliente a la lista de espera para ser asignado en una
 * sesión.
 * @param gameClient El cliente de Juego.
 */
void addAwaitingClient(server::game::client::GameClient* gameClient);

/**
 * Agrega un cliente a una sesión de Juego dado el id de la sesión.
 * Elimina al cliente de la lista de espera.
 * @param gameClient El cliente a ser agregado.
 * @param gameSessionId El Id de la Sesión de Juego.
 */
void addClientToGameSession(server::game::client::GameClient* gameClient,
                             GameSession* gameSession);

/**
 * Determina si un nick está siendo utilizado en alguna sesión.
 * @return Verdadero si el nick está en uso. Falso, si no.
 */
bool isNickInUse(const std::string& nick) const;

/**
 * Desconecta todas las sesiones de Juego.
 */
void disconnectAllGameSessions();
```

GameSession

Esta clase representa la sesión en sí y se encarga del manejo de los detalles propios de la misma.



```
/**
 * Agrega un cliente a la sesion.
 * @param gameClient El Cliente a ser agregado a la sesion.
 */
void addClient(server::game::client::GameClient* gameClient);

/**
 * Retorna si la Sesion de Juego esta vacia.
 * @return Verdadero si la Sesion esta vacia. Sino, falso.
 */
bool isEmpty() const;

/**
 * Retorna si la sesion es multijugador o no.
 * @return Si la sesion es multijugador o no.
 */
bool isMultiplayer() const;

/**
 * Retorna si la sesion tiene la cantidad maxima de usuarios.
 * @return Si la sesion tiene la cantidad maxima de usuarios.
 */
bool isFull() const;

/**
 * Retorna los clientes conectados a la sesion.
 * @return Los clientes conectados a la sesion.
 */
const std::vector<server::game::client::GameClient*> & getGameClients()
const;

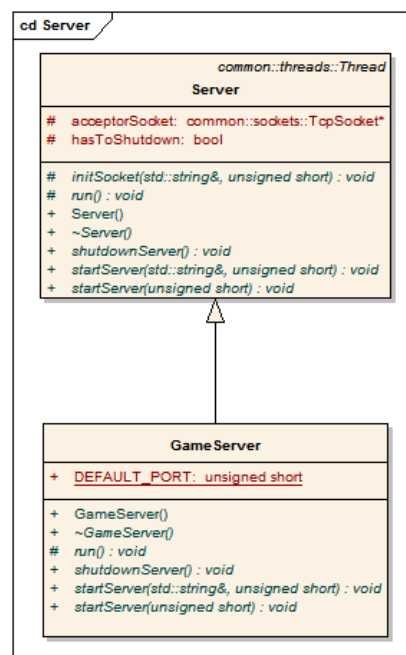
/**
 * Remueve aquellos clientes que no estan conectados.
 * Retorna si la Sesion ha quedado vacia.
 * @return Verdadero si la Sesion quedo vacia. Sino, falso.
 */
bool removeDisconnectedClients();

/**
 * Envia un Mensaje a los usuarios conectados en la sesion de Juego.
 * @param message El mensaje a ser enviado.
 */
void broadcastMessage(const std::string& message) const;

/**
 * Determina si un nivel debe comenzar o no.
 */
bool hasLevelToStart();
```

Diagramas UML

SERVIDOR

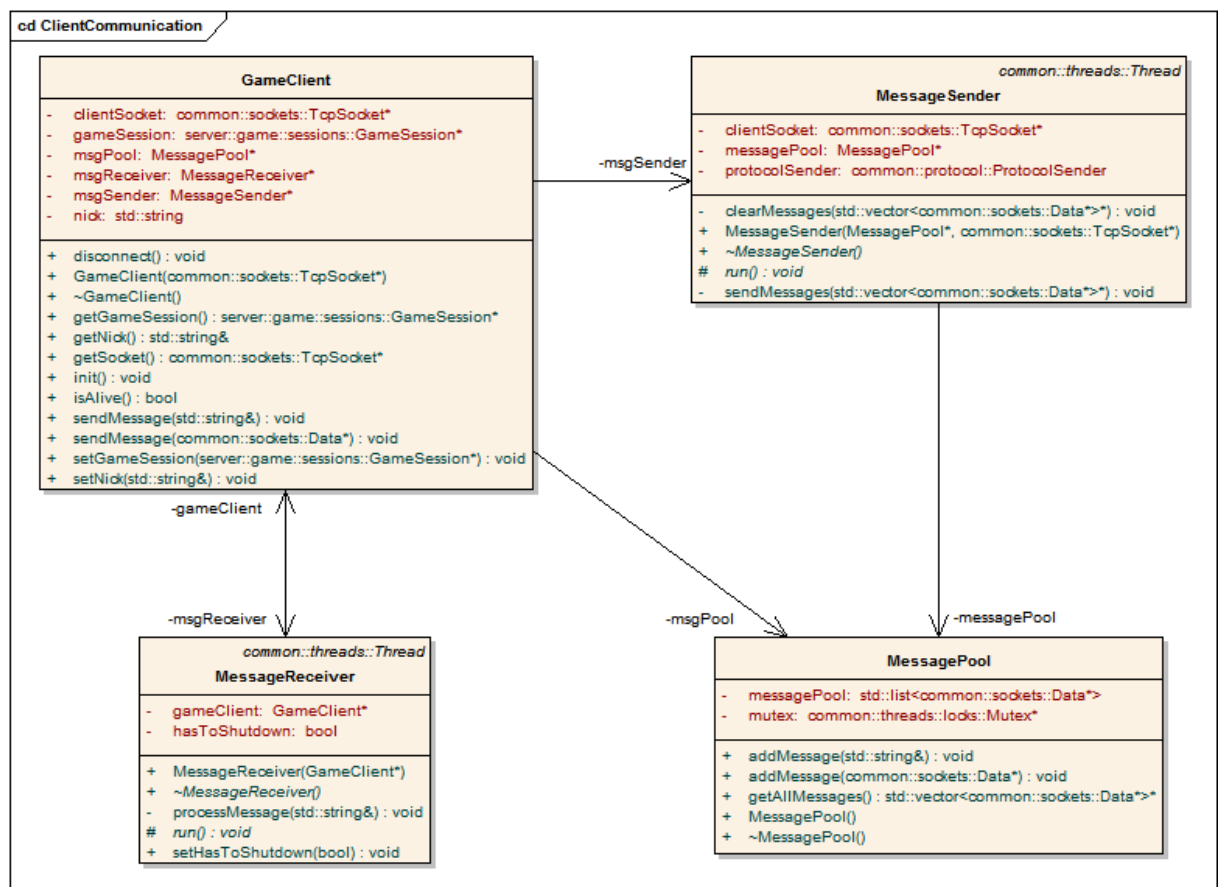




Este diagrama nos muestra la clase abstracta *Server* y su clase heredera *GameServer*. La primera contiene aspectos de conectividad genéricos, mientras que la segunda se especializa en los aspectos de conexión relacionados al juego: agregado de clientes en espera, unión de clientes a sesiones, etc.

Originalmente existía un *ChatServer* que utilizaba un *socket* distinto al *GameServer*. No nos pareció una mala idea el hecho de separar conversaciones del tráfico del juego. Luego se decidió optar por este modelo más sencillo (vale aclarar también que el ayudante que nos acompañó durante el desarrollo del presente trabajo discrepó del hecho de poseer dos *sockets*).

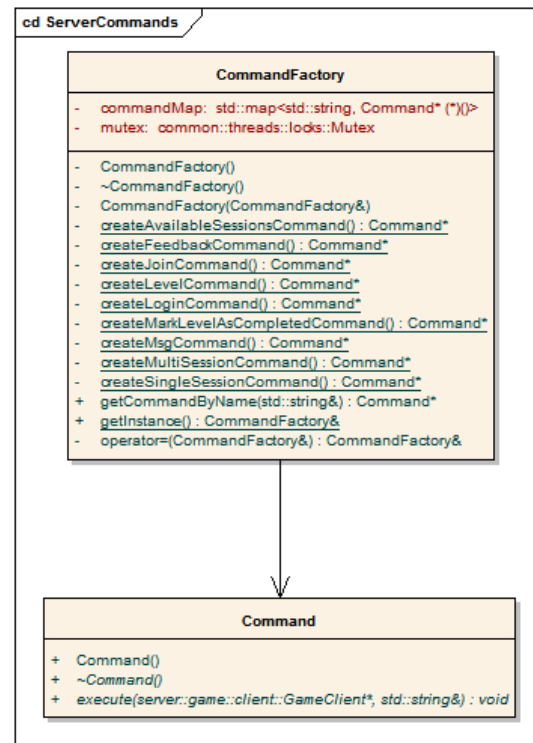
COMUNICACIÓN CON EL CLIENTE



Este diagrama expone la manera en que se maneja la comunicación con el cliente. Podemos observar que tanto el *MessageReceiver* como el *MessageSender* corren cada uno en un hilo aparte. El primero es quien recibe los mensajes del cliente, mientras que el segundo es quien efectúa los envíos. Asimismo, existe un *pool* de mensajes, el cual es consumido por el *MessageSender* para enviarlos al cliente en cuestión.



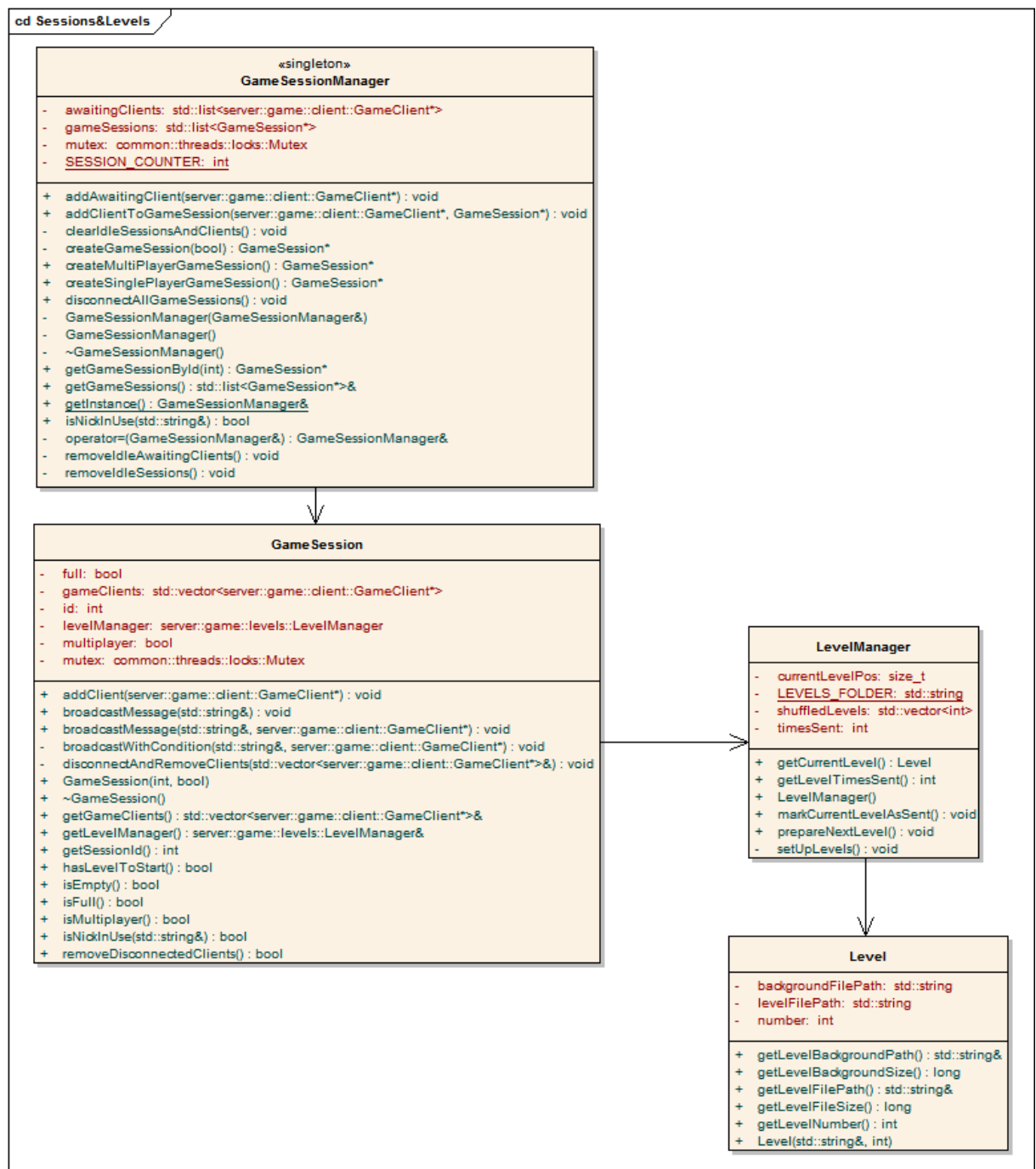
COMANDOS



Cada comando que es ejecutado por el servidor, normalmente proviene de una petición o señal del cliente. Esta petición es analizada y en base a ella se crea el comando necesario para procesarla. En este diagrama se expone cómo se lleva a cabo la creación de comandos. Es decir, esto se hace a través de un *Factory* de Comandos. Cada comando hereda de la clase abstracta *Command* y además posee un nombre a través del cual se lo puede obtener con una simple llamada al método *getCommandByName* del *Factory*.



MANEJO DE SESIONES Y NIVELES



Este diagrama involucra el manejo de sesiones y niveles. El *GameSessionManager* es el encargado de manejar todas las sesiones abiertas en el servidor: las crea, las mantiene, conoce su estado y, eventualmente, las elimina. La clase *GameSession* es quien maneja la sesión en sí: agrega clientes, realiza *broadcast* de mensajes a todos los clientes conectados, elimina clientes que se desconectaron. Cada sesión contiene un *LevelManager* que será el encargado de manejar los niveles: armará una sucesión de niveles aleatoria y los irá devolviendo a la sesión.



Descripción de Archivos y Protocolos

NIVELES

El servidor contendrá una serie de niveles precargados que serán enviados a los distintos jugadores conectados. Estos niveles deberán estar contenidos en una estructura específica y además respetar una convención para los nombres. Si la estructura o convención de nombres no es respetada, el comportamiento del servidor será completamente inestable, pudiendo finalizar abruptamente.

Estructura de Carpetas

```
└─ {Directorio de Instalación del Servidor}
    └─ levels
        └─ 1
        └─ 2
        └─ ...
        └─ {Nivel enésimo}
```

Convención de Nombres

Dentro de cada carpeta correspondiente a un nivel habrá dos archivos: uno de ellos será la imagen de fondo del nivel y el otro será el archivo XML de descripción del nivel.

```
└─ {Nivel enésimo}
    └─ background.img
    └─ level.xml
```

PROTOCOLO DE COMUNICACIÓN

Tanto el cliente como el servidor se comunican entre sí a través de comandos. Estos comandos permitirán al cliente efectuar solicitudes o bien ejecutar acciones en el servidor. Por otro lado, el servidor responderá a estos pedidos con los correspondientes comandos de respuesta. Asimismo, podrá enviar señales al cliente utilizando comandos destinados para tal fin.

Los mensajes serán enviados respetando el protocolo general de la aplicación (Ver sección *Biblioteca Común*, en página 26).

Aquí, la sección [data] del protocolo general dependerá del comando que se ejecute.

Envío de un Mensaje

El objetivo de este comando es el envío de un mensaje a un determinado cliente. Puede ser utilizado para reenviar mensajes de *chat* enviados al resto de los jugadores o bien puede ser utilizado para enviar mensajes de aviso respecto de eventos que sucedan. Por ejemplo: Notificar que un jugador se ha conectado a la sesión de juego.

Formato

```
[data] = "/msg"\30'[nick]\30'[message]
"/msg" = Literal (Tipo: char*).
'\30' = Es el carácter ASCII no imprimible, Record Separator.
[nick] = Es el nick del usuario remitente (Tipo: char*).
'\30' = Es el carácter ASCII no imprimible, Record Separator.
```



[message] = Es el mensaje en sí (Tipo: char*).

Comando Login

El objetivo de este comando es notificar el resultado de la operación al jugador que ha intentado loguearse al servidor.

Formato

```
[data] = "/login" [resultado]
"/login" = Literal (Tipo: char*).
[resultado] = Es el resultado de la petición de login (Tipo: char).
Los posibles resultados pueden ser:
    "0" (Error por nickname duplicado)
    "1" (Éxito)
```

Comando de Unión a Sesión

Este comando está destinado a notificarle al jugador el resultado de la petición de unión a una sesión. Asimismo, es utilizado para notificar a los jugadores de una sesión que un nuevo jugador se ha unido a la misma.

Formato

```
[data] = "/join" [resultado]
"/join" = Literal (Tipo: char*).
[resultado] = Es el resultado de la petición de unión (Tipo: char*).
Los posibles resultados pueden ser:
    "0" (Error: La sesión ha alcanzado la cantidad máxima de jugadores permitida)
    "1" (Éxito)
    "2" (Notificación de conexión de un oponente)
```

Comando de Devolución de Sesiones Disponibles

Este comando devolverá las sesiones disponibles a un jugador. Las sesiones disponibles son aquellas que tienen la particularidad de ser Multijugador y además no haber superado el límite de jugadores permitido⁴.

Formato

```
[data] = "/sessions" [sessionList]
"/sessions" = Literal (Tipo: char*).
[sessionList] = sessionId'\30'sessionName'\30'
    sessionId = Es el ID de la sesión (Tipo: char*)
    sessionName = Es el nombre identificativo de la sesión (Tipo: char*)
```

Comando de Envío de Niveles

Este comando está destinado a efectuar el envío de niveles a un jugador. Se divide en tres partes: *LevelHeader*, *BackgroundImage* y *XmlDescriptor*. La primera parte enviará el encabezado del nivel, luego se enviará la imagen de fondo del nivel y por último el archivo descriptor del nivel.

Formato LevelHeader

```
[data] = "/level" [levelID]
"/level" = Literal (Tipo: char*).
[levelID] = Es el ID del Nivel.
Los posibles valores son:
    "0" (Error: No hay más niveles disponibles)
    "Número de Nivel" (Es el número de nivel que se está enviando)
```

Formato BackgroundImage

```
[data] = "/img" [imgSize] [image]
"/img" = Literal (Tipo: char*).
[imgSize] = Es el tamaño en bytes de la imagen de fondo (Tipo: char*).
[image] = Es la imagen enviada como tira de bytes (Tipo: char*).
```

Formato XmlDescriptor

⁴ Hemos hecho hincapié en la genericidad del código para permitir jugar entre más de dos personas. De todas maneras, el juego está configurado para que puedan enfrentarse como mucho dos jugadores.



```
[data] = "/xml" [xmlSize] [xmlFile]
"/xml" = Literal (Tipo: char*).
[xmlSize] = Es el tamaño en bytes del archivo XML (Tipo: char*).
[xmlFile] = Es el archivo XML enviado como tira de bytes (Tipo: char*).
```

Comando de Comienzo de un Nivel

Una vez que estén dadas las condiciones⁵ para comenzar un nivel, el servidor utilizará esta señal para dar aviso a los jugadores de que el nivel puede iniciarse.

Formato

```
[data] = "/levelStart"
"/levelStart" = Literal (Tipo: char*).
```

Comando de envío de Feedback

Este comando tiene como fin el envío del diseño⁶ del oponente a los integrantes de una sesión. El servidor recibirá de un jugador el diseño y este será enviado al resto de los jugadores que integran la sesión.

Formato

```
[data] = "/feedback [xml]"
"/feedback" = Literal (Tipo: char*).
[xml] = Es un XML conteniendo el detalle de la disposición de los elementos en pantalla (Tipo: char*).
```

⁵ El nivel se debe haber terminado de transferir a todos los jugadores.

⁶ Con “diseño” nos referimos a la disposición de elementos o herramientas en la pantalla de juego.



Cliente

Descripción General

El cliente es quien permitirá al usuario poder jugar a “The Astounding Machine” ofreciéndole una interfaz amigable, posibilitando crear una partida solitaria, crear una partida multijugador o bien unirse a una partida existente. Esto lo hará comunicándose con un servidor cuya IP o Nombre deberá ser especificado.

Clases

A continuación se detallarán las clases y métodos principales de este módulo por espacio de nombres (*namespace*).

CLIENT::GAME::FACADE

CommandFacade

```
/**
 * Ejecuta una solicitud login en el server utilizando el nombre
 * indicado como parametro.
 * @param loginName El nombre utilizado para loguearse.
 */
static void login(const std::string& loginName);

/**
 * Ejecuta una solicitud de sesiones disponibles al servidor.
 */
static void getAvailableSessions();

/**
 * Ejecuta una solicitud de union a la sesion indicada como
 * parametro.
 * @param sessionId El Id de sesion al cual quiere unirse.
 */
static void joinSession(const std::string& sessionId);

/**
 * Ejecuta una solicitud de creacion de Sesion Monousuario.
 */
static void createSinglePlayerSession();

/**
 * Ejecuta una solicitud de creacion de Sesion Multiusuario.
 */
static void createMultiPlayerSession();

/**
 * Ejecuta una solicitud del siguiente nivel disponible.
 */
static void getNextLevel();

/**
 * Envia un mensaje de chat.
 * @param message El mensaje a ser enviado.
 */
static void sendChatMessage(const std::string& message);

/**
 * Ejecuta una solicitud al servidor para marcar el nivel actual como
 * completado enviando el puntaje obtenido en el mismo.
 * @param score El puntaje obtenido en el nivel.
 */
static void markLevelAsCompleted(const std::string& score);

/**
 * Ejecuta una solicitud al servidor para marcar el nivel actual como
 * NO completado.
 */
static void markLevelAsFailed();

/**
 * Envia el feedback (disposicion de los elementos en pantalla) al
 * servidor.
 * @param feedbackStr Cadena de caracteres conteniendo el feedback.
 */
static void sendFeedback(const std::string& feedbackStr);
```



CLIENT::GAME::MODEL

GameClient

```
/**
 * Conecta el cliente a un Game Server utilizando la ip y puerto
 * especificados.
 * @param ip La IP del Game Server.
 * @param port Opcional. El puerto del Game Server.
 * @throws ConnectionException en caso de error en la conexion.
 */
void connectToGameServer(const std::string& ip,
    unsigned short port = DEFAULT_PORT);

/**
 * Retorna si una respuesta del servidor ha sido recibida.
 * @return Verdadero si ha llegado una respuesta del servidor.
 * Falso, si no.
 */
bool wasResponseReceived() const;

/**
 * Retorna la lista de Sesiones disponibles.
 */
std::list<Session*>* getAvailableSessions() const;

/**
 * Retorna si el cliente esta logueado o no.
 * @return Verdadero si el cliente esta logueado. Falso, si no.
 */
bool isLogged() const;

/**
 * Retorna si la union fue exitosa o no.
 * @return Verdadero si la union fue exitosa. Falso, si no.
 */
bool isJoined() const;

/**
 * Retorna si el oponente esta conectado.
 * @return Verdadero si el oponentes esta conectado. Falso, si no.
 */
bool isOpponentConnected() const;

/**
 * Retorna si el nivel debe comenzar.
 * @return Verdadero si el nivel debe comenzar. Falso, si no.
 */
bool hasLevelToStart() const;

/**
 * Retorna si hay mas niveles.
 * @return Si hay mas niveles.
 */
bool hasMoreLevels() const;

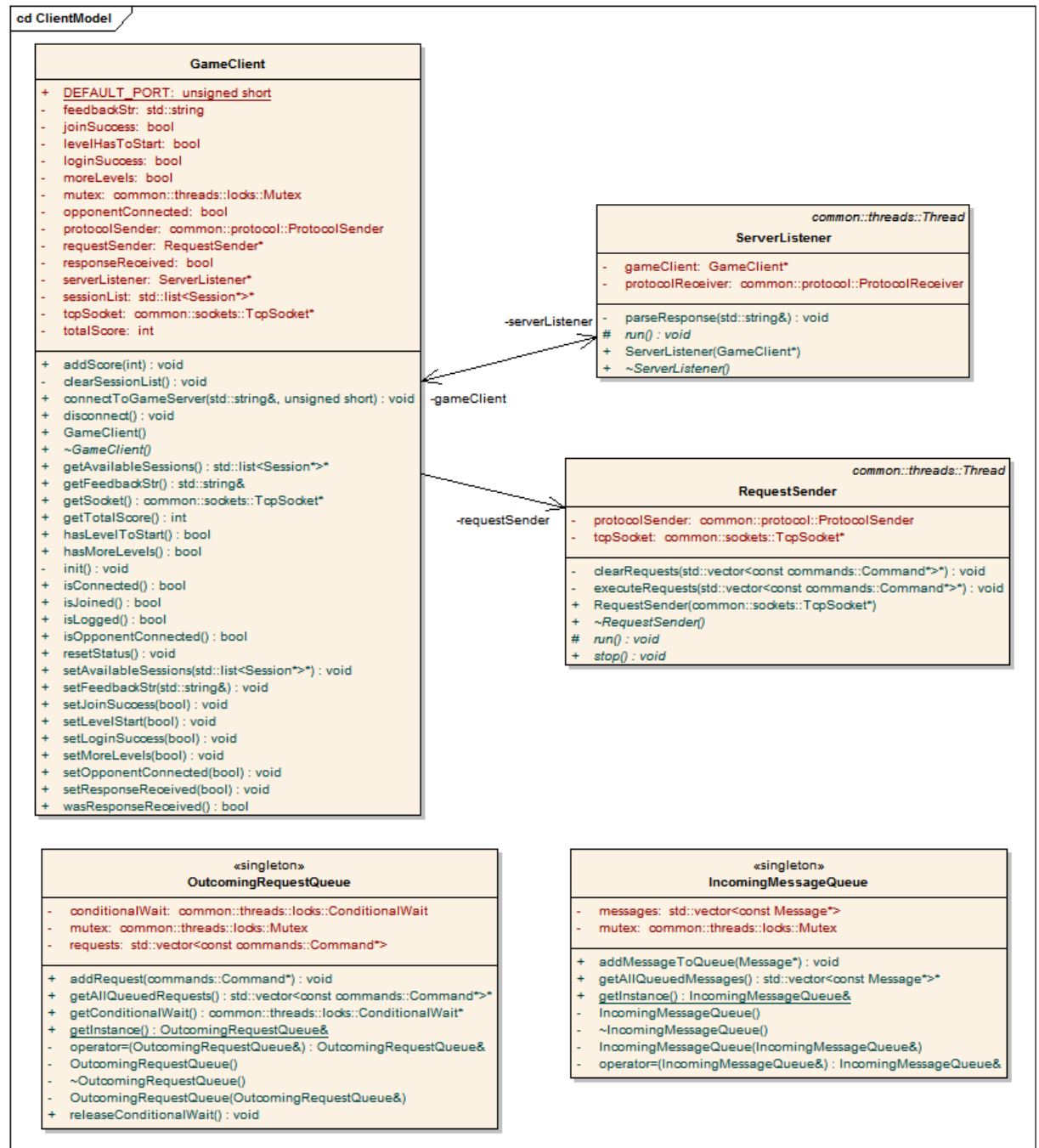
/**
 * Agrega un puntaje parcial al puntaje Total.
 * @param score El puntaje total a ser agregado.
 */
void addScore(int score);

/**
 * Devuelve el puntaje total.
 * @return El puntaje total.
 */
int getTotalScore() const;

/**
 * Retorna la cadena de caracteres conteniendo el feedback del
 * oponente.
 * @return La cadena de caracteres conteniendo el feedback del
 * oponente.
 */
const std::string& getFeedbackStr() const;
```

Diagramas UML

MODELO



Este diagrama representa el modelo de la aplicación Cliente. *GameClient* es quien contendrá los datos de los envíos del servidor para que puedan ser accedidos por el controlador de la vista.

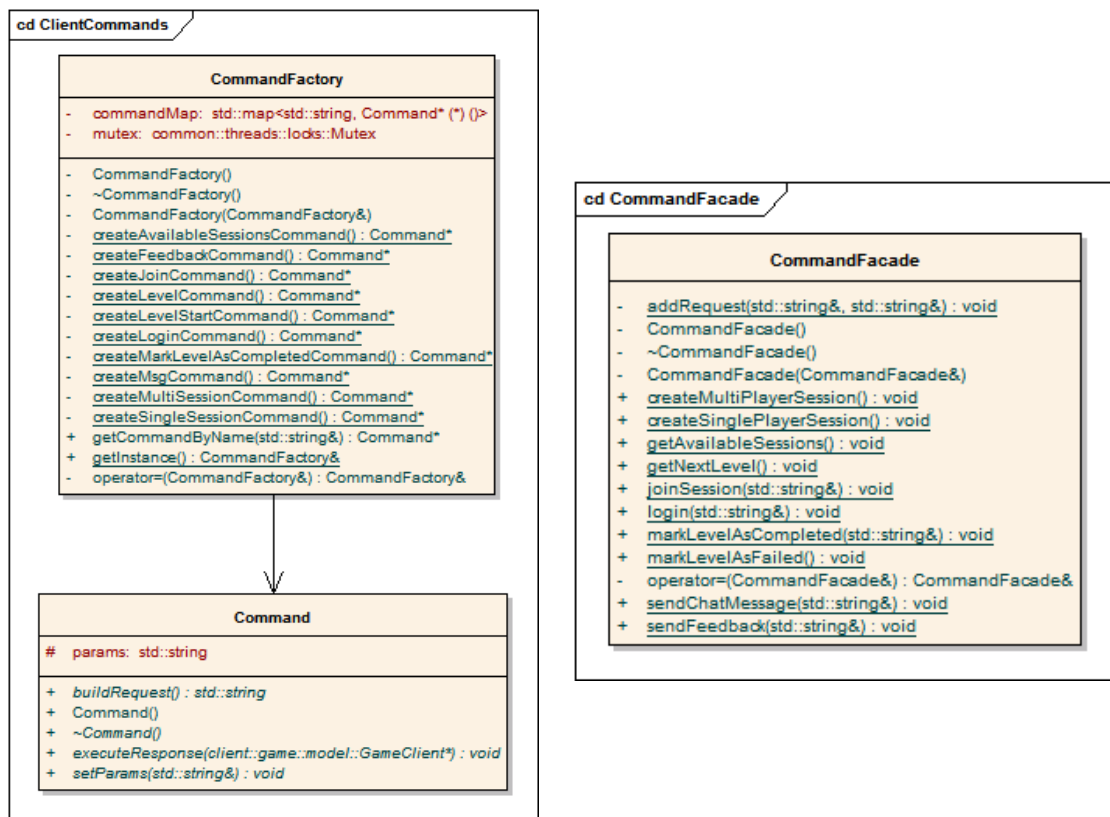
Por otro lado, nos encontramos con *ServerListener* y *RequestSender* que se encargan de la comunicación con el servidor, corriendo cada uno en un hilo separado. El primero de ellos escucha los envíos del servidor mientras que el segundo efectúa los envíos al mismo.

Asimismo, podemos observar dos clases Singleton que corresponden a colas de mensajes y pedidos: *IncomingMessageQueue* y *OutcomingRequestQueue*. La primera de ellas contiene los



mensajes entrantes, ya sea de chat o de notificaciones del servidor. La segunda de ellas corresponde a las solicitudes salientes.

COMANDOS



Los comandos en el cliente son utilizados tanto como para enviar solicitudes al servidor como para ejecutar las respuestas que este envíe. En el diagrama se observa algo análogo a lo que se tiene en el servidor respecto de la creación de comandos: un *Factory* que se encarga de crear comando en base a su nombre. Cada comando implementará la clase abstracta *Command* para sobrescribir los métodos *buildRequest* (arma la solicitud para el servidor) y *executeResponse* (ejecuta una acción en base a la respuesta recibida del servidor).

Por otro lado, podemos observar el *CommandFacade*. Su objetivo es organizar y simplificar la ejecución de comandos. Es decir, evita tener que lidiar con la creación de comandos y la configuración de parámetros.

CONTROLADORES DE VISTA

Los controladores de la vista son clases que se encargan de manejar los eventos de la interfaz de usuario. Tendremos dos controladores, uno para la pantalla de inicio del juego, que es la que tiene como fin la conexión al servidor y la creación de una sesión o bien la unión a una sesión ya existente; y por otro lado, la pantalla del juego en sí.





Descripción de Archivos y Protocolos

NIVELES

El cliente contendrá una carpeta destinada a almacenar temporalmente los niveles que el servidor le envíe. Esta carpeta deberá existir para garantizar el correcto funcionamiento del Cliente. Cada vez que se reciba un nivel, el anterior será sobrescrito dado que se respeta la misma convención de nombres.


Estructura de Carpetas


 {Directorio de Instalación del Cliente}


 tmp

Convención de Nombres

Dentro de la carpeta temporal de niveles habrá dos archivos: uno de ellos será la imagen de fondo del nivel y el otro será el archivo XML de descripción del nivel. Ambos serán enviados por el servidor.

 tmp

 background.img

 level.xml

PROTOCOLO DE COMUNICACIÓN

Comando para Login en el Servidor

Este comando solicitará el *login* en el servidor.

Formato

[data] = `"/login" [nick]`
`"/login"` = Literal (Tipo: char*).
[nick] = Es el nick con el que se *logueará* el jugador.

Comando para Envíos de Mensajes de Chat

El objetivo de este comando es enviar mensajes de chat a los distintos usuarios conectados.

Formato

[data] = `"/msg" [nick]'\30'[message]`
`"/msg"` = Literal (Tipo: char*).
[nick] = Es el nick del usuario (Tipo: char*).
`'\30'` = Es un carácter ASCII no imprimible llamado Record Separator.
[message] = Es el mensaje en sí (Tipo: char*).

Comando de Solicitud de Sesiones Disponibles

Este comando solicitará al servidor las sesiones disponibles.

Formato

[data] = `"/sessions"`
`"/sessions"` = Literal (Tipo: char*).

Comando de Creación de Sesiones Solitarias

Este comando creará una partida o sesión solitaria en el servidor.

Formato

[data] = `"/createSingle"`



`"/createSingle" = Literal (Tipo: char*).`

Comando de Creación de Sesiones Multijugador

Este comando creará una partida o sesión multijugador en el servidor.

Formato

`[data] = "/createMulti"`
`"/createMulti" = Literal (Tipo: char*).`

Comando de Unión a Sesión

Este comando tiene como objetivo solicitar al servidor la unión de un jugador a una determinada sesión multijugador.

Formato

`[data] = "/join" [sessionId]`
`"/join" = Literal (Tipo: char*).`
`[sessionId] = Es el ID de la sesión a la cual se quiere unir al jugador (Tipo: char*).`

Comando de Solicitud de un Nivel

Este comando solicitará al servidor la entrega del nivel actual.

Formato

`[data] = "/level"`
`"/level" = Literal (Tipo: char*).`

Comando de Notificación de Finalización de un Nivel

Este comando notificará al servidor que el jugador ha finalizado un nivel.

Formato

`[data] = "/levelCompleted" [score]`
`"/levelCompleted" = Literal (Tipo: char*).`
`[score] = Es el puntaje que obtuvo el jugador al completar el nivel. En caso de que no haya podido completarlo se enviará este campo con un valor de "-1" (Tipo: char*).`

Comando de envío de Feedback

Este comando tiene como finalidad el envío al servidor de la disposición de los elementos en la pantalla de juego. Es decir, del diseño de la solución del nivel. Este comando será ejecutado cada cierta cantidad de tiempo y será enviado al servidor para que éste a su vez notifique al resto de los participantes de la sesión.

Formato

`[data] = "/feedback [xml]"`
`"/feedback" = Literal (Tipo: char*).`
`[xml] = Es un XML conteniendo el detalle de la disposición de los elementos en pantalla (Tipo: char*).`



Biblioteca Común

Descripción General

Esta biblioteca fue creada con el fin de ser compartida por todos los módulos dado que contiene una serie de clases útiles que resuelven distintas cuestiones presentes a lo largo de todo el proyecto.

Clases

A continuación se detallarán las clases y métodos principales de este módulo por espacio de nombres (*namespace*).

LOGGER

En este *namespace* encontraremos el código fuente de la biblioteca *Trivial C++ Logger*.

COMMON::PROTOCOL

En este *namespace* se ubicaron aquellas clases que encapsulan el protocolo de intercambio general de la aplicación.

ProtocolSender

Esta clase encapsula los envíos de datos utilizando el protocolo de intercambio general.

```
/**
 * Envía una cadena a un Socket utilizando el protocolo.
 * @param tcpSocket El Socket al cual se le enviara el mensaje.
 * @param message El mensaje a ser enviado.
 */
void sendToSocket(const common::sockets::TcpSocket& tcpSocket,
                 const std::string& message) const;

/**
 * Envía Datos a un Socket utilizando el protocolo.
 * @param tcpSocket El Socket al cual se le enviara el mensaje.
 * @param data Los datos a ser enviados.
 */
void sendToSocket(const common::sockets::TcpSocket& tcpSocket,
                 const common::sockets::Data& data) const;
```

ProtocolReceiver

Esta clase encapsula las recepciones de datos utilizando el protocolo de intercambio general.

```
/**
 * Recibe Datos de un Socket utilizando el protocolo.
 * @param tcpSocket El Socket del cual se recibirán los datos.
 * @return Los Datos recibidos del Socket.
 */
const common::sockets::Data receiveFromSocket(
    const common::sockets::TcpSocket& tcpSocket) const;

/**
 * Recibe una cadena de un Socket utilizando el protocolo.
 * @param tcpSocket El Socket del cual se recibirán los datos.
 * @return La cadena recibida del Socket.
 */
const std::string receiveStrFromSocket(
    const common::sockets::TcpSocket& tcpSocket) const;
```



COMMON::SOCKETS

Este *namespace* contiene aquellas clases cuyo objetivo es el manejo de *sockets* e intercambio de datos a través de ellos.

Data

Esta clase tiene como fin el encapsulamiento de datos para su intercambio. Se trata de una clase inmutable. Es decir, una vez construido el objeto, su contenido no es modificable.

```
/**
 * Constructor.
 * Recibe los datos y la longitud de los mismos.
 * Se efectúa una copia de los datos según el parámetro
 * copyData lo indique.
 * @param data Los datos.
 * @param length La longitud de los datos.
 * @param copyData Indica si deben copiarse o no los datos.
 */
Data(const char* data, int length, bool copyData = true);
/**
 * Constructor.
 * Recibe una cadena de caracteres.
 * Se efectúa una copia de ellos.
 * @param str La cadena de caracteres.
 */
Data(const std::string& str);
/**
 * Destructor.
 * Libera el puntero a char contenido.
 */
~Data();
/**
 * Retorna el puntero a char contenido.
 * @return El puntero a char contenido.
 */
const char* getData() const;
/**
 * Retorna la longitud de los datos contenidos.
 * @return La longitud de los datos contenidos.
 */
int getLength() const;
```

TcpSocket

Esta clase encapsula el manejo íntegro de las operaciones que pueden efectuarse a través de un *socket*.



```
/**
 * Asigna el socket para que funcione sobre la IP y puerto
 * parametrizados.
 * @param ipAddress La IP a la que se asignara el socket.
 * @param port El puerto al que se asignara el socket.
 * @throws TcpSocketException En caso de que ocurra algún error con el
 * Socket.
 */
void bindSocket(const std::string& ip, unsigned short port) const;
/**
 * Asigna el socket para que funcione sobre la IP local y el puerto
 * parametrizado.
 * @param port El puerto al que se asignara el socket.
 * @throws TcpSocketException En caso de que ocurra algún error con el
 * Socket.
 */
void bindSocket(unsigned short port) const;
/**
 * Inicializa la cola de espera recibiendo su tamaño máximo.
 * @param backlog El tamaño máximo de la cola de espera.
 * @throws TcpSocketException En caso de que ocurra algún error con el
 * Socket.
 */
void listenSocket(int backlog = DEFAULT_BACKLOG) const;
/**
 * Conexión a un Host utilizando la ip y puerto parametrizados.
 * @param ipAddress La IP a la que se conectara.
 * @param port El puerto al que se conectara.
 * @throws TcpSocketException En caso de que ocurra algún error con el
 * Socket.
 */
void connectToHost(const std::string& ip, unsigned short port) const;
/**
 * Conexión a un Host utilizando la ip y puerto parametrizados.
 * @param ipAddress La IP a la que se conectara.
 * @param port El puerto al que se conectara.
 * @throws TcpSocketException En caso de que ocurra algún error con el
 * Socket.
 */
void connectToHost(const std::string& ip, unsigned short port) const;
/**
 * Acepta una petición de conexión, devolviendo un socket para
 * comunicarse a través de la conexión aceptada.
 * @throws TcpSocketException En caso de que ocurra algún error con el
 * Socket.
 * @return Un puntero a TcpSocket para permitir la comunicación.
 */
TcpSocket* acceptConnection() const;
/**
 * Enviar un stream de datos.
 * @param data Los datos a enviar.
 * @throws TcpSocketException En caso de que ocurra algún error con el
 * Socket.
 */
void sendData(const Data& data) const;
/**
 * Recibir un stream de datos de longitud parametrizada.
 * @param length La longitud que se desea recibir.
 * @return Los datos recibidos.
 * @throws TcpSocketConnectionClosedException En caso de que el socket
 * remoto haya cerrado la conexión.
 * @throws TcpSocketException En caso de que ocurra algún error con el
 * Socket.
 */
const Data receiveData(int length) const;
/**
 * Desconecta el Socket utilizando el tipo dado.
 * @param type El tipo de Shutdown.
 * @throws TcpSocketException En caso de que ocurra algún error con el
 * Socket.
 */
void shutdownSocket(ShutdownType type) const;
/**
 * Cierra el Socket.
 * @throws TcpSocketException En caso de que ocurra algún error con el
 * Socket.
 */
void closeSocket() const;
```



COMMON::THREADS

Este *namespace* contiene una gran cantidad de clases que tienen como fin el manejo de hilos y temas de concurrencia en general como exclusión mutua y candados de lectura y escritura.

Thread

Esta es la clase principal del *namespace* y encapsula todas las operaciones con hilos.

```
/**
 * Crea e inicia el hilo.
 */
void start();
/**
 * Modifica el estado del hilo a stopped. No cancela la ejecución del
 * mismo.
 */
void stop();
/**
 * Cancela el hilo.
 */
void cancel();
/**
 * Retorna si el hilo fue detenido.
 * @return Verdadero si el hilo esta detenido. Falso, sino.
 */
bool isStopped() const;
/**
 * Retorna el ID del hilo.
 * @return El ID del hilo.
 */
pthread_t getTid() const;
/**
 * Retorna el ID del hilo.
 * @return El ID del hilo.
 */
pthread_t getTid() const;
/**
 * Retorna el lock de lectura/escritura perteneciente al hilo.
 * @return El lock de lectura/escritura.
 */
common::threads::locks::ReadWriteLock& getLock() const;
/**
 * Detiene el hilo que lo invoca hasta la finalización del hilo actual.
 */
void join();
/**
 * Detiene el hilo que lo invoca hasta la finalización del hilo
 * que se recibe como parámetro.
 * @param pthread_t El ID del hilo a ser detenido.
 */
static void join(pthread_t threadId);
/**
 * Retorna el ID del hilo que lo invoca.
 * @return El ID del hilo que invoca el método.
 */
static pthread_t getCurrentThread();
/**
 * Duerme al hilo invocante por (al menos) los milisegundos recibidos
 * como parámetro.
 * @param ms Los milisegundos que debe dormir el hilo invocante.
 */
static void sleep(int ms);
```

COMMON::UTILS

Este *namespace* tiene como objetivo proveer utilidades generales.

Math

Esta clase provee utilidades matemáticas generales.



```
/**
 * Cuenta la cantidad de dígitos que contiene el número.
 * @param number El numero.
 * @return El numero de dígitos.
 */
static char countDigits(int number);
```

StringUtils

Esta clase provee utilidades relacionadas al manejo de cadenas de caracteres.

```
/**
 * Convierte una cadena a un número.
 * El tipo del número esta definido por T.
 * @param str La cadena a ser convertida.
 * @return El numero convertido.
 */
template <typename T>
static T stringToNumber(const std::string& str) {
    std::istringstream stringStream(str);
    T n;
    stringStream >> n;
    return n;
}

/**
 * Convierte un número en una cadena de caracteres.
 * El tipo del número esta definido por T.
 * @param number El numero a ser convertido.
 * @return La cadena de caracteres con el numero convertido.
 */
template <typename T>
static std::string numberToString(T number) {
    std::stringstream stringStream;
    stringStream << number;
    return stringStream.str();
}

/**
 * Retorna si la cadena pasada como parámetro es un número.
 * @param str La cadena a ser evaluada.
 * @return Verdadero si la cadena es numérica. Falso, sino.
 */
static bool isNumeric(const std::string& str) {
    std::istringstream stringStream(str);
    double n = 0;
    return stringStream >> n;
}

/**
 * Retorna una copia de cadena de caracteres recibida agregando el
 * carácter de fin de cadena en caso de que se especifique.
 * @param str La cadena a ser copiada.
 * @param length Longitud de la cadena.
 * @param appendEOS Especifica si debe agregarse o no el carácter de
 * fin de línea a la cadena retornada.
 * @return La copia de la cadena.
 */
static char* stringCopy(const char* str, int length, bool appendEOS);

/**
 * Retorna Verdadero si la cadena de caracteres recibida esta vacía o
 * solo contiene caracteres en blanco. Falso sino.
 * @param La cadena a ser chequeada.
 */
static bool isEmptyOrWhitespace(const std::string& str);
```

FileUtils

Esta clase provee operaciones portables relacionadas con el *FileSystem*.



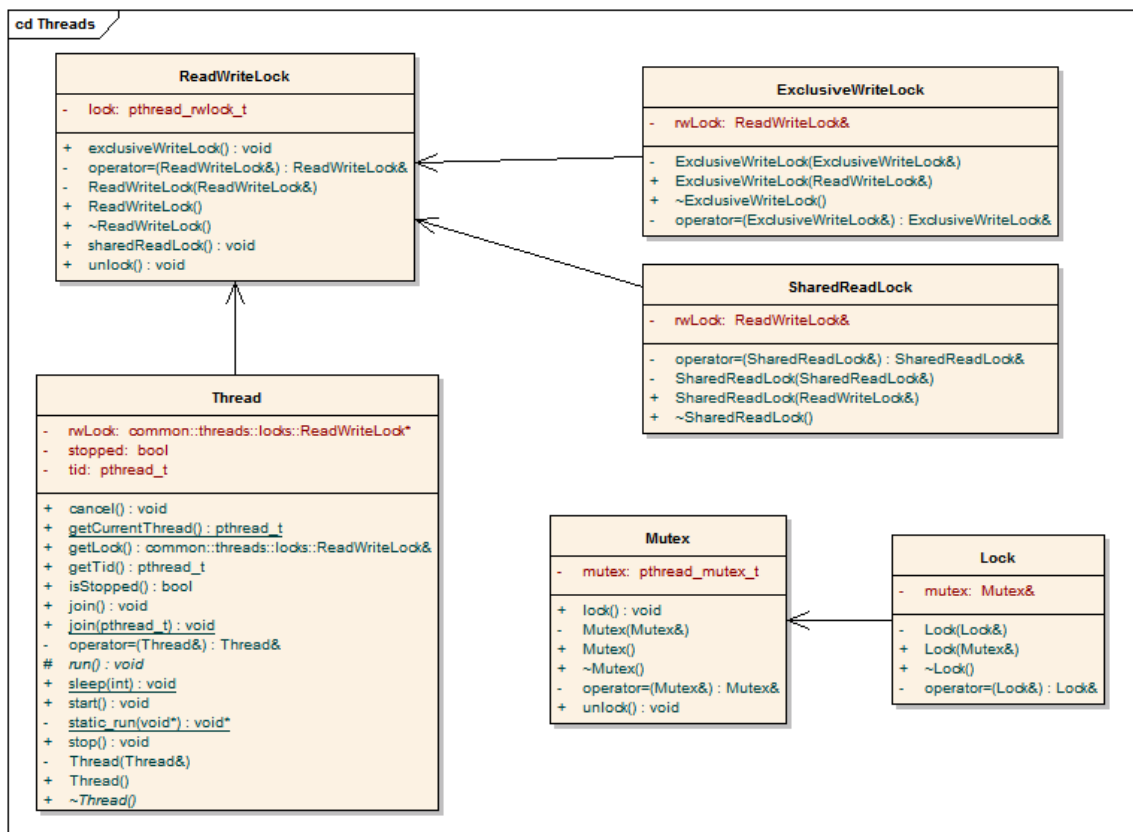
```
/**
 * Retorna la longitud de un archivo dado su dirección.
 * @return La longitud del archivo.
 */
static long getFileSize(const std::string& path);

/**
 * Retorna si un archivo o directorio existe o no.
 * @return Verdadero si el archivo o directorio existe. Sino, falso.
 */
static bool fileExists(const std::string& path);

/**
 * Cuenta la cantidad de archivos o directorios dentro de un directorio
 * base.
 * @param basePath El directorio base a partir del cual buscar.
 * @param excludeDotDirs Excluye los directorios "." y "..".
 */
static int countDirs(std::string& basePath, bool excludeDotDirs);

/**
 * Copia un archivo.
 * @param src Ruta y nombre del archivo de origen.
 * @param dst Ruta y nombre del archivo de destino.
 */
static void copy(const std::string& src, const std::string& dst);
```

Diagramas UML



Descripción de Archivos y Protocolos

PROTOCOLO GENERAL DE INTERCAMBIO

A continuación describiremos el protocolo general de intercambio utilizado en la aplicación.



Cada vez que se efectúe un envío será realizado de la siguiente manera:

```
[header][data]
[header] = [digits][length]
[digits] = Cantidad de dígitos que posee la longitud del mensaje (Tipo: char).
[length] = Longitud del mensaje (Tipo: char*).
[data] = Es el mensaje en sí.
```

Ejemplo:

```
[1][4][Hola]
```

Este ejemplo nos indica que la longitud tendrá solamente un dígito y será de cinco bytes y el mensaje en sí es: "Hola".

```
[2][1][4][#%dfkl+!!P@1?]
```

En este caso, la longitud tendrá dos dígitos y será de catorce bytes.

El mensaje será: "#%dfkl+!!P@1?" (representando datos binarios).

¿Por qué decimos que este protocolo es general?

Dado que es el que utilizará todo módulo que necesite enviar datos a través de la red. Luego, cada módulo podrá a su vez implementar *subprotocolos* dentro de éste. Por ejemplo, el módulo de chat podrá utilizar algún *subprotocolo* dentro de lo que es la sección *data* del protocolo general.

¿Por qué se envía la longitud de los datos como un vector de caracteres?

Este tema fue ampliamente discutido entre todos los integrantes del grupo. El problema radica en que C++ ISO 98 no posee ningún tipo numérico estándar de tamaño fijo y por ende, portable. Es decir, un *int* dependiendo de la arquitectura bajo la que corra el programa (32 bits, 64 bits o, inclusive, 128 bits) tendrá distintos tamaños y esto nos causaría muchos inconvenientes a la hora de enviar una variable de este tipo por la red. Una de las soluciones que quisimos implementar fue el hecho de utilizar *uint_32*, pero rápidamente nos dimos cuenta que este tipo de dato forma parte de C99 y ha quedado fuera de C++ ISO 98 (¡por tan sólo un año!).



Biblioteca de Juego

Descripción General

Esta biblioteca define e implementa cuatro funcionalidades relacionadas. En primer lugar, contiene las clases encargadas de realizar los cálculos físicos y matemáticos. Esta funcionalidad está agrupada dentro del espacio de nombres *core*, e incluye el integrador numérico físico, utilidades de cálculo vectorial, utilidades de resolución de intervalos temporales y clases encargadas de representar el modelo físico integrado.

Por otro lado, esta biblioteca expone un conjunto de clases que representan los conceptos propios del juego, como por ejemplo las pelotas, barras y plataformas; así como también contiene las clases que representan ciertas acciones sobre esos elementos, como mover, eliminar y conectar. Esta segunda funcionalidad está agrupada dentro del espacio de nombres *design*.

Además se exponen servicios de manejo de la lógica de juego, como el manejo de puntuación, control de objetivos del nivel y orquestación entre elementos del diseño y los propios de la simulación. Estas clases se encuentran en el espacio de nombres *logic*.

Por último, es en esta biblioteca donde se encuentra la lógica de serialización de los elementos anteriormente mencionados, tanto la asociada a la persistencia de niveles como la propia para el envío del feed del oponente. Las clases relacionadas se encuentran dentro del espacio de nombres *serialization*.

Clases

A continuación se detallarán las clases y métodos principales de este módulo por espacio de nombres (*namespace*).

GAME::CORE

Particle

Esta clase encapsula una partícula dentro del sistema de masas y partículas. Expone las propiedades físicas de la misma, como son su masa, radio, coeficiente de restitución elástico para colisiones, posición, velocidad y aceleración. Cabe destacar que a diferencia del sistema físico de la cátedra, la masa es agnóstica con respecto a las fuerzas que se ejercen sobre ella. Para determinar su aceleración, una masa posee una lista de afectores, que son objetos dedicados a ejercer una fuerza sobre un grupo de masas. De esta forma, la partícula recorre los afectores registrados en ella y obtiene la fuerza provista por cada uno de ellos utilizando llamadas polimórficas. Esto da una gran versatilidad a la hora de la definición de los efectos que ejercen sobre las partículas algunos elementos, y desacopla a la partícula (y por tanto, al sistema físico) de estos detalles.



```
/**
 * Crea una nueva instancia de Particle.
 * @param mass Masa de la nueva partícula
 * @param radius Radio de la nueva partícula
 * @param position Posición de la nueva partícula
 * @param collisionElasticRatio Coef. de restitución.
 */
Particle(double mass, double radius,
         const Vector2d &position = Vector2d(),
         double collisionElasticRatio = 0.0);

/**
 * Obtiene la masa de la partícula.
 * @return Masa de la partícula
 */
double getMass() const;

/**
 * Obtiene el radio de la partícula
 * @return Radio de la partícula
 */
double getRadius() const;

/**
 * Obtiene la posición de la partícula
 * @return Posición de la partícula
 */
const Vector2d &getPosition() const;

/**
 * Obtiene una referencia a la posición de la partícula
 * permitiendo la actualización de la misma sin realizar
 * copias
 * @return
 */
Vector2d &getUpdateablePosition();

/**
 * Obtiene la velocidad de la partícula
 * @return Velocidad de la partícula
 */
const Vector2d& getSpeed() const;

/**
 * Obtiene una referencia a la velocidad de
 * la partícula, permitiendo actualizar la
 * misma sin realizar copias.
 * @return
 */
Vector2d &getUpdateableSpeed();

/**
 * Obtiene la aceleración de la partícula
 * @return Aceleración de la partícula
 */
Vector2d getAcceleration() const;

/**
 * Obtiene si la partícula es o no fija
 * @return true si la partícula es fija, false en caso contrario
 */
bool isFixed() const;

/**
 * Obtiene si la partícula es o no colisionable
 * @return true si la partícula es colisionable, false en caso
 * contrario
 */
bool isCollisionable() const;

/**
 * Registra un afectador para que este aplique una fuerza sobre
 * la misma, variando entonces la aceleración de la partícula.
 * @param affector Afectador a registrar.
 * @see affectors::ForceAffector
 */
void registerAffector(affectors::ForceAffector *affector);

/**
 * Chequea si una partícula se superpone con otra.
 * @param other Partícula contra la cual se desea
 * chequear la colisión.
 * @return true si las partículas están colisionando, false en
 * caso contrario.
 */
bool collidesWith(const Particle& other);
```



Spring

Esta clase encapsula a un resorte dentro del sistema de masas y resortes. Vincula dos partículas, y registra el afectador correspondiente en ambas.

```
/**
 * Crea una nueva instancia de Spring
 * @param first Primer partícula vinculada
 * @param second Segunda partícula vinculada
 * @param k Coeficiente de elasticidad
 */
Spring(Particle &first, Particle &second, double k);

/**
 * Obtiene la primer partícula vinculada
 * @return Primer partícula vinculada
 */
const Particle& getFirst() const;

/**
 * Obtiene la segunda partícula vinculada
 * @return Segunda partícula vinculada
 */
const Particle& getSecond() const;
```

PhysicsEngine

Esta clase es la encargada de la simulación física de un sistema de partículas bajo el efecto de fuerzas. Posee funcionalidad para registrar partículas para su simulación, y también métodos de manejo de afectores de fuerzas globales (como puede ser la gravedad y viscosidad). Su funcionalidad principal es la de realizar la integración numérica del sistema, actualizando la posición y la velocidad de cada partícula registrada. Para la integración aplica internamente un método modificado de Runge-Kutta de cuarto orden, aunque la estrategia implementada no genera dependencias con ninguna otra sección del código. De esta forma, se puede alterar el método de integración sin alterar el código que utiliza los servicios de simulación. Además, se logró desacoplar el método de simulación del manejo de las partículas y de las fuerzas aplicadas a ella. Son estos dos factores los que nos llevaron a reimplementar los servicios de simulación propuestos por la cátedra.

```
/**
 * Crea una nueva instancia de PhysicsEngine
 */
PhysicsEngine();

/**
 * Registra una partícula para ser manejada por el engine
 * @param particle Partícula a registrar
 */
void registerParticle(Particle *particle);

/**
 * Registra un afectador a aplicar sobre todas las partículas
 * manejadas.
 * @param affector Afectador a registrar
 */
void addSystemWideAffector(affectors::ForceAffector *affector);

/**
 * Actualiza las posiciones y velocidades de todas las partículas
 * manejadas en función de sus aceleraciones
 */
void doTimeStep();
```



GAME::CORE::AFFECTORS

ForceAffector

Esta es una clase abstracta pura que funciona como interfaz para aquellos objetos que aplican una fuerza sobre una partícula. Presenta un sólo método que se reimplementa en los distintos afectores concretos, como *GravityForceAffector*, *DampingForceAffector*, *DecayingForceAffector* y otros tantos más.

```
/**
 * Obtiene la fuerza que afecta a una partícula específica
 * @param particle Partícula a la que se está afectando
 */
virtual Vector2d GetAffectingForce(const Particle& particle) = 0;
```

GAME::DESIGN

ElementContainer

Como su nombre lo indica, esta clase representa un contenedor de elementos. Expone los distintos elementos utilizando tanto interfases genéricas *Element* como los tipos propios del elemento en si que se está inspeccionando. Posee además métodos para buscar elementos de acuerdo a posiciones geométricas, y maneja el *lifecycle* de los elementos agregados en él. Por último, expone de manera unificada los cambios realizados sobre los elementos contenidos a través de un sistema de señales homogéneo.

ElementFactory

Esta clase es un *factory* abstracto de elementos. Existe para encapsular la creación de los elementos, que obedece lógicas dispares en el cliente y en el editor de niveles.

GAME::DESIGN::ELEMENTS

Connection

El objeto *connection* es el responsable de la interconexión entre dos puntos de conexión contenidos por un *Element*. Hereda de *Element*, dado que reutiliza la mayoría de los servicios expuestos por este (como el manejo automático de memoria, la exposición a otros elementos, la posibilidad de ser movido y / o borrado, etc). Dado que las conexiones entre dos elementos se hacen compartiendo una partícula entre ambos, es el objeto *connection* quien maneja y expone una referencia a esta partícula de conexión.

```
/**
 * Crea una nueva instancia de Connection. Registra
 * la instancia creada en los dos ConnectionPoint
 * especificados
 * @param first Primer ConnectionPoint conectado
 * @param second Segundo ConnectionPoint conectado
 * @return
 */
Connection(connect::ConnectionPoint &first,
           connect::ConnectionPoint &second);

/**
 * Obtiene la partícula compartida entre
 * los elementos conectados, conocida como
 * la partícula de conexión
 */
core::Particle *getConnectionParticle();
```



Element

Esta es una de las clases principales del engine. Representa un elemento abstracto del que heredan directa o indirectamente todas las herramientas del juego, como son MetalBar, MetalPlatform, Rope, Wheel, etc. Contiene la funcionalidad común a todos los elementos, como la posibilidad de registrarse en un engine físico para someterse a una simulación, la exposición de la lista de partículas, resortes, puntos de conexión y puntos de control de movimiento que componen al elemento y la infraestructura de observación de cambios a través de señales.



```
/**
 * Construye este Widget en el engine. Esto registra
 * en el engine todas las particulas que componen al
 * Widget. No pueden realizarse modificaciones sobre
 * el Widget una vez construido.
 * @param engine Engine fisico sobre el que se contruye
 * el Widget
 */
void build(core::PhysicsEngine &engine);

/**
 * Limpia las particulas y resortes del elemento. Este
 * metodo debe llamarse siempre antes de reconstruir
 * al elemento en un engine.
 */
void scrap();

/**
 * Devuelve una lista con todas las particulas que
 * componene el widget. Solamente puede llamarse a este
 * metodo luego de haber construido el widget a traves de
 * la funcion build
 * @return Particulas que componen al widget
 * @see Widget#build
 */
const particle_list &getParticles() const;

/**
 * Devuelve una lista con todos los resortes que
 * componen el widget. Solamente puede llamarse
 * a este metodo luego de haber construido el widget
 * @return Resortes que componen al widget
 * @see Widget#build
 */
const spring_list &getSprings() const;

/**
 * Devuelve una lista con todos los puntos de movimiento
 * que posee el widget. Un punto de movimiento permite
 * mover alguna de las posiciones que definen al elemento.
 * Los puntos de movimiento se agregan a la lista de puntos
 * de movimiento protegida durante la construccion del elemento,
 * solamente si el elemento es editable
 */
const mpoint_list &getMovePoints() const;

/**
 * Devuelve una lista con todos los puntos de conexion que
 * posee el widget. Un punto de conexion permite conectar
 * dos widgets entre si, de manera que el movimiento
 * de cada uno de ellos esté restringido por el otro.
 * Los puntos de conexion se agregan a la lista de
 * puntos de coinexion protegida durante la construccion del
 * elemento
 */
const cpoint_list &getConnectionPoints() const;

/**
 * Elimina el elemento.
 */
virtual void erase();

/**
 * Devuelve true si el punto especificado esta contenido
 * dentro del area que ocupa el elemento
 */
virtual bool containsPoint(const core::Vector2d &vector) const = 0;

/**
 * Señal lanzada cuando el elemento es modificado
 */
modified_signal_type &modified();

/**
 * Señal lanzada cuando el elemento es eliminado
 */
erased_signal_type &erased();
```



GAME::DESIGN::CONNECT

Connect

Esta es la clase abstracta de la que heredan todos los puntos de conexión. Un punto de conexión es un objeto encargado de proveer una interfaz consistente para la interconexión entre elementos distintos. Cada elemento expone una colección de puntos de conexión; las conexiones entre dos elementos se realizan creando un objeto de tipo *Connection* utilizando dos puntos de conexión cualquiera. El punto de conexión tiene entonces una doble responsabilidad. Por un lado, sirve a este objeto *Connection* como interfaz genérica de acceso al *Element* que lo contiene. Por el otro, sirve al *Element* como interfaz de acceso a los objetos de conexión. Existen diversas subclases de esta clase, cada una especializada para el tipo de *Element* en la que son contenidas.

```
/**
 * Obtiene la posicion en donde esta ubicado el punto
 * de conexion
 */
virtual core::Vector2d getPosition() const = 0;

/**
 * Obtiene el radio del area ocupada por el punto
 * de conexion
 */
double getRadius() const;

/**
 * Obtiene el elemento al que pertenece este
 * punto de conexion
 */
virtual elements::Element &getParentElement() = 0;

/**
 * Añade una conexion al punto de conexion
 * @param connection Conexion a añadir
 */
void addConnection(elements::Connection *connection);

/**
 * Devuelve true si alguna de las conexiones
 * añadidas al punto de conexion continua viva
 */
bool isConnected() const;

/**
 * Devuelve una lista con todas las particulas de
 * conexion de las conexiones asociadas a este
 * punto de conexion.
 * @return
 */
particle_list getConnectionParticles();
```

GAME::DESIGN::MOVE

MovePoint

Así como existe una clase *ConnectionPoint* que maneja todo lo relacionado con la lógica de interconexión entre objetos, proveyendo así una interfaz única y consistente para el acceso a la funcionalidad de conexión, la clase *MovePoint* expone la misma funcionalidad para el concepto de movimiento de elementos. Existen diversas especializaciones de esta clase que realizan el movimiento de zonas específicas de elementos de tipo específico; todas estas variantes están abstraídas en esta clase base.



```
/**
 * Obtiene la posicion donde esta ubicado el
 * punto de movimiento
 */
virtual core::Vector2d getPosition() const = 0;

/**
 * Obtiene el radio del area ocupada por el
 * punto de movimiento
 */
double getRadius() const;

/**
 * Mueve la posicion asociada a este punto de
 * movimiento a una nueva posicion
 * @param newPosition Nueva posicion
 */
virtual void move(const core::Vector2d &newPosition) = 0;
```

GAME::LOGIC

DesignToolbox

Esta clase es una implementación de *ElementFactory* que incluye la lógica de creación de elementos propia del juego. Incluye métodos para manejar la disponibilidad de los elementos, y también administra los precios y el gasto en la adquisición de elementos.

GameEngine

Clase principal del manejo del juego. Proporciona por un lado una fachada para el acceso a la funcionalidad de manejo de elementos y simulación física, al heredar de *ElementContainer*. Pero además incluye métodos de manejo de los eventos propios del juego, como resolución de un nivel, timeouts, cálculo de puntaje, etc.

```
/**
 * Realiza un paso de tiempo en la simulacion
 * fisica del juego. Solo se puede llamar
 * estando en modo de ejecucion.
 */
void doTimeStep();

/**
 * Cambia el modo actual a modo de diseño
 */
void swapToDesignMode();

/**
 * Cambia el modo actual a modo de ejecucion
 */
void swapToRuntimeMode();

/**
 * Devuelve verdadero si se supero
 * el tiempo maximo para pasar el nivel
 * y el mismo no ha sido resuelto
 */
bool isLevelFailed() const;

/**
 * Devuelve verdadero se se resolvio
 * el nivel
 */
bool isLevelCompleted() const;

/**
 * Calcula el puntaje obtenido
 * por la resolucion del nivel
 */
int calculateScore();
```




GAME::SERIALIZATION

Este módulo contiene a todas las entidades que representan a los conceptos de la aplicación.

GAME::SERIALIZATION::SERVICE

Este módulo contiene a todos los servicios de la aplicación. Un servicio se define como toda acción o evento que pueda recibir una entidad del modelo pero que no pueda incluirse dentro de la clase de la entidad por cuestiones de bajo acoplamiento y alta cohesión. Por ejemplo, un método que retorne una posición de la pelota iría en la clase de la pelota ya que no involucra conceptos externos, en cambio el método que persiste a la pelota en un xml no puede ir en la clase pelota ya que en este caso se requiere insertar código externo al modelo, como ser el manejo de un archivo xml. Si este código se insertara en la misma clase pelota esta clase cumpliría dos funciones, representar a una pelota y persistir a una pelota, con lo cual hay baja cohesión. Además se requeriría acoplar código dependiente de la librería xml a usar, con lo cual hay alto acoplamiento. Para solucionar esto el método para persistir una pelota debe estar en una clase aparte a la que llamaremos service.

StageService

Es una interfaz que posee todos los servicios que afectan a la entidad Stage y a las entidades que componen a Stage. Es el único servicio necesario ya que Stage es la única *aggregate root*.

StageServiceImpl

Es la implementación de StageService. Esta clase utilizará a las clases provistas por el módulo de persistencia a medida que las necesite.

ServiceLocator

Esta clase juega el papel de Facade del módulo service y al mismo tiempo es un Factory de servicios. El business delegate nunca instanciará un service directamente, sino que lo obtendrá a partir del ServiceLocator.

GAME::SERIALIZATION::PERSISTENCE

Este módulo contendrá a todas las clases involucradas en la persistencia del modelo.

StageXmlRepository

Es una interfaz que representa a un repositorio de escenarios, o sea que provee los metodos necesarios para persistir y recuperar un escenario en un archivo XML.

StageXmlRepositoryTinyXml

Es la implementación del StageXmlRepository utilizando a la librería TinyXml.

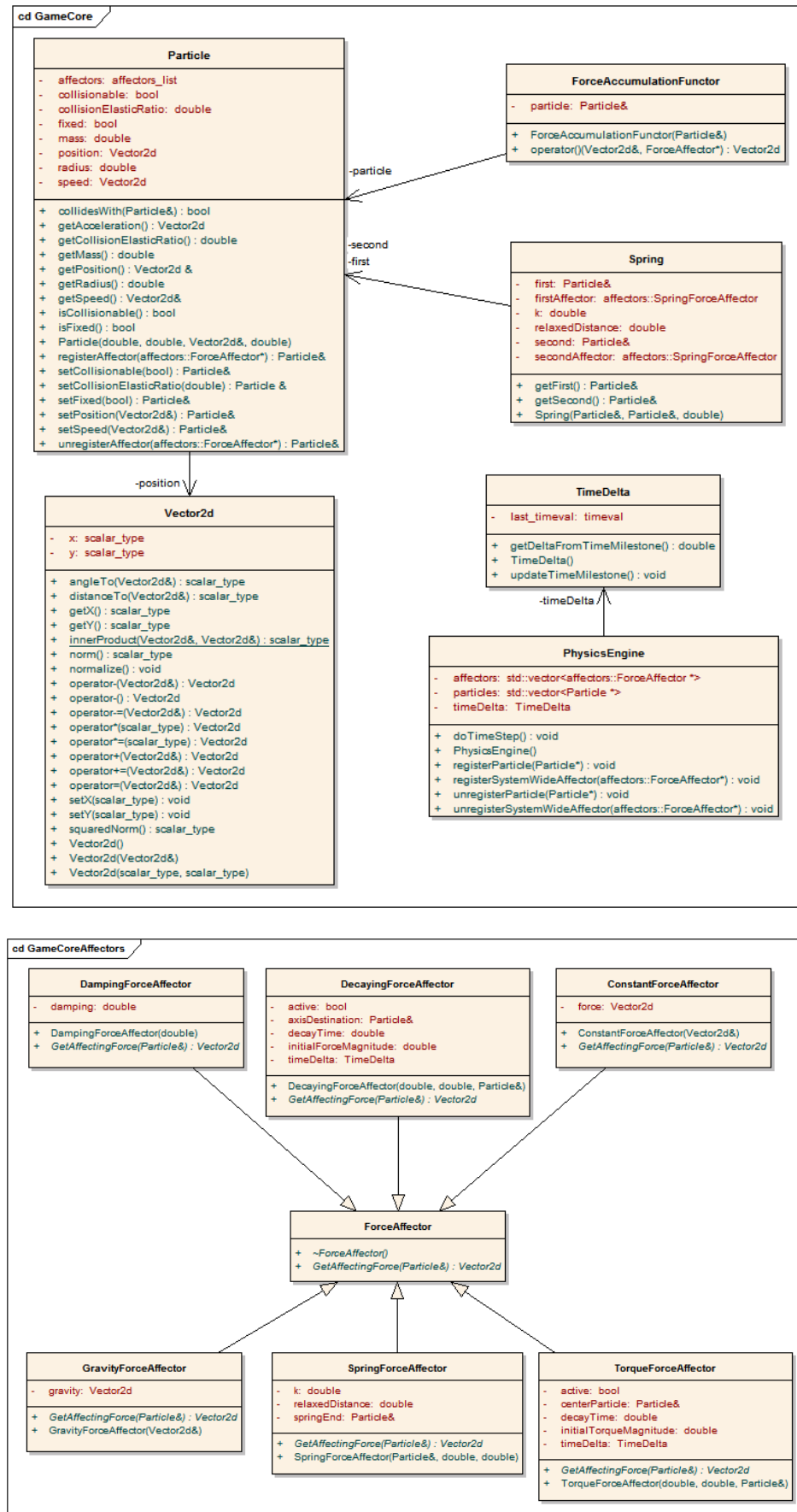
RepositoryFactory

Esta clase juega el papel de Facade del módulo y a su vez de factory proveyendo repositorios a los services a traves de interfaces, independizando así a los servicios de la librería *framework* utilizada.



Diagramas UML

CORE

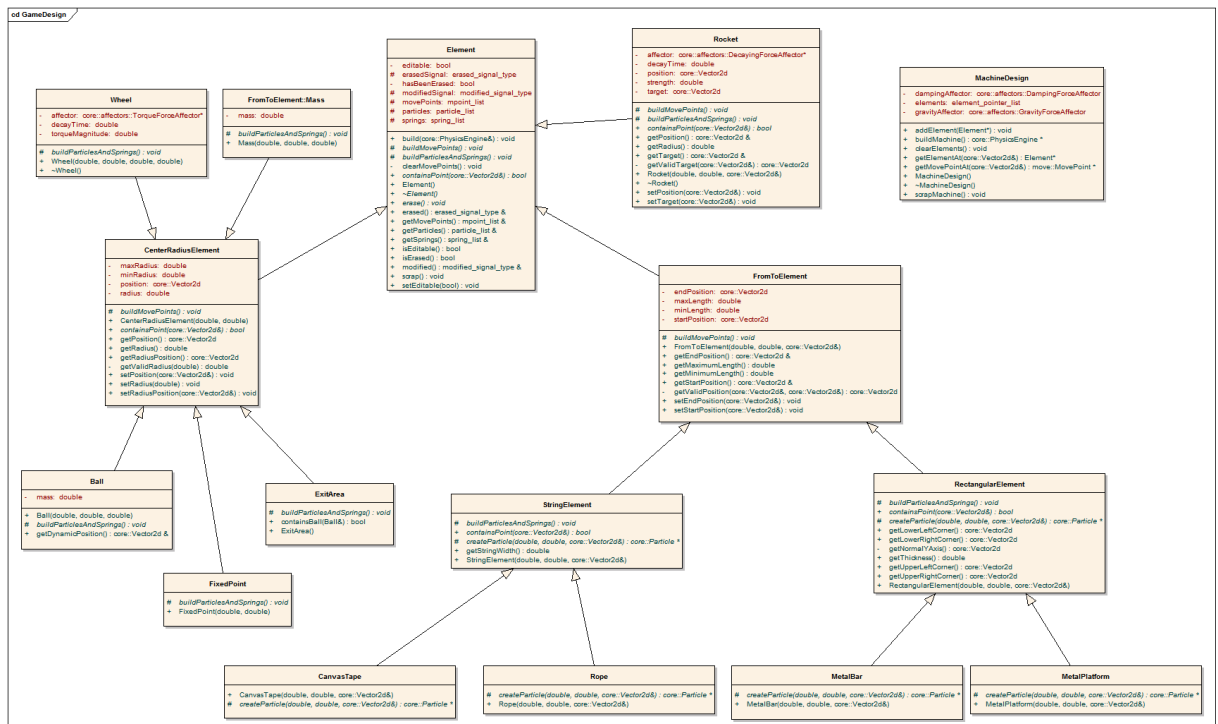




LOGIC



DESIGN





cd GameDesign

ElementContainer

```
- ball: elements::Ball*
- canvasTapeCollection: CanvasTapeCollection
- connectionCollection: ConnectionCollection
- designModifiedSignal: design_modified_signal_type
- elementCollection: std::vector<elements::Element*>
- exitAreaCollection: ExitAreaCollection
- fixedPointCollection: FixedPointCollection
- massCollection: MassCollection
- metalBarCollection: MetalBarCollection
- metalPlatformCollection: MetalPlatformCollection
- obstacleCollection: ObstacleCollection
- rocketCollection: RocketCollection
- ropeCollection: RopeCollection
- wheelCollection: WheelCollection

+ addCanvasTape(elements::CanvasTape*) : void
+ addConnection(elements::Connection*) : void
+ addExitArea(elements::ExitArea*) : void
+ addFixedPoint(elements::FixedPoint*) : void
+ addGenericElement(elements::Element*) : void
+ addMass(elements::Mass*) : void
+ addMetalBar(elements::MetalBar*) : void
+ addMetalPlatform(elements::MetalPlatform*) : void
+ addObstacle(elements::Obstacle*) : void
+ addRocket(elements::Rocket*) : void
+ addRope(elements::Rope*) : void
+ addWheel(elements::Wheel*) : void
+ clearElements() : void
+ designModified() : design_modified_signal_type &
+ ElementContainer()
+ ~ElementContainer()
+ getBall() : elements::Ball*
+ getCanvasTapeCollection() : CanvasTapeCollection&
+ getConnectionCollection() : ConnectionCollection&
+ getConnectionPointAt(core::Vector2d&) : connect::ConnectionPoint *
+ getElementAt(core::Vector2d&) : elements::Element *
+ getElementCollection() : ElementCollection&
+ getElementConstCollection() : ElementCollection&
+ getExitAreaCollection() : ExitAreaCollection&
+ getFixedPointCollection() : FixedPointCollection&
+ getMassCollection() : MassCollection&
+ getMetalBarCollection() : MetalBarCollection&
+ getMetalPlatformCollection() : MetalPlatformCollection&
+ getMovePointAt(core::Vector2d&) : move::MovePoint *
+ getObstacleCollection() : ObstacleCollection&
+ getRocketCollection() : RocketCollection&
+ getRopeCollection() : RopeCollection&
+ getWheelCollection() : WheelCollection&
- launchDesignModifiedSignal() : void
+ setBall(elements::Ball*) : void
```

ElementFactory

```
+ createBall() : elements::Ball *
+ createCanvasTape(core::Vector2d&) : elements::CanvasTape *
+ createExitArea() : elements::ExitArea *
+ createFixedPoint() : elements::FixedPoint *
+ createMass() : elements::Mass *
+ createMetalBar(core::Vector2d&) : elements::MetalBar *
+ createMetalPlatform(core::Vector2d&) : elements::MetalPlatform *
+ createObstacle(core::Vector2d&) : elements::Obstacle *
+ createRocket(core::Vector2d&) : elements::Rocket *
+ createRope(core::Vector2d&) : elements::Rope *
+ createWheel() : elements::Wheel *
+ ~ElementFactory()
```



Biblioteca de Interfaz de Usuario

Descripción General

Esta biblioteca implementa los servicios de interacción con el usuario, reutilizados tanto en el cliente del juego como en el editor de niveles. Estos servicios se dividen en dos grandes tipos. Por un lado, los servicios relacionados con la presentación gráfica de los conceptos del juego, contenidos dentro del espacio de nombres *views*. Por otro lado, esta biblioteca contiene las clases que definen la infraestructura necesaria para responder al input del usuario. Estas clases están agrupadas dentro del espacio de nombres *controllers*.

Clases

A continuación se detallarán las clases y métodos principales de este módulo por espacio de nombres (*namespace*).

UI::CONTROLLERS

ConfigurableRenderAreaController

Este es un controlador genérico preparado para manejar la interacción del usuario con un *GameRenderArea*. Captura los eventos disparados por este y los delega a un *strategy* interno configurable. Esto permite desacoplar al *GameRenderArea* de las diversas operaciones de creación, movimiento y eliminación de elementos

```
/**
 * Crea una nueva instancia de ConfigurableRenderAreaController
 * @param renderAreaWidget Render area cuyo input de usuario
 * se estará controlando
 */
ConfigurableRenderAreaController(
    ui::views::GameRenderArea &renderAreaWidget);

/**
 * Establece un nuevo strategy de manejo de input de usuario.
 * El lifecycle del handler configurado es manejado por
 * el controlador, por lo que no es necesario guardar una
 * referencia a el.
 * @param handler Strategy a configurar
 */
void configureCurrentHandler(RenderAreaInputHandler *handler);
```

RenderAreaInputHandler

Strategy de manejo del input del usuario a través de un *GameRenderArea*. Esta es una clase abstracta implementada por los diversos controladores específicos de creación, movimiento y eliminación de elementos. Es configurada dentro de un *ConfigurableRenderAreaController*, encargado de redireccionar los eventos específicos del *GameRenderArea* a las llamadas a las funciones que esta clase abstracta define.



```
/**
 * Ocurre cuando se presiona el botón izquierdo del
 * mouse sobre el area de diseño
 * @param x Posicion x del mouse cuando se hizo
 * click
 * @param y Posicion y del mouse cuando se hizo click
 */
virtual void handleMousePressed(int x, int y) = 0;

/**
 * Ocurre cuando se suelta el botón izquierdo del
 * mouse sobre el area de diseño
 * @param x Posicion x del mouse cuando se solto
 * el click
 * @param y Posicion y del mouse cuando se solto
 * el click
 */
virtual void handleMouseRelease(int x, int y) = 0;

/**
 * Ocurre cuando se mueve el mouse sobre el
 * area de diseño
 * @param x Posicion x del mouse a la que se movio
 * @param y Posicion y del mouse a la que se movio
 */
virtual void handleMouseMove(int x, int y) = 0;
```

UI::VIEWS

Renderer

Esta es una interfaz utilizada para describir un objeto que puede ser usado por el *GameRenderArea* para dibujar contenido en su interior. Es implementada por diversas clases que se encargan de dibujar aspectos concretos del modelo de diseño de la máquina, como por ejemplo *ConnectionPointsRenderer*, que dibuja todos los connection points de los elementos, *ElementContainerDesignRenderer*, que dibuja la vista de diseño de cada elemento, o *BackgroundRenderer*, que dibuja la imagen del fondo del nivel.

```
virtual void render(Cairo::RefPtr<Cairo::Context> drawingContext)
const = 0;
```

GameRenderArea

Este es una especialización de un Widget GTK orientado al dibujo customizado. Posee funcionalidad de dibujo en capas que pueden ser activadas individualmente. Esto se logra delegando el repintado de la superficie del control a una lista de objetos que implementan la interfaz *Renderer*. Estos objetos dibujan sobre la superficie que expone el *GameRenderArea*, desacoplando así el que se dibuja del control donde se dibuja. Además, esta clase expone una serie de eventos de manejo de la interacción con el usuario de alto nivel, permitiendo al controlador apropiado una conexión a una serie de señales para manejar el input del usuario.



```
enum RenderLayers {
    BackgroundLayer = 0,
    ElementDesignLayer,
    ElementRuntimeLayer,
    MovePointLayer,
    ConnectPointLayer,
    OverlayLayer
};

/**
 * Crea una nueva instancia de GameRenderArea
 * @param scale Parametro de escalamiento para
 * las coordenadas de dibujo
 */
GameRenderArea(double scale = 1.0);

/**
 * Añade un renderer a un layer especificado.
 * El lifecycle del renderer agregado pasa a ser
 * responsabilidad del GameRenderArea.
 * @param layer Grupo al que pertenecera el renderer
 * @param renderer Renderer a agregar
 */
void addRendererToLayer(RenderLayers layer, Renderer *renderer);

/**
 * Habilita un grupo de renderers para ser dibujados.
 * Estos seran dibujados la proxima vez que se llame
 * al metodo GameRenderArea#redraw.
 * @param layer Grupo de renderers a habilitar.
 */
void enableLayer(RenderLayers layer);

/**
 * Deshabilita un grupo de renderers para ser dibujados.
 * Estos no seran dibujados la proxima vez que se llame
 * al metodo GameRenderArea#redraw.
 * @param layer Grupo de renderers a habilitar.
 */
void disableLayer(RenderLayers layer);

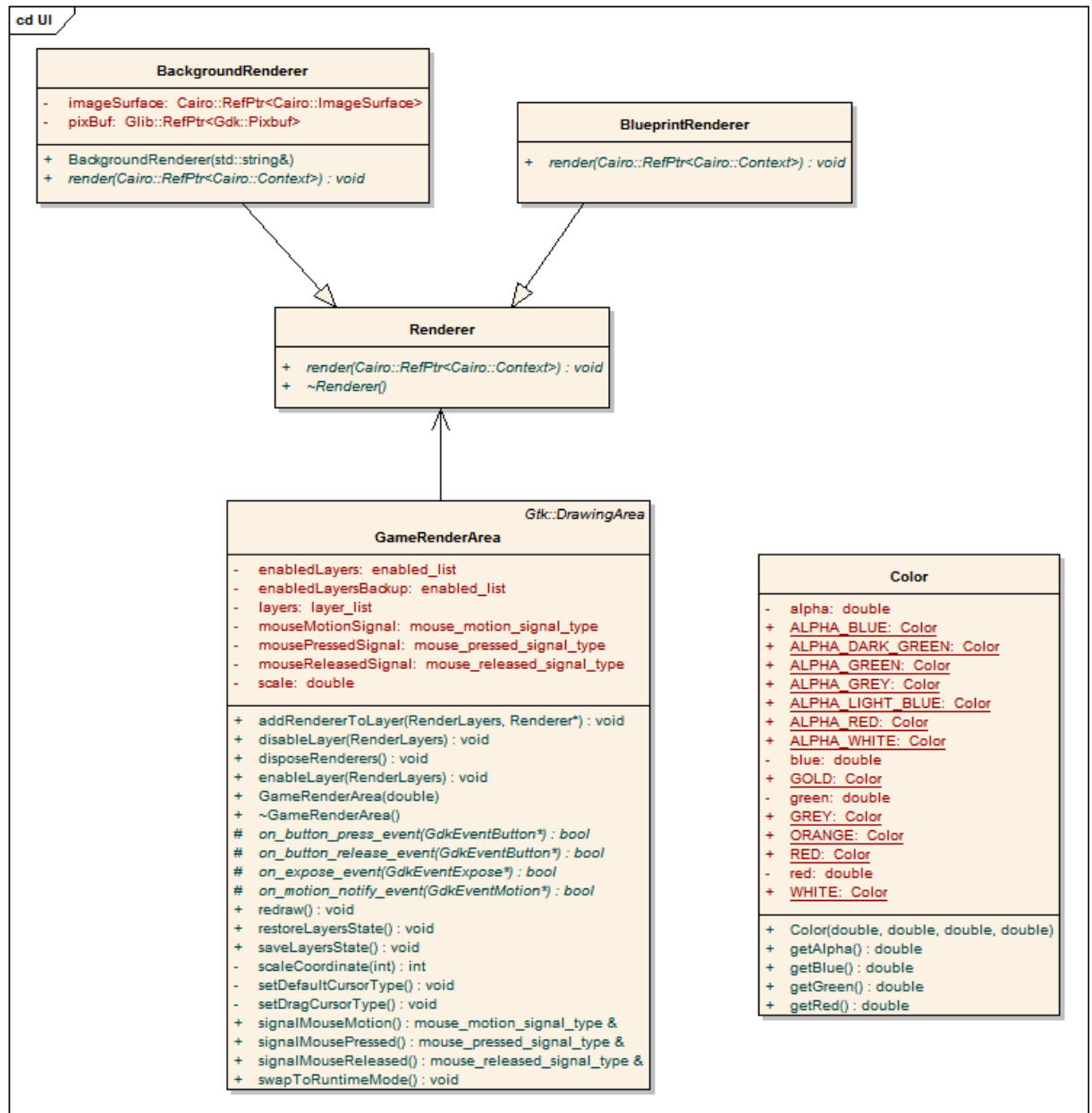
/**
 * Almacena una copia del estado de las habilitaciones
 * de los distintos grupos de renderers para poder ser
 * restaurada posteriormente
 */
void saveLayersState();

/**
 * Restaura el estado de las habilitaciones de los
 * distintos grupos de renderers grabados anteriormente
 * a traves de GameRenderArea#saveLayersState
 */
void restoreLayersState();

/**
 * Invalida el area del GameRenderArea, lo que schedulea
 * en la cola de eventos de GTK una peticion de redibujado
 */
void redraw();
```




Diagramas UML





Programas Intermedios y de Prueba

Tal como se explicó anteriormente, para las pruebas no se utilizó ningún *framework* en particular sino que se han creado pequeños programas de prueba. Estos programas de prueba se utilizaron sólo al principio, es decir cuando la infraestructura del presente trabajo no era suficiente como para poder efectuar las pruebas directamente sobre el juego mismo. A medida que se fue avanzando y, tanto la integración como la infraestructura estaban más maduras, se fueron dejando de lado estos programas de prueba y se procedió directamente a utilizar la misma aplicación para evaluar aquellos segmentos de código o módulos que se iban agregando.

PRUEBAS DE CONECTIVIDAD Y PROTOCOLO

Para garantizar el correcto funcionamiento del protocolo y del envío de comandos y archivos (tanto XML como imágenes) se efectuaron pequeñas pruebas de concepto. La idea fue separar del complejo ámbito del presente trabajo aquellas funcionalidades relacionados con el envío y recepción de datos a fines de efectuar sencillas pruebas que permitan indicar si el código escrito era correcto.

PRUEBAS GRÁFICAS

Muchas veces nos encontramos con que los resultados visuales de la simulación no eran del todo agradables. Para ellos procedimos a crear un reducido entorno de pruebas gráficas en el que permitíamos agregar elementos a la pantalla y correr una simulación. Esto nos permitió mejorar la calidad gráfica del presente trabajo puesto que pudimos evaluar de manera rápida y precisa todos los componentes gráficos intervinientes.

PRUEBAS ALGORÍTMICAS

En algunos casos, hubo que codificar lógicas complejas o confusas. Para ello, nuestra idea fue crear pequeñísimos programas en los que se evaluara el algoritmo sobre los casos típicos y, una vez probado, mover el código resultante al código del presente trabajo.



Código Fuente

En este apéndice procederemos a incluir todo el código fuente de las tres aplicaciones principales (Servidor, Cliente y Editor de Niveles) como así también de las bibliotecas estáticas.