Taller de Programación I - 75.42

Trabajo práctico grupal: Z

Documentación Técnica

Profesores:

- Roca, Pablo Daniel
- Lafroce, Matías

Grupo: 3

Integrantes:

- Lavandeira, Lucas - 98042
- Llauro, Manuel - 95736
- Rozanec, Matías - 97404

Facultad de Ingeniería, Universidad de Buenos Aires

1° Cuatrimestre, 2017

Requerimientos de Software

Idóneamente el programa puede ser compilado en cualquier distribución de Linux, habiendo instalado previamente los siguientes paquetes:

- *Gtkmm-3.0-dev*

El paquete es fácilmente accesible en distribuciones de la familia Debian, a través de `apt`. El programa fue desarrollado y ejecutado únicamente en sistemas Ubuntu 16.04 o superior, por lo que sólo puede garantizarse el funcionamiento correcto bajo ese contexto.

Para compilar el programa, es necesario cmake, de versión `2.5` o mayor, y luego ejecutarlo pasando el archivo `CMakeLists.txt` como parámetro. Si se quiere ejecutar en modo *debug*, para poder acceder al *stack trace* en `gdb`, debe pasarse además el flag `-DCMAKE_BUILD_TYPE=Debug`. El compilador utilizado por `cmake` en los entornos utilizados para el desarrollo fue `gcc 5.4.0`.

El servidor acepta como parámetro la ruta a un archivo de configuración, con el formato adecuado, para poder ser ejecutado. Uno es provisto entre los archivos entregados, `cfg.xml`. Además, la lectura de mapas es llevada a cabo en un directorio de nombre `maps` ubicada a la misma altura que el ejecutable. Al menos un mapa debe existir en ese directorio. Son provistos dos mapas distintos con la entrega, con la posibilidad de generar más a partir de una aplicación.

El cliente requiere varios archivos gráficos para su funcionamiento. El cliente debe ser ejecutado con una carpeta `res` a la misma altura que el ejecutable, que contenga todos los recursos utilizados. La interfaz gráfica es construida a través de un archivo de *Glade*, denominado `Z.glade`, que también debe estar ubicado en el directorio raíz.

La depuración del programa durante su desarrollo fue hecha a través de las herramientas provistas por *CLion*, un entorno de desarrollo de C/C++ de la compañía JetBrains.

Descripción del programa

Z es un juego de estrategia 2D multijugador por red. Varios clientes se conectan a un único servidor, y se juegan partidas en donde un equipo resulta victorioso sobre los demás cuando se cumplan determinadas condiciones. Cada jugador cuenta con territorios propios, en donde puede emplear las fábricas que contiene para generar robots y vehículos capaces de atacar a las estructuras enemigas.

Un ciclo de juego consiste en varios jugadores (hasta cuatro) conectándose al servidor, eligiendo el mapa que deseen jugar de los disponibles e iniciando la partida. Luego, cada jugador cuenta con un fuerte, un conjunto de fábricas y de robots a su disposición, y su meta es destruir los recursos de los demás jugadores. Cada fábrica puede generar distintos tipos de unidades, cada uno con distintas ventajas y desventajas. Existen territorios neutrales en el mapa de juego que cada jugador puede capturar para aumentar su velocidad de producción de robots y pasar a su disposición las fábricas dentro de ellos, pero debe tener cuidado porque de la misma manera los territorios pueden ser robados por sus contrincantes. Cuando el fuerte de un jugador es destruido, se lo considera como perdedor. El último jugador en tener su fuerte en buen estado es considerado victorioso.

A partir del código fuente se generan tres programas: El servidor, el cliente, y una herramienta generadora de mapas de juego aleatorios.

- El generador de mapas cuenta con un par de clases que consisten en escribir archivos `XML` y generar números aleatorios, respectivamente. No cuenta con funcionalidad de redes o concurrencia.
- El servidor se separa en dos grandes módulos, uno de una unidad de control que procesa la lógica del juego de manera sincrónica (utilizando el concepto de *ticks* común en juegos de red), y otra que se dedica a la interacción con los clientes. Esto ocurre de manera concurrente, habiendo un hilo de ejecución por cliente, y uno adicional para la unidad de control.
- El cliente también se separa en dos módulos grandes, uno que maneja la interfaz gráfica, y otro para la comunicación con el servidor a través de *sockets*.

El proyecto entero está desarrollado en ISO C++11, respetando una estructura de el paradigma de orientación a objetos. Cada programa cuenta con varios módulos internos, y existe una librería de clases comunes tanto al cliente como al servidor, principalmente las

relacionadas a la comunicación de red (*sockets*) y de hilos y concurrencia. Se utilizaron dos librerías externas, *gtkmm* para la interfaz gráfica del cliente (única dependencia externa del proyecto) y *pugi*, una librería de lectura y manejo de archivos de formato XML usada principalmente para los mapas y archivos de configuración, que se implementaron respetando este formato.

La comunicación entre el servidor y los clientes es a través del protocolo TCP/IP. Para estructurar la comunicación se optó por seguir un protocolo de envío de comandos, que respetan un formato particular, de cada mensaje representando un comando, con su identificador como primer palabra, y seguido de varios argumentos, cada uno separado del siguiente por un carácter de guión medio (`'-'`). Tanto el servidor como el cliente aceptan y envían varios comandos distintos, que controlan el flujo entero del programa y de la partida siendo ejecutada.

El servidor fue pensado y diseñado para poder mantener varias partidas a la vez. Para ello se divide el estado de cada cliente conectado en tres etapas, una inicial de menú principal, luego cada cliente puede crear o unirse a partidas no empezadas (denominadas como *lobby*) y por último una vez que hayan al menos dos jugadores conectados y listos en un *lobby*, se puede comenzar la partida, en donde se levanta una unidad de control que maneja el flujo de juego específico. Cuando algún jugador haya obtenido la victoria, se termina la partida, la unidad de control se destruye, y los jugadores vuelven al menú principal. Así se puede jugar varias partidas entre clientes sin tener que reiniciar las aplicaciones ejecutadas.

Módulo Generador

El generador de mapas es un módulo con alcance reducido: se le pasan como argumentos varios parámetros sobre el mapa a ser generado, y el programa escribe como resultado un archivo de formato `XML` a el directorio de mapas, listo para ser utilizado en un juego. Cuenta con una simple clase `MapGenerator`, y una clase auxiliar `Random` que genera números aleatorios en sucesión.

El generador acepta cinco argumentos obligatorios, en orden:

- Tamaño del mapa: un entero que determina el máximo de cantidad de celdas a generar. El mapa resultado siempre es cuadrado, por lo que el tamaño especifica tanto la cantidad de filas como columnas de celdas. El resultado es un mapa de, a lo sumo, `NxN` celdas.

- Porcentaje de agua: valor de 0 a 100 de proporción de celdas de agua en el mapa. Se recomiendan valores bajos, hasta 15% para un mapa con una jugabilidad aceptable.
- Porcentaje de lava: Igual que el anterior, con celdas de lava inaccesibles por unidades. Se recomienda un valor inferior a las de agua.
- Cantidad de territorios a generar
- Nombre del mapa: el mapa será guardado en la carpeta “maps” bajo ese nombre otorgado.

Los algoritmos usados como implementación son simples intentos probabilísticos hasta obtener resultados positivos. Por ejemplo, para la generación de rocas sobre el mapa se itera sobre las celdas y a cada posición vacía se le otorga una chance reducida (2% por defecto) de contener una roca. De manera similar, la posición de las banderas o los fuertes en el mapa es hecha a través de generación de posiciones aleatorias en regiones reducidas del mapa hasta que salga una que no esté ocupada por agua o por lava.

El flujo de la ejecución entera es la instanciación de un objeto `MapGenerator` con los parámetros de la línea de comando, previamente validados. Luego se llama el único método público del objeto, `generate`, que genera aleatoriamente el mapa y lo guarda en la ruta especificada. Una vez finalizado esto, se termina la ejecución.

El programa tiene códigos de salida: 1 para un error en la validación de la entrada de parámetros, y 0 para una generación exitosa.

El constructor de `MapGenerator` instancia varios valores para ir haciendo un seguimiento. El resultado de la aplicación es un archivo XML con el formato visible en la próxima página. Los tipos de terreno cargados en la versión entregada son tres, `Tierra, Lava, Agua`. Una vez que se tiene el terreno base, se generan las estructuras: las rocas, los territorios, y las fábricas asociadas a ellos. De los territorios, cuatro de ellos deben tener un fuerte, uno por cada posible jugador en el mapa, que actúan como territorio inicial y la base que deben defender para asegurar su victoria. Los demás son de tipo bandera, capturables. Las posiciones de las estructuras son inteligentes, asegurándose de que las posiciones aleatorias dadas sean de tipo tierra.

La generación de los ríos de lava y agua se intentó hacer con memoria de la dirección del flujo, es decir que se generen ríos que aproximadamente siguen una dirección y un camino específicos. El algoritmo agarra una dirección de inicio y va “coloreando” celdas como de tipo agua/lava según corresponda, y cada iteración tiene una probabilidad de finalizar. Para asegurar la proporción de agua y lava especificada como parámetro lo que se decidió hacer es mantener una cuenta aproximada de cuantas celdas de lava y cuantas de

agua deben generarse en total en el mapa, y a medida que se hace la generación, ir descontando hasta llegar a cero. Para ir llevando el seguimiento de cuales celdas están ocupadas por líquidos, se optó guardar en memoria un vector doble de valores booleanos, que indican verdadero si la celda del índice x-y del vector es ocupada por un líquido. Este dato es particularmente útil para ubicar estructuras más adelante.

Las estructuras presentes en el mapa son tres: rocas, fábricas de robots y de vehículos. Se generan de manera diferente: las rocas son aleatoriamente ubicadas sobre terrenos vacíos con un porcentaje pequeño por celda. Las fábricas, se generan de a dos por territorio para mantener el juego parejo, y cada fábrica tiene un 50% de probabilidad de ser de alguno de los dos tipos en particular. Sobre los territorios, se eligen cuatro de manera aleatoria para albergar los fuertes de los jugadores. Los que no sean escogidos tienen en cambio una bandera en su posición central.

Los territorios son divididos de manera uniforme, es decir que ningún territorio sea más grande que otro. Esto presenta un problema: quizás el tamaño especificado por el usuario no es divisible de manera equitativa por la cantidad de territorios especificada. Lo que se decidió optar como la mejor solución fue declarar que el tamaño pasado no es el tamaño final del mapa, sino una cota superior del tamaño, y usar como tamaño real el valor más cercano al tamaño original que pueda ser dividido uniformemente.

```
<Map>
      <Terrain>
            <Row>
                  <Cell terrain=”...” />
                  …
            </Row>
            <Row>
                  …
            </Row>
            …
      </Terrain>
      <Structures>
            <Struct type=”Rock” x=”...” y=”...” />
            …
      </Structures>
      <Territories>
```

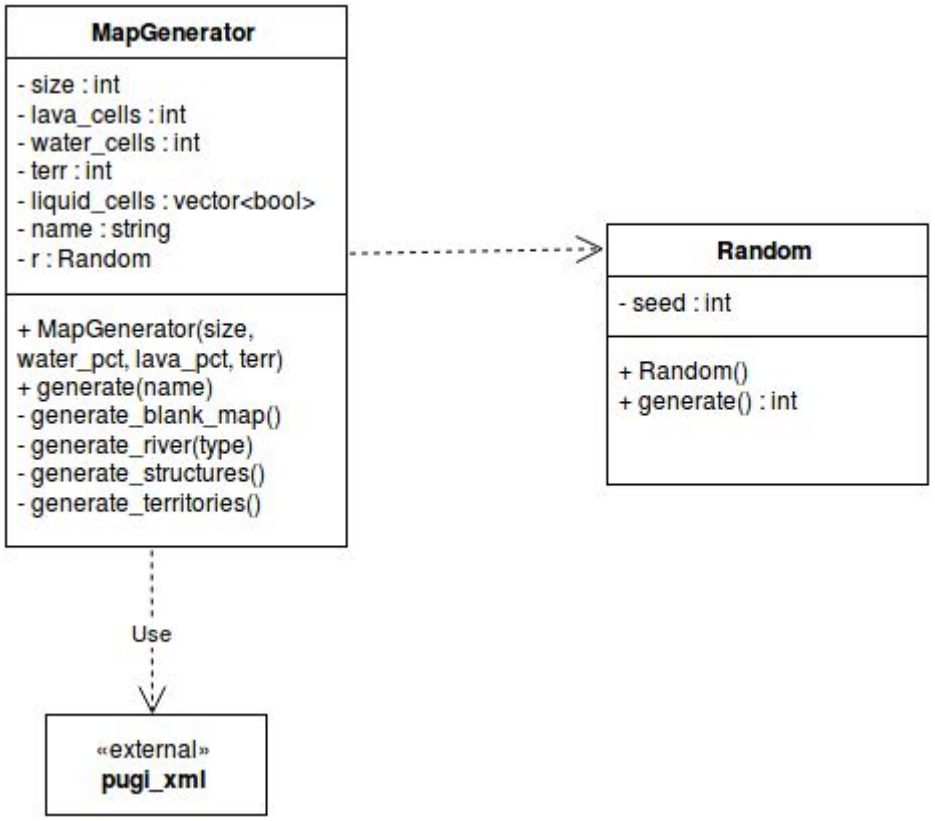
```
            <Fort x=”...” y=”...” />
                  <UnitFactory x=”...” y=”...” />
                  <VehicleFactory x=”...” y=”...” />
            </Fort>
            <Flag x=”...” y=”...” />
                  <UnitFactory .../>
                  ...
            </Flag>
      </Territories>
</Map>
```

Formato del archivo resultado del mapa, en XML

La generación de números aleatorios se hace a través de la función `rand_c` de la `stdlib` de C. Como la función acepta una semilla como parámetro, que guarda el estado entre llamadas, se extrajo esa funcionalidad a una clase aparte, que se encarga de proporcionar la semilla a la función cada vez que se necesite un número aleatorio.

Toda la lectura y escritura del archivo resultado es hecho a través de la librería de XML especificada anteriormente. La secuencia de acciones en la generación es la especificada: primero un mapa completamente vacío, luego ríos de agua, luego ríos de lava, luego rocas y por último los territorios con sus fábricas.

A continuación se muestra un simple diagrama de clases. Los métodos internos del generador son llamados en el orden secuencial especificado anteriormente.



Módulo Cliente

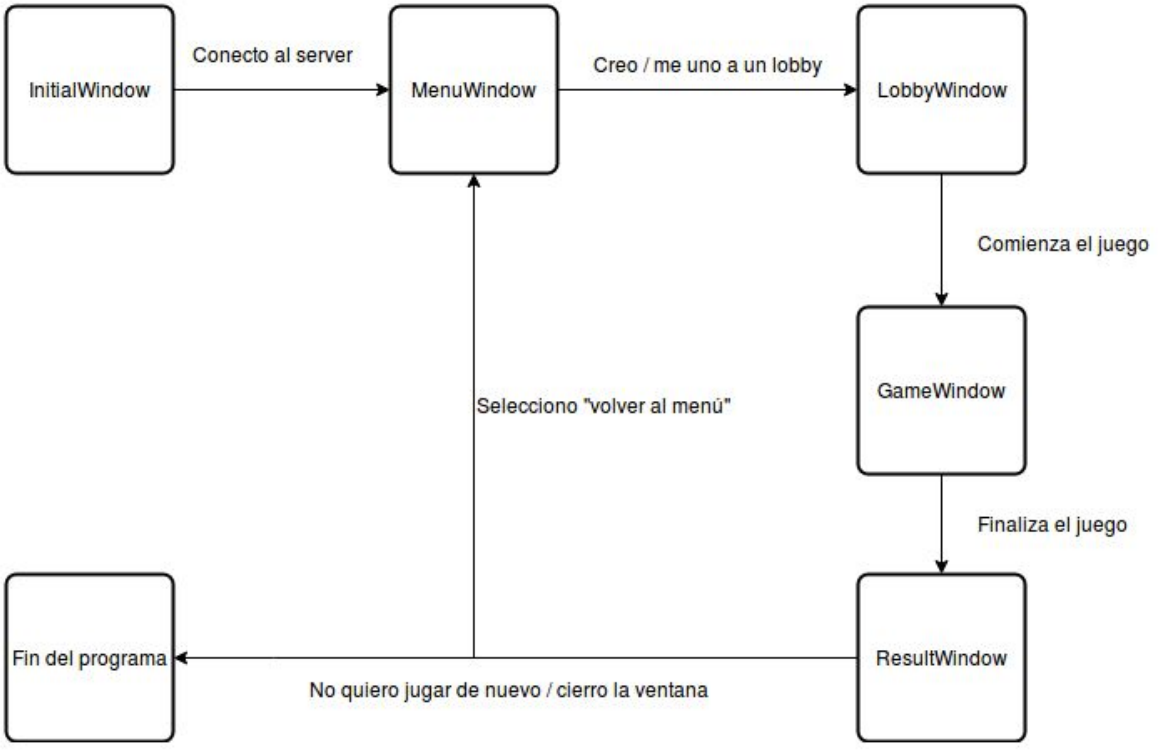
La aplicación del cliente se encarga de graficar y mostrar toda la información del juego, y controlar todas las unidades y ejecutar decisiones por parte de los jugadores. La aplicación no recibe argumentos por comando de línea, toda interacción y entrada de datos es hecho a través de la interfaz gráfica.

El cliente fue implementado utilizando las librerías *gtkmm* (junto con *cairomm,* y varias otras librerías de uso interno). La mayoría de la interacción del usuario con el programa es hecha a través del sistema de eventos de GTK, aprovechando la jerarquía de clases.

La interfaz gráfica entera es construida a través de *Glade*, un programa que permite construir interfaces gráficas fácilmente a través de una ventana interactiva, para poder confirmar de manera visual lo que se está desarrollando. *Glade* produce un archivo de formato XML legible por GTK para instanciar de manera práctica todos los objetos usados a lo largo del programa. El ciclo de ejecución del cliente del juego comienza con la lectura de este archivo. La lectura del archivo de Glade se realiza a través de una clase particular, que

fue llamada `GameBuilder`, que instancia todas las clases de los *widgets* de GTK usados a lo largo de la ejecución. Se utilizó la herencia disponible por *gtkmm* para instanciar las ventanas con sus *widgets* respectivos por separado de manera ordenada.

A continuación se muestra un diagrama de estados de la ejecución del cliente. El flujo sigue siempre el mismo orden, pasando de la ventana inicial donde se pide la dirección del servidor a conectarse, a un menú principal donde se elige la partida (*lobby*) al que se desea entrar. Luego, una vez que todos los jugadores de la partida estén marcados como listos para jugar, se inicia el juego donde se corre la ventana gráfica con el mapa, los edificios y las unidades visibles. Por último cuando finaliza se muestra una ventana de resultados, que indica si el jugador salió victorioso o perdió. Desde esa ventana se puede elegir volver al menú principal, para poder unirse a una nueva partida. Si en cualquiera de estos estados se cierra la ventana gráfica, el cliente termina su ejecución de manera ordenada, interrumpiendo todos los procesos activos (en particular los de comunicación con el servidor).



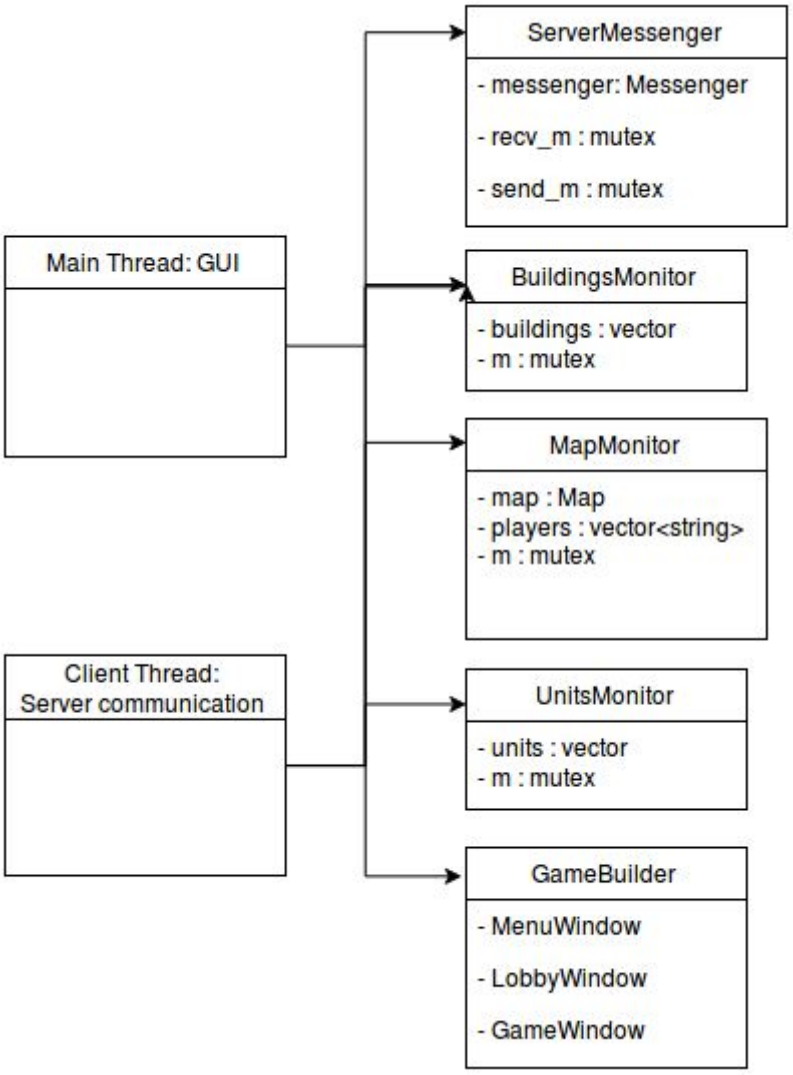
El programa inicialmente presenta una pequeña ventana de conexión con el servidor, en donde se pide una dirección y puerto al cual intentar conectarse, junto con su nombre. Luego cuando se logra establecer una conexión, se instancia un hilo aparte que se

encarga de recibir y procesar los comandos enviados desde el servidor. En el hilo principal se sigue manejando la interfaz gráfica.

Una vez establecida la conexión, el manejo del flujo anterior, potencialmente cíclico infinito si el jugador siempre escogiera volver al menú, se realiza en un ciclo *while*, instanciando una clase que maneja cada partida individual, denominada `Game`. Esta clase tiene como responsabilidad la implementación del diagrama de estados anterior, de ir controlando que el jugador a medida que escoge las opciones correctas, vaya mostrando la ventana siguiente. Al finalizar una partida, setea un valor booleano *play_again* a verdadero si el jugador escoge jugar de nuevo. Entonces en el ciclo *while* lee esa variable, y sólo si es falsa, finaliza el programa. De esta manera también se maneja la salida prematura: *play_again* es seteada a falsa.

Los recursos compartidos entre ambos hilos son protegidos a través de un monitor. También el hilo de comunicación tiene guardado referencias a las ventanas, y a través de métodos selectos puede modificar los valores mostrados en pantalla. Por como maneja GTK los eventos a procesar, ese proceso es *thread-safe*. Lo único que se hace es modificar valores, que luego GTK, en su manejo interno de eventos, lee para actualizar lo mostrado en pantalla. Se aseguró que la forma en que se decidió implementar esto no implique condiciones de carrera entre los hilos.

A continuación se muestra un diagrama con los monitores existentes en el cliente:

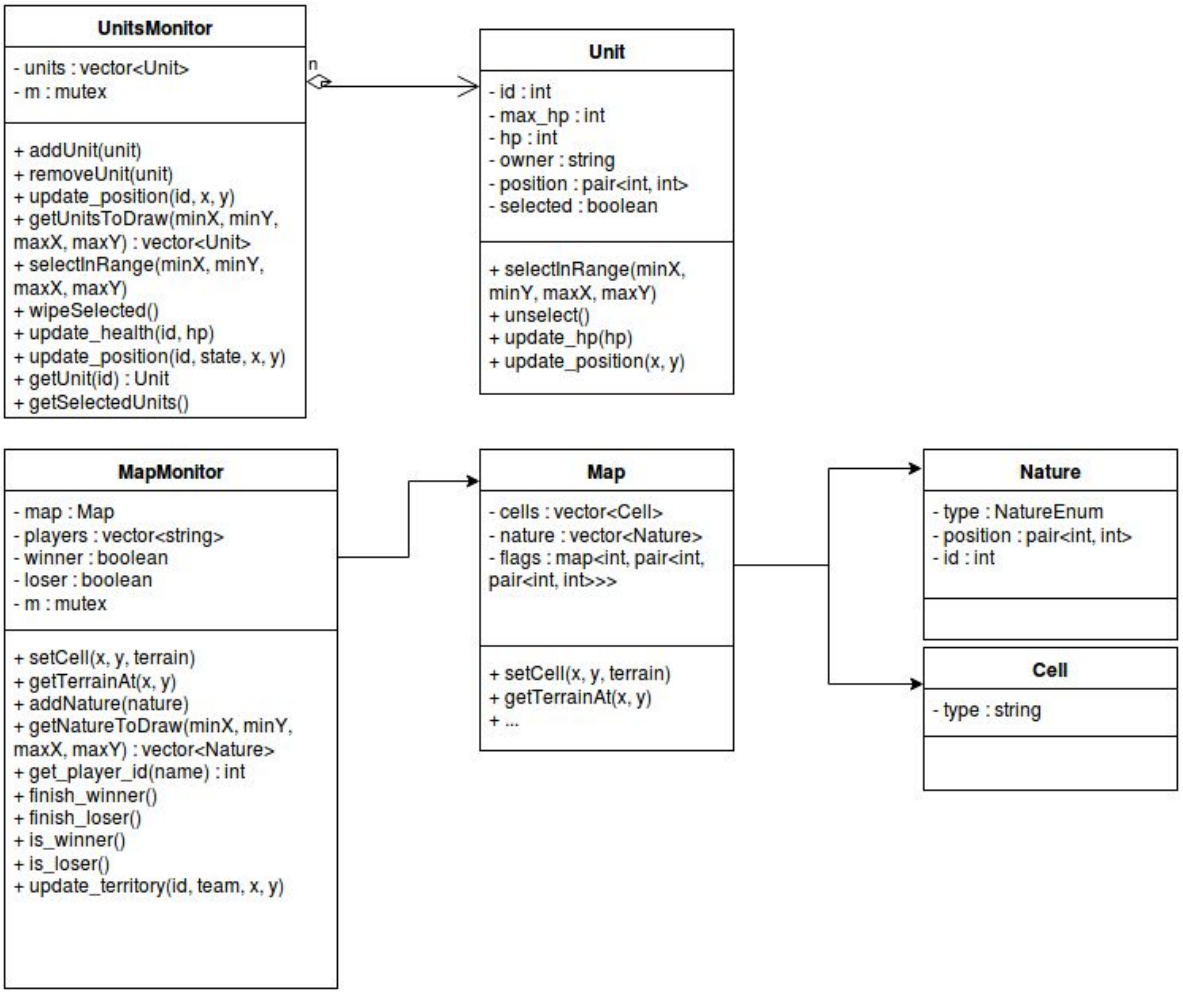


La mayoría de las variables son escritas sólo desde el hilo de comunicación, que hace de unidad lógica del sistema, y leídas desde ambos.

Todas las ventanas salvo la del juego principal operan de manera similar: se les pasa la referencia del monitor de mensajería con el servidor para poder mandar los mensajes, y cuando se hace un click en un botón, por ejemplo el de unirse a una partida, se envía el comando apropiado al servidor, y (en el hilo de comunicación) se recibe una validación como respuesta. De ser una respuesta positiva se hace la acción apropiada, por lo general el cambio de pantalla a la siguiente. En el menú principal y en el lobby, además, hay entrada de textos para elegir la sala a la que se desea unir y el mapa que se desea jugar, respectivamente. Todos los procesos de enviado, lectura de las entradas y validación ocurren cuando se clickea un botón: son los únicos eventos atados a estas ventanas.

En algunos momentos se hacen validaciones preventivas, es decir, chequeos previos al enviado de información al servidor, para no perder el tiempo mandando comandos que son obviamente inválidos. Por ejemplo, en el menú principal, se le pide al usuario ingresar en un campo de texto el ID del lobby a unirse. Este ID debe ser numérico, entonces si se hace click en el botón de unirse cuando la entrada no es un número, directamente se aborta el proceso antes de enviar el comando, porque ya se sabe que va a fallar.

Recursos protegidos



Por razones de brevedad, se omitió el monitor de edificios, BuildingMonitor, por ser muy similar al de unidades, sólo variando los campos particulares de la clase (los Building tienen campos tales como el tiempo de construcción, a diferencia de las unidades), y el

ServerMessenger, por ser una clase acotada, que solo protege los métodos del Messenger (ver el módulo de clases comunes para más detalles sobre esta última).

En particular cabe resaltar que, además de los campos típicos esperados como los puntos de salud o las coordenadas de su posición, las unidades (y los edificios también) tienen un campo de 'selected', que marca si fueron seleccionados por la última selección realizada, y que las unidades seleccionadas pueden ser devueltas a través del método `getSelectedUnits` del monitor. Otro punto que vale la pena mencionar es que las unidades se pueden copiar libremente, porque sus campos son todos valores simples que no es necesario mantenerse únicos.

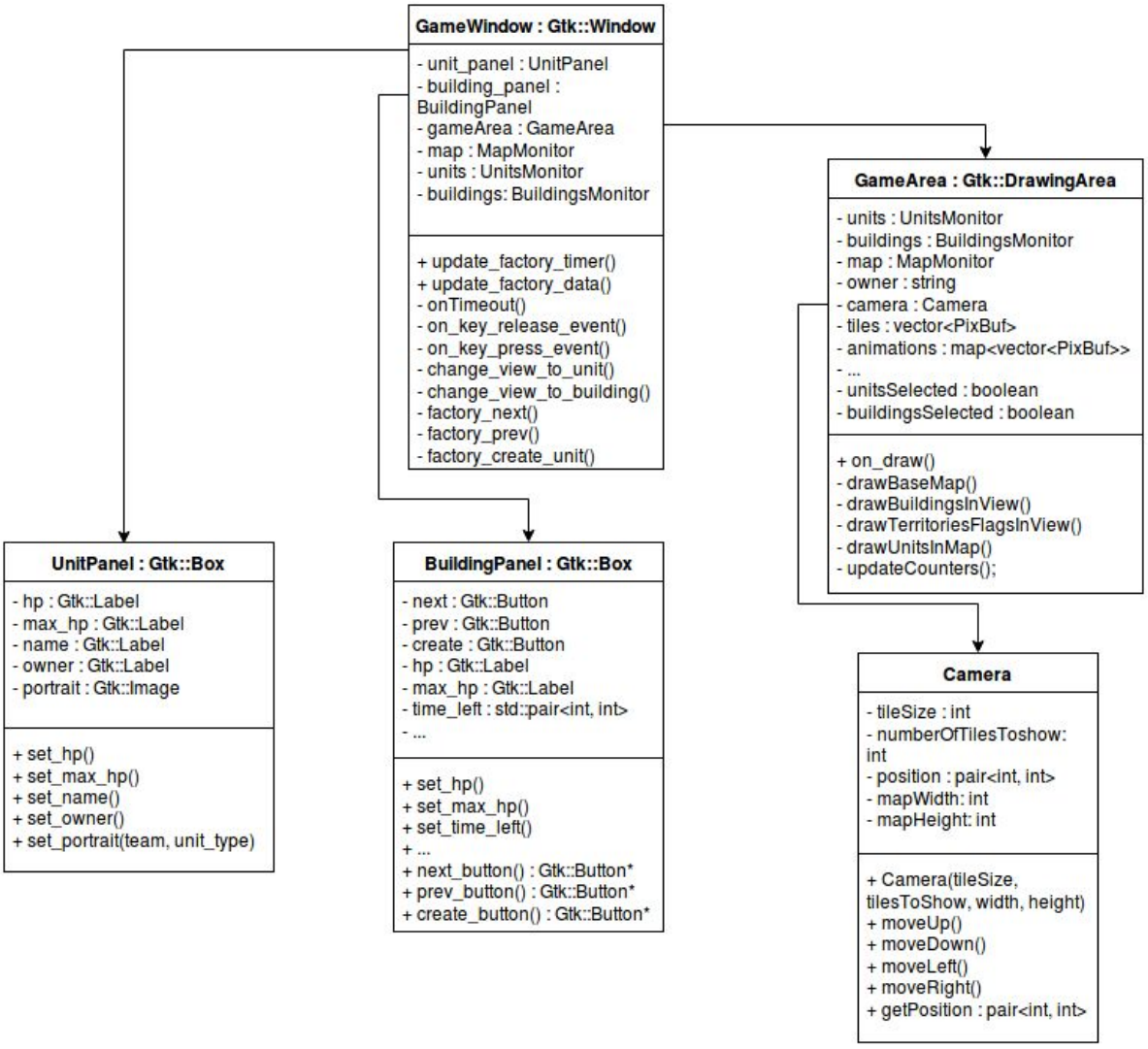
Ventana de juego principal

Una vez que comienza el juego se tiene una interfaz mucho más compleja que las anteriores, que consistían de un par de botones y algunos textos y entradas. La ventana de juego principal está modelada con una clase que hereda de `Gtk::Window`, denominada `GameWindow`. La ventana entera se subdivide en dos, en un área de juego en donde es visible el mapa, las unidades y los edificios, y un panel al costado con información de la selección actual. La ventana contiene referencias a los monitores anteriormente mencionados para poder leer el estado actual del juego arbitrariamente. Se le conecta un evento a la señal de *timeout* de GTK para que constantemente se redibuje el mapa entero con la información actualizada, y también mantenga actualizada los datos mostrados al costado. A continuación se muestra un diagrama de clases de los *widgets* de la ventana principal de juego. Los métodos públicos son llamados desde el hilo de comunicación con el servidor, para actualizar la información pertinente con el estado más reciente. Sólo se muestra uno de los dos paneles a la vez: las funciones *change_view_to_unit* y *change_view_to_building* hacen el toggle de que panel mostrar, mientras esconden el otro. Los eventos de *press_event* y *release_event* son utilizados tanto en GameArea como en GameWindow para la selección del objeto actual.

Varias veces por segundo (configurable) a través del evento timeout se llama la función onTimeout de GameWindow. La función limpia gameArea completamente de objetos, por lo que causa la activación de la señal de *draw* de gameArea, y esta redibuja completamente la información del mapa, basándose en un sistema de coordenadas manejado por el objeto Camera. Cabe aclarar que el dibujado es parcial: en el área de dibujo solo se muestra una fracción del mapa entero, y el objeto camera dicta la región a ser dibujada. A través de eventos de *key_press* en GameArea, detectando las flechas del teclado, se puede mover la cámara, usando los métodos públicos proporcionados por el

objeto (moveUp, moveDown, moveLeft, moveRight). La función onTimeout, además, actualiza la información de el panel abierto actual. Esto es necesario así se puede ver como el tiempo de fabricación de una unidad va disminuyendo en vivo, o como disminuye la vida actual de una unidad que está siendo atacada.

La selección de unidades se lleva a cabo principalmente en GameArea. Los eventos *on_key_press_event* y *on_key_release_event* se llaman cuando el usuario apriete y levante un botón. Lo que se decidió hacer es, pensando generalizar el código para eventualmente implementar una selección grupal de unidades, tomar las coordenadas iniciales del puntero cuando se aprieta el click izquierdo y tomar las coordenadas cuando se levante el puntero. Así se obtiene un rectángulo de coordenadas, y, haciendo una conversión de píxeles a coordenadas internas de juego, marcar como seleccionadas todos los elementos del juego que caigan dentro de ese rango. Luego de hacer una selección, se marcan dos booleanos: unitsSelected y buildingsSelected, que indican si en este ciclo de selección se encontraron unidades y edificios, respectivamente, seleccionados.



La selección continúa luego en GameWindow, puesto que el evento es propagado hacia arriba. Aquí se deicidó dividir varias acciones en dos botones distintos para simplificar el algoritmo. Los eventos con click izquierdo son puramente de selección: si en el área de dibujado se seleccionaron unidades o edificos nuevos, marcamos como selección actual el primero de ellos y se actualiza el panel de costado apropiadamente. Sin embargo, con el click derecho se pasa a las acciones de movimiento o ataque: al hacer click derecho con una unidad seleccionada, se intenta mover a un espacio vacío, o si está ocupado, atacar al ocupante. Cada una de estas acciones desemboca en un envío del comando apropiado al servidor.

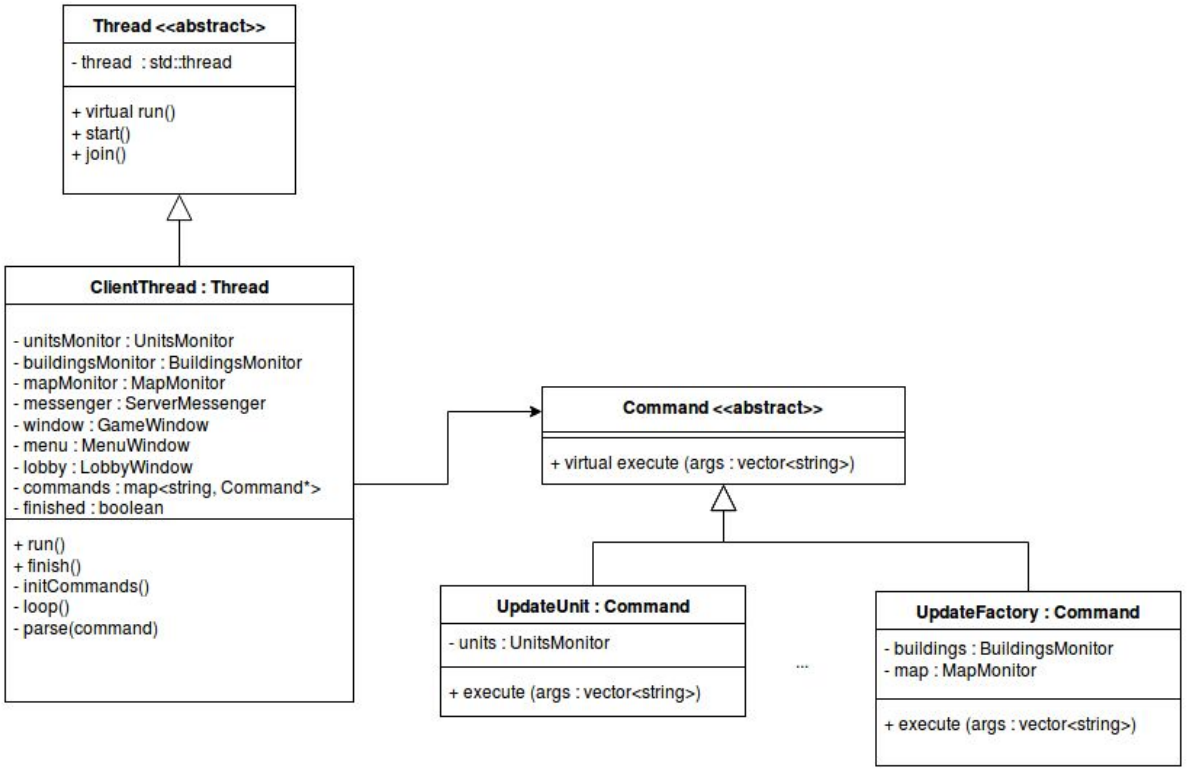
Luego hay un tercer evento, que escucha el presionado de la tecla Escape del teclado, para deseleccionar a todas las unidades seleccionadas. Fue implementado por un tema de comodidad. Al deseleccionar, se limpia la información al costado de la pantalla.

Los eventos de presionado de botones de la fábrica son implementados en GameWindow. los métodos *factory_next, factory_prev, factory_create* son los llamados cuando se presionan los botones con los nombres respectivos. Son simples envíos de mensajes al servidor.

Sobre los panales UnitPanel y BuildingPanel, vale la pena mencionar que los métodos de actualización de los campos nunca actualizan la etiqueta gráfica directamente, sino que modifican una variable que luego es leída desde el onTimeout para hacer la modificación. Ante la incertidumbre de como funciona GTK con los eventos y que tan seguro es la modificación concurrente de los textos mostrados en pantalla, se optó encararlo de esta manera, para asegurarse que no haya posible condición de carrera haciendo todos los cambios ocurran a través de los eventos de la librería.

Comunicación con el servidor

El hilo de comunicación fue nombrado ClientThread. Bajo este hilo se basa la mitad receptora de la comunicación con el servidor, es decir, este hilo está constantemente recibiendo comandos, los parsea y los ejecuta. Su diagrama de clases es más simple

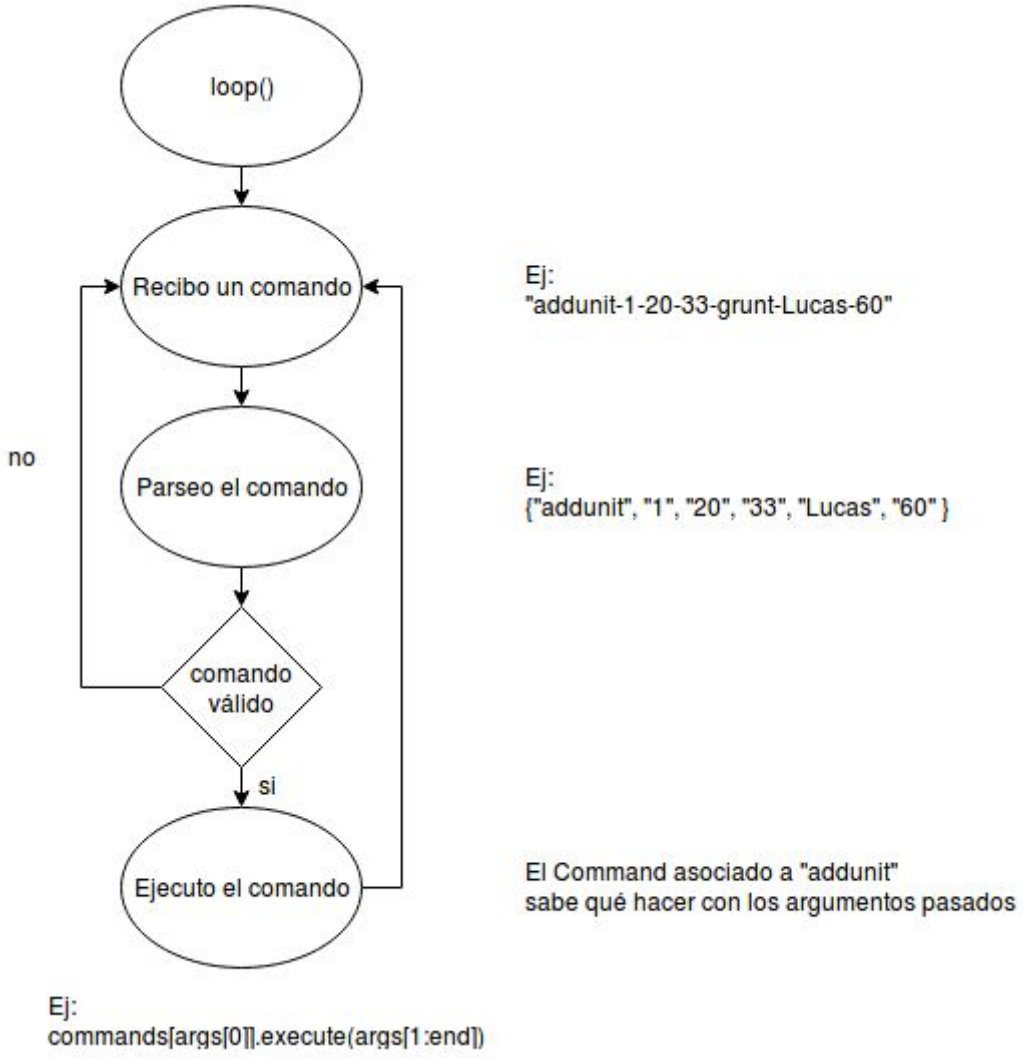


Al constructor del hilo ClientThread se le pasan todos los monitores del juego, necesarios para efectuar cambios a partir de los comandos recibidos. Este hilo es instanciado apenas se establece una conexión válida con el servidor, es decir, apenas el usuario entró al menú principal del juego, pasada la ventana inicial de conexión. El método `run` levanta al thread y lo pone en marcha. Este método lleva a cabo dos operaciones: inicializa los comandos y luego ejecuta el ciclo de comunicación con el servidor.

Los comandos que se reciben del servidor fueron modelados cada uno como una clase que implementa a la abstracta Command. Cada comando tiene un identificador y una cantidad de parámetros. El identificador es utilizado en ClientThread como la clave del mapa `commands`, y el comando apropiado es el valor al que apunta esa clave. En la función initCommands, entonces, se instancian todos los comandos de este estilo, y se agregan al mapa.

El ciclo principal de la comunicación es implementado en el método `loop`. El algoritmo es el siguiente: se recibe un comando como tipo string, se lo divide en parámetros, se busca al comando asociado con el identificador recibido, y se ejecuta con los argumentos recibidos. Luego se repite la operación hasta que el hilo reciba la orden de finalizado, que sucederá sólo cuando el cliente cierre el juego.

A continuación se muestra el simple flujo del loop, con un ejemplo al costado. Esta implementación tiene por ventaja la fácil selección del comando a ser ejecutado, no hay que hacer grandes chequeos condicionales para buscar el comando en particular.



El ciclo siempre estará a la espera de recibir un comando del servidor. La comunicación se hace a través del objeto ServerMessenger, que internamente implementa la comunicación de redes a través de las funciones de *sockets* dadas por la librería estándar de C, que fue utilizada en este caso para ser bloqueante a la espera de un comando.

Los comandos en particular tienen todos implementaciones bastante diferentes, y pueden verse en el módulo de protocolo. Son todos muy variados y controlan el flujo entero del juego, desde que el jugador está en el menú principal y pide unirse a una partida específica, pasando por todos las actualizaciones de estado de todas las unidades y entes en general del juego, hasta hasta la notificación de victoria o derrota que indica el fin. Por

eso es necesario que este hilo tenga referencias a todos los recursos compartidos, y hasta algunas ventanas para actualizar las vistas.

Librería de clases comunes a ambos módulos

Dentro de el cliente y el servidor se encontraron algunas clases que se usan por igual en ambos lados, en particular las clases relacionadas con la concurrencia y algunas que encapsulan el comportamiento necesario de redes.

Existe la clase Thread mencionada anteriormente, que se usa como base de cualquier clase que quiera implementarse para representar un hilo de ejecución independiente. Es utilizada en todo momento que se habla de hilos en este proyecto. Asociada a la concurrencia también se cuenta con una simple clase Lock, que implementa el bloqueo de *mutex* de manera RAII, que automatiza el desbloqueo del mismo cuando la variable de tipo Lock salga del scope de un bloque.

Sobre redes, se cuentan con dos clases (y una excepción): Socket, y Messenger. Socket actúa como encapsulador de las funciones de la librería estándar de C para implementar los sockets. El Socket implementado cuenta con dos constructores distintos, por si se quiere actuar como servidor y escuchar conexiones entrantes en un puerto, o establecer una conexión a otro sistema. Se aprovecha el uso de excepciones para informar al usuario programador que una conexión fue rota imprevistamente, por ejemplo cuando la conexión es cerrada remotamente y se estaba esperando recibir un mensaje.

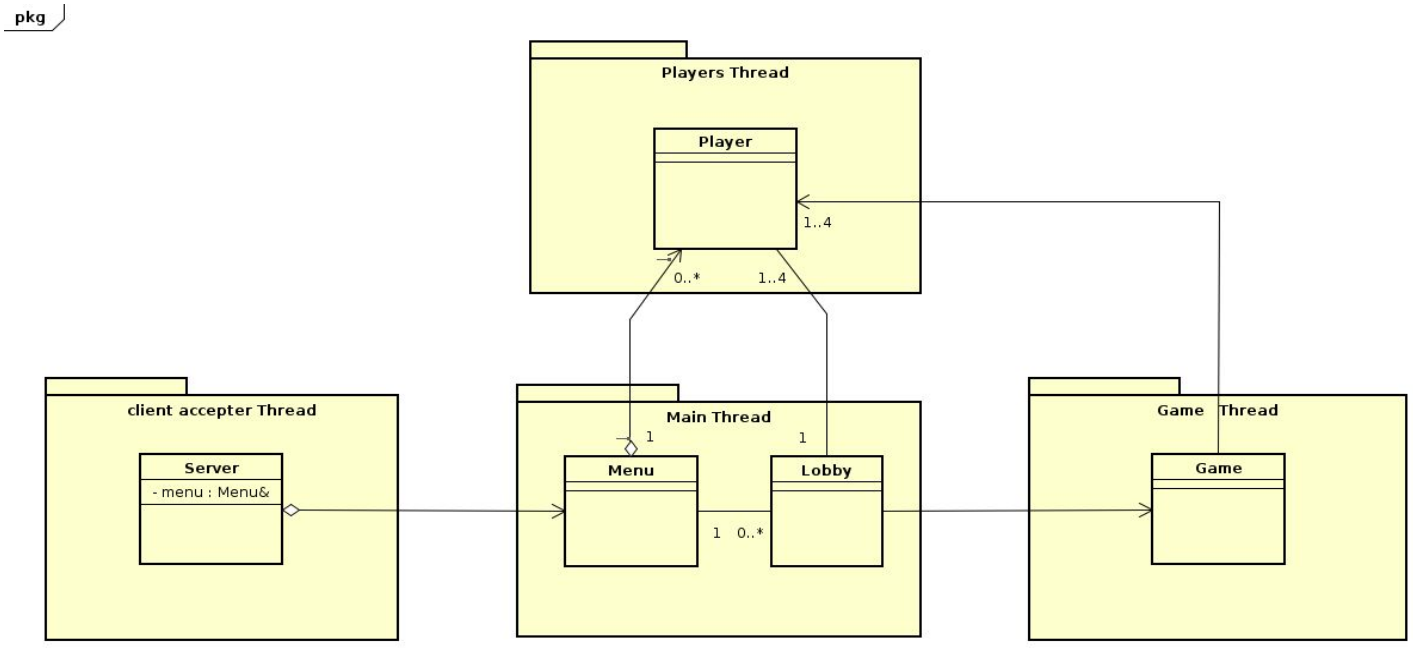
El Messenger, en cambio, es una clase de comunicación ya de nivel más alto. Su función permite el enviado y recepción de mensajes de manera directa, es decir, usando métodos que piden y devuelven únicamente tipos *std::string* de la STL de C++. Internamente la clase sigue un protocolo de comunicación en donde primero se mandan 4 bytes que representan la longitud del mensaje en *big endian* (incluyendo el caracter nulo al final de cada string), y luego el mensaje en cuestión. Habiendo establecido ese protocolo de antemano, la función de recepción primero debe recibir exactamente 4 bytes desde el socket, para luego recibir la cantidad determinada por el primer mensaje. Esta forma de trabajar permite abstraerse en el resto del proyecto de la comunicación, y luego de un leve depurado inicial se pudo asegurar que nunca hubo problemas relacionados a los sockets a lo largo del desarrollo del proyecto.

Módulo Servidor

El servidor se inicia pasando como parámetro el archivo de configuración que se utilizará para cargar las distintas características de los objetos pertenecientes al juego, además del mapa en sí.

Al principio se crea un Menu en el Thread principal del servidor y se instancia, en otro Thread un objeto Server que se encarga de aceptar clientes a medida que se conectan al juego.

Una vez conectados estos nuevos clientes se crea una nueva clase Player para cada cliente nuevo, siendo esta la clase a través de la cual van a interactuar todo el programa con el servidor. Cada Player corre en su propio Thread para no interrumpir la ejecución del programa.

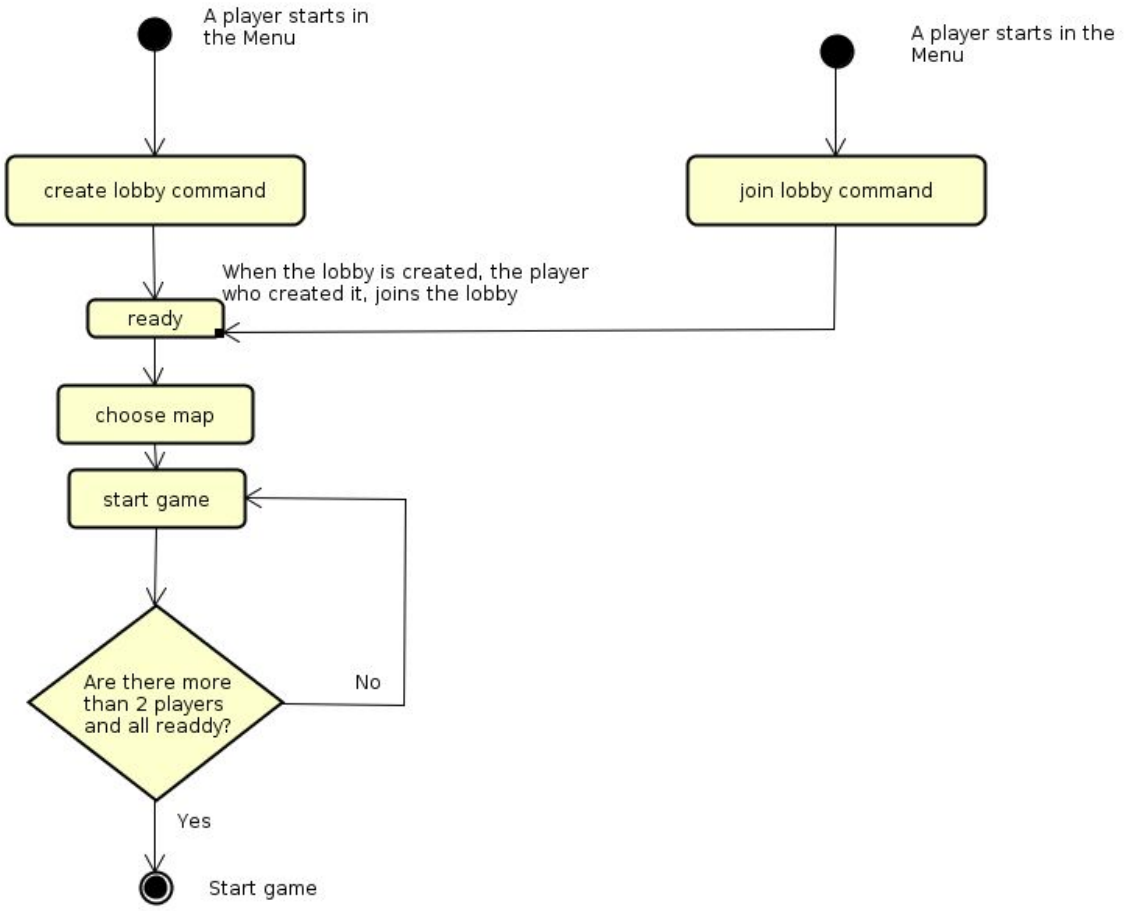


Una vez instanciada la clase Player, en primer lugar se encuentra en el Menu, donde será capaz de unirse a un Lobby o crear Lobbys nuevos.

El Lobby funciona como el lugar donde se configura la partida, se elige el mapa y se van agregando jugadores hasta completar el cupo de 4 jugadores como máximo. Una

partida puede empezar solo si existen 2 o más jugadores en el Lobby , y además todos los jugadores apretaron el botón de ready del lado del cliente. Dadas estas condiciones, una vez elegido el mapa se puede empezar una partida.

La partida se empieza con la creación de un Game, que vive en su propio Thread. Dentro del Game se leen las configuraciones pedidas en el Lobby y se crean todos los objetos necesarios para iniciar el juego.



Para cargar todos los datos necesarios para el juego, dentro de Game se instancia un MapLoader quien se encarga de cargar los terrenos del mapa, los objetos de la naturaleza, y los fuertes y edificios.

Una vez logrado esto se crean las unidades iniciales de cada jugador y se crea una ControlUnit, encargada del loop de juego. Esta última clase interactuar con todo el modelo para poder garantizar un correcto funcionamiento del programa, permitiendo a los usuarios realizar los comandos requeridos y enviando la información de lo que está sucediendo.

Explicación del modelo

Todos los objetos que se dibujan del lado del cliente se encuentran representados en el servidor.

Existe una clase padre llamada Teamable. En ella se guarda la información requerida del equipo en el que se encuentra un objeto y el Size del objeto, que es básicamente, el espacio que ocupa en el mapa.

Solo existe un objeto que es Teamable puro que es la representación de las banderas en el mapa. Esto es así, para que cualquier unidad pueda caminar sobre las banderas.

La clase Size consiste en la posición de izquierda y arriba del objeto, y que tan largo y que tan ancho es en el mapa. Esta resultó la solución para asegurarnos que no hayan dos objetos que ocupan un lugar en el mapa, no se choquen, o para saber si se están chocando, como se utilizó en el caso de las balas del juego.

Heredera de Teamable existe la clase Occupant. Los Occupant son aquellos objetos que ocupan un lugar en el mapa. Para asegurarnos de que no hayan dos objetos en la misma posición del mapa se hacen chequeos contra Occupants, utilizando el Size heredado de Teamable. Esta misma guarda el tipo del objeto, que nos indica si es una roca, un grunt o una fábrica el Occupant con el que estoy interactuando.

Un objeto Occupant puro son las rocas, cuyo team es “Neutral” y su tipo es “Rock” y tiene un Size configurable.

Herederas de Occupant existen dos clases Unit y Factory.

La clase Unit es la que representa cualquier unidad en el mapa, sin importar que sea un vehículo o un robot. Esta es la que se encarga de, en cada TIC del loop de juego, hacer una micro acción, dependiendo de si la unidad está atacando, en movimiento, yendo a capturar una bandera o simplemente parada en el mapa. Cada uno de los estados lleva a hacer una acción diferente.

Para el movimiento la Unit cuenta con una clase Compass que es la que se encarga de utilizar un algoritmo A* para saber cómo llegar a la posición pedida por el usuario.

Para el ataque cuenta con un Size que en vez de representar su tamaño, representa el rango de alcance de su ataque. Además tiene una clase Weapon, que se encarga de crear las municiones de cada unit.

Cuando quiere ir a tomar una bandera o un vehículo neutral utiliza el compass ya mencionado para saber a donde ir, y también, todas las clases tienen un rango para el comando grab a una distancia de +1.

Finalmente, cuando la unidad mantiene la posición, hace un chequeo de que no haya enemigos cerca. De haber enemigos lo ataca, pero no los persigue.

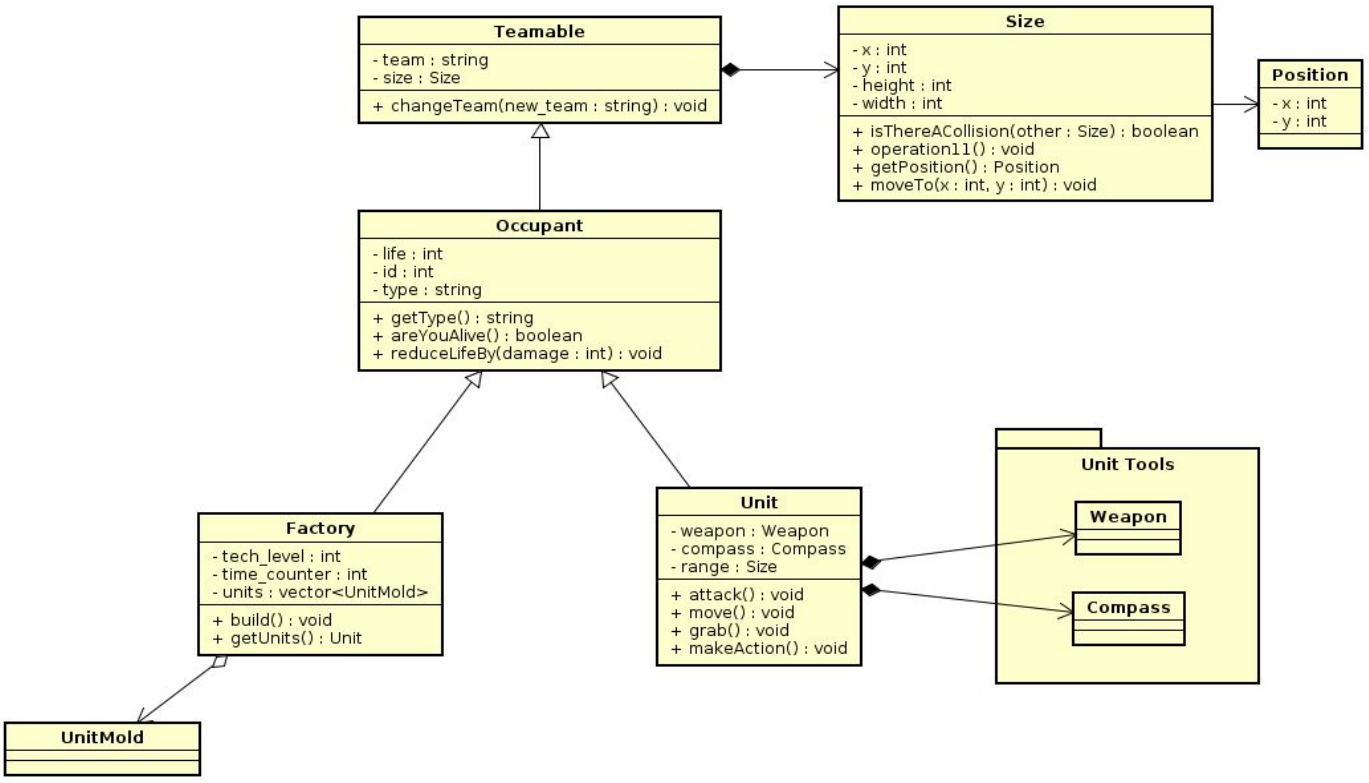
Todas las unidades van a priorizar los comandos del usuario.

Si una Unit muere, desaparece del modelo.

La clase Factory es quien se encarga de crear las unidades. Puede recibir comandos para que el usuario sepa que robots puede crear y además puede iniciar la creación de unidades con el botón correspondiente. Para lograr esto, Factory cuenta con unos UnitMolds que son clases creadas a través del archivo configurable, con los datos necesarios para crear cada Unit. Además del vector con UnitMolds también tiene un mapa de Weapons para saber qué Weapon darle a cada Unit en el momento de crearla.

Factory son las fábricas de robots, las de vehículos y los fuertes. La diferencia entre ellos es el Type y el vector de UnitMolds que tienen.

A diferencia de las units, si una Factory llega a tener vida 0, no desaparece en el modelo, pero quedan inhabilitadas todas sus funciones. Esto se refleja con las ruinas del lado del cliente.



Para completar la correcta interacción del modelo, existe una clase Map que se encarga de tener una referencia al vector de Occupants para conocer exactamente donde se encuentran todos los objetos que interactúan en el modelo. Además es la clase que guarda el conjunto de Cell que es la clase que guarda para un Size específico el tipo de terreno que hay en ese conjunto de posiciones.

La clase Game es quien contiene el vector de Occupants y son Map y ControlUnit clases con la referencia a ese vector, para lograr que siempre esté sincronizado el mapa con el modelo.

Explicación del movimiento de unidades

Como ya se mencionó, todas las Units tienen un Compass que calcula utilizando el algoritmo A* para conocer cómo ir a alguna determinada posición en el mapa. Para su mayor eficiencia y velocidad del algoritmo se le hicieron algunas modificaciones.

El algoritmo original recorre posición por posición, y se fija si la unidad en cuestión puede o no ocupar esa posición. Al ser un mapa con muchas posiciones se le introdujo al

algoritmo lo que llamamos una aceleración y desaceleración. Esta misma consiste en lo siguiente:

Desde la posición de donde se encuentra el objeto, se dibuja un área cercana en la cual el algoritmo recorre posición por posición si puede o no pasar por esos puntos. una vez que sale de esa zona, se toman como nodos adyacentes aquellas posiciones que estén a una distancia n, definida en el algoritmo. De esa manera, se recorren menos nodos para llegar a mi destino. Para asegurarnos de encontrarlo, cuando estoy en un área cercana al destino, se vuelve a calcular posición por posición, y de esa manera encuentra el punto final.

Para que esta aceleración no altere la finalidad del algoritmo, por más que no se vean todos los nodos, si se va sumando la G de los nodos salteados para saber realmente el costo de llegar al nodo, teniendo la aceleración.

Explicación del ataque de unidades

Como ya se mencionó, las Units utilizan la clase Weapon para los ataques. Para lograr esto, la clase Weapon tiene una Bullet, capaz de calcular un camino recto hacia un objetivo, y con esta se fija si la unidad está en rango de ataque o no. Esto quiere decir, que si una Unit tiene un enemigo que está dentro del Size del rango, pero en el medio tiene un edificio, roca, o unidad, no va a poder atacar, por lo que se tiene que acercar más al objetivo. Si el objetivo se encuentra fuera de rango, la unidad se acerca hasta estar en rango y recien ahi empieza a disparar.

Las balas son creadas en Weapon y las guarda la Unit. Dentro del TIC de la ControlUnit, le pide todas las balas que tiene la Unit y las pone en movimiento en el modelo. Las balas siempre recalculan su camino al objetivos, por lo que no pueden fallar.
Una vez que el Size de la bala choca con el Size del objetivo, es este último se le proporciona un daño correspondiente y la bala desaparece.

Finalización de la partida

La partida finaliza cuando queda un único jugador con su fuerte con vida. Terminada la partida los jugadores tienen la opción de volver al Menu si lo desean.

Protocolo de comunicación entre cliente y servidor

Comandos dentro del Game

Enviados por el servidor

1. - addunit-[id]-[x]-[y]-[type]-[Team]-[lifePoints] : comando para indicarle al cliente que agrege la unidad de id [id], en la posición (x,y) del mapa, del tipo Type, que tiene un equipo Team y la cantidad de vida lifePoints
2. - addbuilding-[id]-[x]-[y]-[type]-[Team]-[lifePoints]
3. - addnature-[id]-[x]-[y]-[type]-[Team]-[lifePoints]
4. - updateunit-[id]-[state]-[x]-[y]-[lifePoints]-[Team]
5. - updateoccupant-[id]-[x]-[y]-[lifePoints]
6. - updateterritory-[id]-[team]-[x]-[y]
7. - updatefactory-[id]-[unitType]-[min]-[sec]-[lifePoints]-[team]