Documentación Técnica

Requerimientos de software

SDL

Para correr el TP es necesario tener SDL2 instalado.

```
yum install SDL2-devel
apt-get install libsdl2-dev
```

También SDL_Image

```
yum install SDL2_image-devel
apt-get install libsdl2-image-dev
```

Y por último SDL_ttf y SDL_Mixer

```
yum install SDL2_ttf-devel
apt-get install libsdl2-ttf-dev
yum install SDL2_mixer-devel
apt-get install libsdl2-mixer-dev
```

tinyXML2

También es necesario tener TinyXML2 instalado.

```
apt-get install libtinyxml2-dev
```

Para tener la versión más nueva se puede usar el código fuente del github del protyecto: https://github.com/leethomason/tinyxml2. Instrucciones para instalar la librería usando el código fuente del Repositorio pueden encotnrarse aquí: https://charmie11.wordpress.com/2014/03/24/script-for-install-tinyxml2-on-ubuntu/ (recurso en inglés).

Make

Una vez instaladas las librerias

```
cd build
cmake ..
make
```

Ejecución

Esto generara el ejecutable **vista** dentro de la misma carpeta client, el ejecutable **server** dentro de la carpeta server y el ejecutable **generador** dentro de la

carpeta generadorMapa.

Principalmente se debe generar un mapa mediante:

```
cd generadorMapa
./generador
```

Este generara un archivo `mapa.map` y un archivo `configuracion.xml`. El primero contiene los territorios del mapa, y el segundo las posiciones de los objetos iniciales en el mapa. Ambos deben ser movidos a la carpeta server.

Para correr el tp se debe correr primero el servidor especificando:

```
cd server
./server <puerto> <#jugadores>
```

y luego la cantidad de clientes necesarios, ejecutando consecutivamente:

```
cd client
./vista <ip> <puerto>
```

Descripción general

Como se mencionó anteriormente el trabajo consta de tres partes principales, un servidor, un cliente y un generador. El servidor contiene al modelo del Juego y se encarga de la lógica del mismo. El cliente, se comunica constantemente con el servidor y se encarga de reflejar lo que sucede en el modelo de forma visual. Por último, el generador de mapas es el encargado de generar las configuraciones iniciales del juego, que son leidas al inicializarse el juego en el server.

Server

Descripción general

La parte fundamental del servidor es el modelo el cual se encarga de manejar los sucesos y funcionamientos del juego. Pero además, este modelo requiere de "representantes" de los clientes dentro del servidor, los cuales se encargan de realizar las comunicaciones con los clientes.

Clases

- **Juego**: Es la clase principal del modelo. Esta clase contiene a los jugadores, a los objetos (Unidad, Edificio, Bandera, Bloque, etc), a las fábricas de los objetos (FabricaUnidades, FabricaMuniciones, FabricaEdificios, FabricaInmovibles) y a dos colas de mensajes (una para recibir y otra para enviar). Esta clase hereda de Thread, puesto que se busca que se ejecute en un hilo propio. El ciclo principal del juego se encuentran en el método `run`, en el se va llamando a los métodos `chequearColisiones`,

`eliminarMuertos`, `actualizarDisparos` y `actualizarEdificios`. Estos métodos se encargan de actualizar el modelo en base a las acciones realizadas por los jugadores (recibidas en `actualizarRecibidos`), y posteriormente se envían estas actualizaciones a los jugadores (mediante `enviarMensajesEncolados`).

- **Jugador**: Esta clase se encarga de representar a los clientes dentro del servidor, es decir es la que se encarga de establecer las comunicaciones. Al igual que juego, ésta hereda de Thread, ya que cada uno de los jugadores deben estar esperando información de los clientes de forma simultanea. Por lo tanto, en el método `run` el Jugador está constanmente esperando recibir mensajes, los cuales se encolan en la colaDeRecibidos (la misma cola que tiene el servidor). Además contiene un método enviarMensaje que permite establecer una comunicación con el cliente en el sentido inverso al anterior. Estas comunicaciones son posibles ya que contiene un atributo Socket.
- **Objeto**: Todos los objetos presentes en el mapa heredan de Objeto (cuenta con caracteristicas básicas como son la vida, posicion, dimensiones, un identificador y un tipo). Mediante el double dispatch hace posible el chequeo de colisiones entre distintos tipos de objetos, actuando en cada caso en consecuencia. De esta clase heredan **Movible** e **Inmovible**, cada cual con sus herederos respectivos (**Unidad**, **Municion**, **Bandera** y **Bloque**).
- **Fabricas**: La creación de la mayoría de los distintos elementos integrantes del modelo se realiza mediante fábricas. Éstas continenen las características particulares de cada tipo de objeto (leido mediante xml al ser instanciadas) y las van cargando en los objetos devueltos. El modelo contiene varios tipos de fábricas: **FabricaUnidades**, **FabricasTerrenos**, **FabricaMuniciones**, **FabricaInmovibles**, **FabricaEdificios**, etc. En general el método más importante de lás fabricas es el que devuelve un elemento del que es fabricado (ejemplo, getUnidad, getTerreno, getMunicion, etc.), pero además, en algunos casos, se utiliza para conocer de forma externa algunas características de las mismas (getAlcance, getTiempo, etc.).
- **AEstrella**: Es una clase que se encarga de calcular los recorridos AEStrella para cada tipo de unidad (para eso utiliza la función `getRecorrido`), desde una posición de origen a una de destino.

Diagramas UML

En primer lugar, para lograr entender el modelo, es necesario comprender como están relacionados los objetos que están involucrados en el mismo. Como se mencionó anteriormente, todos los objetos que forman parte del juego propiamente dicho heredan de la clase **Objeto**. En el siguiente diagrama de clases se puede observar como se relacionan los mismos.

Como se puede ver, hay dos clases principales que tienen como padre **Objeto**, estas son **Movible** e **Inmovible**. Como sus nombres lo indican, la diferencia

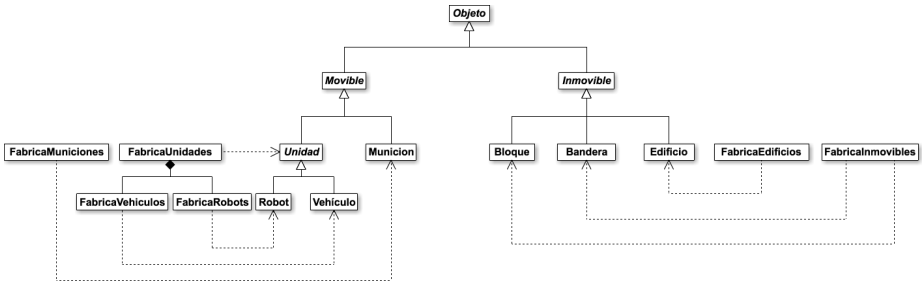

Figure 1: Herencia de los distintos tipos de objeto

entre estos es que los objetos del primer grupo tienen la logica necesaria para poder moverse, mientras que los del segundo grupo no. A su vez, de **Movible** heredan **Unidad** y **Munición**. **Unidad** incluye a los personajes que pueden ser movidos por los jugadores correspondientes. Las unidades guardan la lógica necesaria para realizar disparos y capturar banderas. **Municion** es la clase que representa a las balas generadas cuando una **Unidad** desea efectuar un disparo. Por otro lado, de **Inmovible** se observa que se desprenden **Bloque**, **Bandera** y **Edificio**. El primero, es simplemente un objeto que ocupa un espacio en el mapa, el segundo ocupa un espacio en el mapa pero además al pasar una **Unidad** por el mismo la bandera es capturada, y por último en **Edificio** se guarda la lógica de la creación de unidades (desde el punto de vista del modelo, no es la clase que genera las instancias de **Unidad**).

Dicho esto, es importante mencionar que esta estructuración permite que cualquier objeto pueda colisionar con cualquier objeto, y que internamente dependiendo de las clases de los mismos, las colisiones se manejen según correspondan. Veamos simplemente los métodos de **Objeto** que permiten esto:

Figure 2: Metodos de colisión, en Objeto

Como se puede ver, todo objeto debe saber colisionar con cualquier instancia de una clase heredra de Objeto.

Pero la clase principal del modelo, no es **Objeto**, sino **Juego**. Esta clase es la más extensa e importante del modelo, y tiene la funcionalidad principal de **manejar y modificar los estados del modelo**. Pero estas modificaciones deben ser desencadenadas a partir de mensajes recibidos del cliente. Por lo tanto el **Juego** debe también **interactuar con los jugadores**. Por lo tanto, se puede decir que es una clase con dos "caras". Veamos la primera:

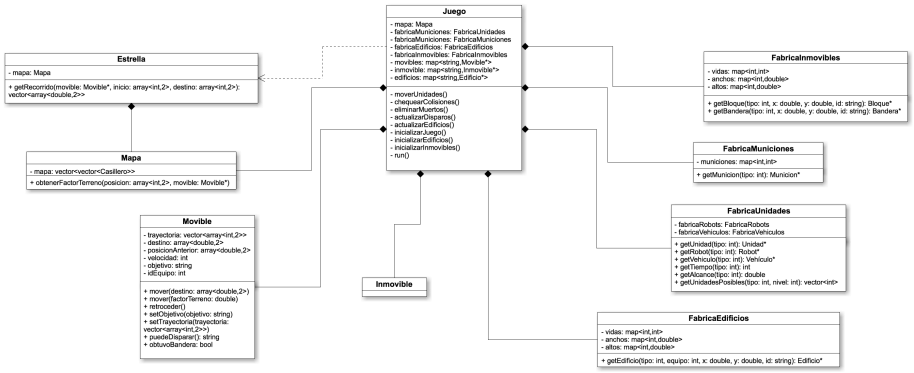

Figure 3: Juego como administrador de estados del modelo

Descripción de archivos y protocolos

Client

Descripción general

El Cliente es el encargado de reflejar el estado de juego mediante animaciones. Asimismo es el encargado de reportar todas las acciones del usuario al servidor, que serán luego reflejadas en el modelo del juego.

Clases

El funcionamiento general del cliente puede ser resumido a tres clases principales: **PaqueteReceiver**, **PaqueteSender** y **Canvas**. Las primeras dos son las que se encargan de la comunicación por sockets con el servidor y la última es la que interactúa con el cliente y renderiza todo el juego. ###### PaqueteReceiver Esta clase es la encargada de recibir los paquetes que llegan del servidor. Esta constantemente recibiendo por el socket y cuando algo llega, lo encola en **ColaPaquete** para que sea procesado por **Canvas**. ###### PaqueteSender Aquí se envían los paquetes previamente encolados por **Canvas** ###### Canvas Esta clase es la que maneja todas las acciones del cliente y todas las visualizaciones, aquí es donde SDL es utilizado.

Diagramas UML

Descripción de archivos y protocolos