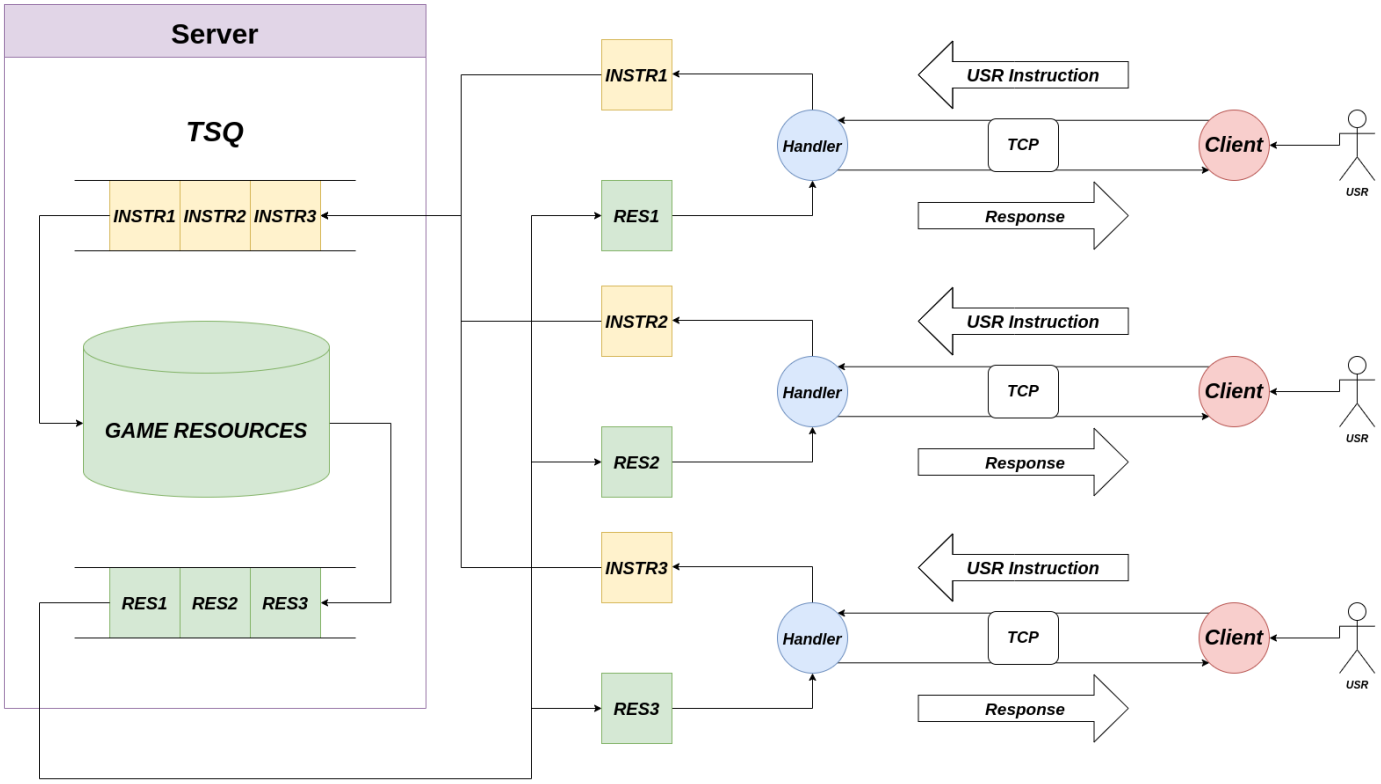
Documentación Técnica

Arquitectura

El programa se basa en un modelo de comunicación **Cliente-Servidor** mediante un protocolo **TCP**. El programa **Servidor** contiene y administra los recursos del juego y maneja la comunicación con los clientes.
Cada **Cliente** es un programa encargado de transmitir los eventos del mouse y teclado del **Usuario**. Del lado del Servidor se encuentra una clase *manejadora* que se encarga de convertir estos eventos en instrucciones concretas para el servidor. Estas instrucciones son almacenadas en una **Thread Safe Queue** que al llenarse pasa a ser recorrida secuencialmente, ejecutandose cada instrucción sobre los recursos del juego para actualizar su estado.
Para cada **Instrucción** el Servidor produce una **Respuesta**. Esta incluye por un lado información específica para cada cliente en función de la instrucción enviada y por el otro información sobre el estado actual del juego que será la misma para cada cliente.

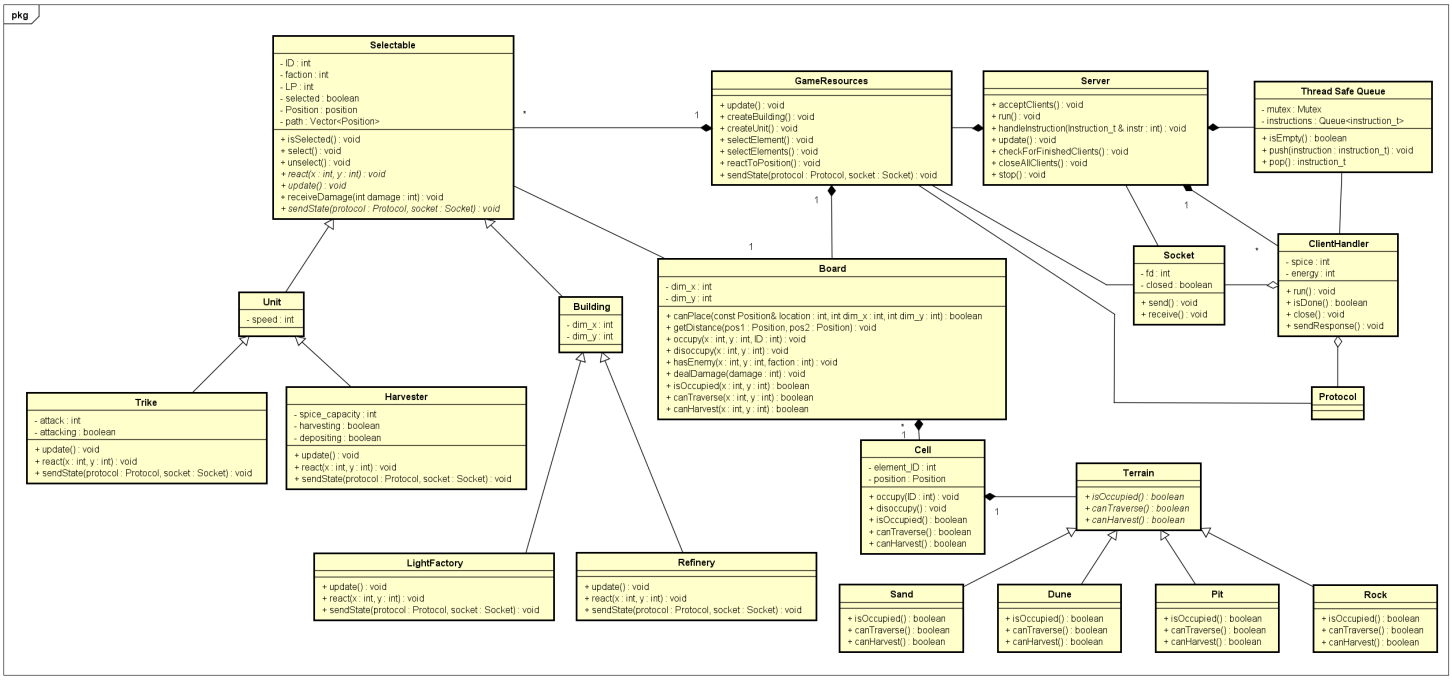


Para la Interfaz de usuario se utilizaron los frameworks **Qt** y **SDL2** (en particular, la librería de wrappers C++ **SDL2pp**). El programa Cliente solo se encarga de procesar eventos del mouse y teclado y de recibir la respuesta del Servidor a partir de la cual produce las vistas en pantalla para el Usuario. Se busco de esta forma lograr una adecuada separación de responsabilidades, donde toda la lógica de manejo de los recursos del juego quede encapsulada del lado del Servidor mientras que el manejo de Qt y SDL2 queda encapsulado del lado del Cliente.

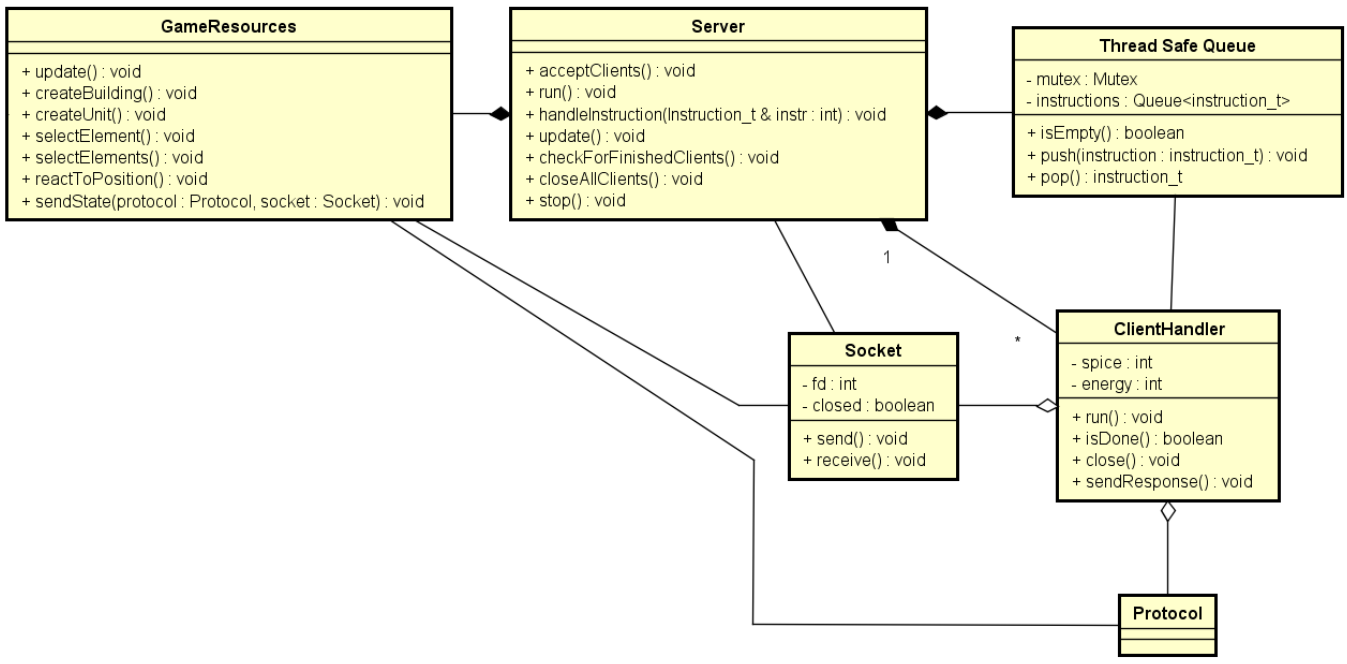
IMPORTANTE: Los siguientes diagramas de Clases y descripciones no son una explicación detallada de cada aspecto del código. Para priorizar la claridad conceptual, aquellos atributos, métodos, estructuras de datos e incluso clases que se consideren auxiliares a la funcionalidad principal del programa fueron **excluidos**.

Programa Servidor

Diagrama de Clases



Comunicación



Descripción General

En una primera instancia, el servidor se encarga de aceptar clientes y esperar nuevas conexiones. Cada nueva conexión se encapsula en un **ClientHandler** que corre en un thread separado.Cuando hay suficientes conexiones para una partida, el servidor ejecuta el loop de juego (también en su propia thread), el cual habilita la lectura de instrucciones del usuario en cada ClientHandler

Server

Atributos

```
Socket accepting_socket;
GameResources game;
std::vector<std::unique_ptr<ClientHandler>> players;
ThreadSafeQueue TSQ;
std::map<int,std::vector<response_t>> responses;
bool running;
```

Donde **response_t** es un tipo enumerativo que representa una respuesta. El mapa de vectores response_t almacena las respuestas (que pueden ser más de una) a la instruccion de cada cliente, y estas respuestas son enviadas por el servidor.

Métodos

```
void acceptPlayers();
```

Este método se ejecuta en la primera parte del programa y su funcionalidad es la de aceptar nuevas conexiones e incorporarlas al arreglo de ClientHandlers. Para el manejo de estos últimos se dispone de métodos.

```
void checkForFinishedClients();
void closeAllClients();
```

Los métodos para el loop de juego son:

```
void run();
void stop();
```

La implementación completa del loop de juego es la siguiente:

```
while (running) {
    checkForFinishedClients();
    while(this->TSQ.getSize() < this->players.size()){}
    for(size_t i = 0 ; i < this->players.size(); i++) { // while (!queue.empty)
        std::unique_ptr<instruction_t> new_instruction = this->TSQ.pop();
        this->handleInstruction(new_instruction);
    }
    sendResponses();
    update();
    enableReading();
}
closeAllClients();
```

Donde

```
void Server::stopGameLoop() {running = false;}
```

Como puede verse, el servidor espera a que los ClientHandlers carguen las instrucciones (encapsuladas en la estructura **instruction_t**) a la cola, luego procesa cada una mediante el método polimórfico `handleInstruction()` , el cual interactua con GameResources y carga las respuestas a estas instrucciones. El Servidor luego envia las respuestas, actualiza el estado del juego y lo envía. Finalmente, habilita una nueva lectura.

ClientHandler

Atributos

```
int player_id;
player_t faction;
int spice;
int energy;
Socket player_socket;
Protocol protocol;
ThreadSafeQueue & instruction_queue;
std::thread thread;
```

Cada ClientHandler tiene asignado un ID y una facción, así como una cantidad de especia y energía. La clase encapsula el Socket recibido al aceptar una nueva conección y la clase Protocolo que maneja la comunicación. Además, debe tener una referencia a la ThreadSafeQueue a modo de ir pusheando las instrucciones que reciba del usuario. Por último, esta clase encapsula su propia thread, la cual se lanza en el momento de construcción de una nueva instancia.

Métodos

```
    void run();
```

Loop principal en donde se reciben instrucciones del usuario y se pushean a la cola.

```
     void sendResponses(std::vector<response_t> & responses);
```

Método que envía por socket las respuestas del servidor

```
     void reportState(GameResources & game);
```

Metodo que envía por socket el estado del juego.

```
     bool isDone();
     void close();
```

Metodos auxiliares utilizados por el servidor.

Protocol

clase que encapsula los métodos de comunicación Cliente-Servidor.

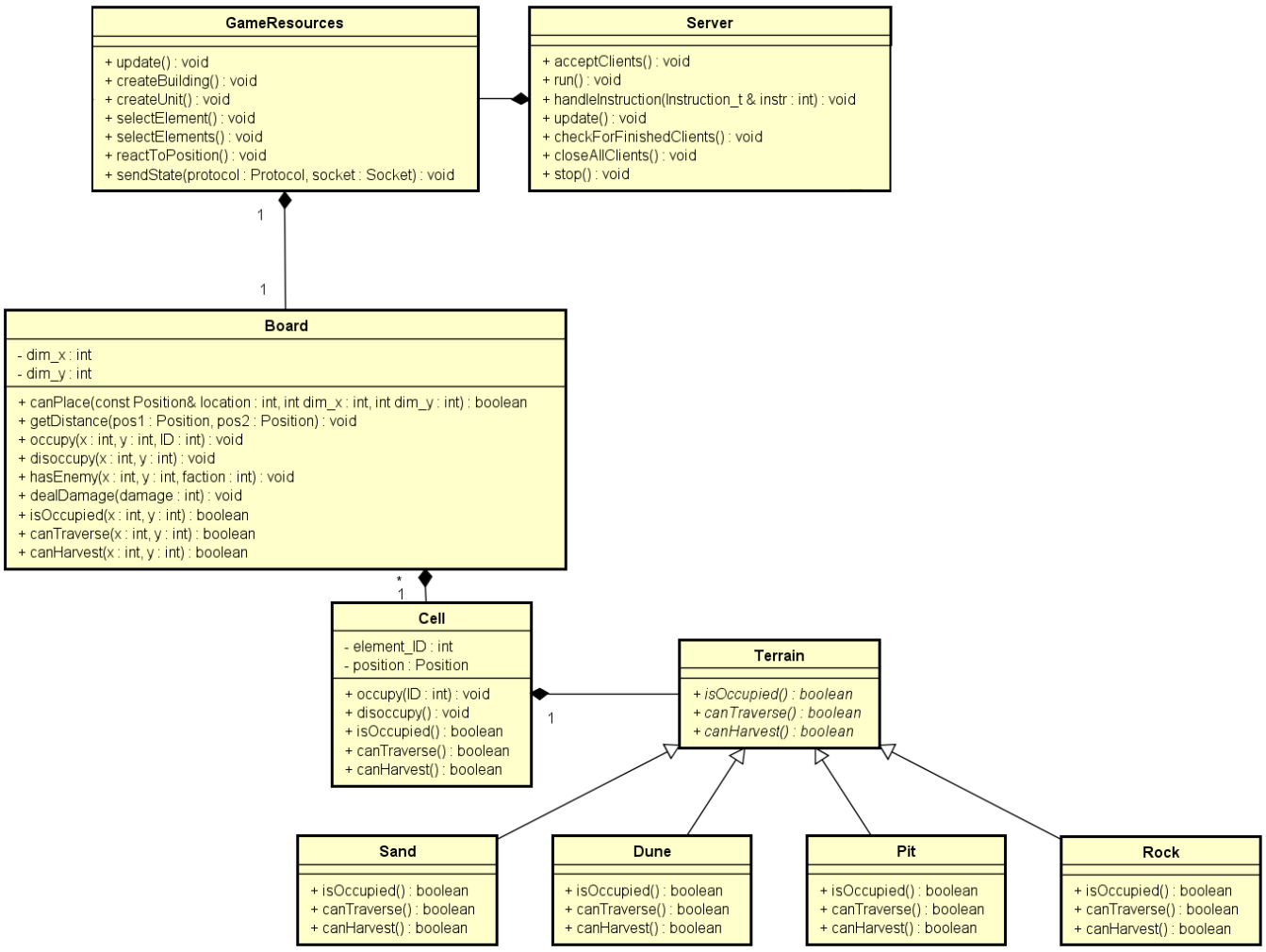
Socket

Clase que encapsula un socket TCP. Esta es utilizada por el Servidor como un socket pasivo que acepta nuevos clientes, y por el ClientHandler como un socket activo por donde se envía y recibe la información del programa Cliente.

Se opto por implementar la clase Socket de la cátedra:

https://github.com/Taller-de-Programacion/sockets-en-cpp

Recursos del Juego



Descripción General

Los recursos del juego son por un lado un **Tablero** y por el otro un mapa de elementos **Seleccionables**. Las instrucciones recibidas por el Servidor de parte de los Clientes pasan a traducirse en operaciones concretas sobre los recursos del juego. Estos actualizan su estado en función de las instrucciones y ese nuevo estado pasa a ser informado a cada Cliente.

GameResources

Atributos

```
std::map<int,std::unique_ptr<Selectable>> elements;
Board board;
```

Contiene tanto al Tablero como el mapa de Seleccionables. Se escogió utilizar un **mapa** en lugar de un arreglo ya que esta estructura de datos facilita la asignación de un **ID** único para cada elemento. Además, el mapa no es a elementos seleccionables en si mismos sino a punteros únicos a seleccionables. Esto fue una decisión de diseño, ya que el uso de punteros permite la implementación de polimorfismo en tiempo de ejecución para estos seleccionables.

Métodos

Los métodos principales son aquellos que encapsulan en forma de operación las instrucciones del usuario. Tales son:

- **Crear Unidad**:
  `response_t createUnit(player_t faction,unit_t type,int & spice);`
  Señalar la creación de una nueva unidad de cierto tipo y facción. Reportar el resultado de la operación. Restar especia en caso de éxito.
- **Crear Edificio**:
  `response_t createBuilding(player_t faction,building_t type,int pos_x,int pos_y,int & spice,int &energy);`
  Señalar la creación de una nueva unidad de cierto tipo y facción. Reportar el resultado de la operación. Restar especia en caso de éxito.
- **Click Izquierdo en el Mapa**:
  `void selectElement(player_t faction,int pos_x, int pos_y);`
  Si hay una unidad en esa posición, marcarla como seleccionada. Deseleccionar todas las otras unidades.
- **Selección del Mapa**:
  `void selectElements(player_t faction,int Xmin, int Xmax,int Ymin, int Ymax);`
  Marcar todos los elementos seleccionables en las posiciones contenidas entre esos límites como seleccionados. Deseleccionar los demás.
- **Click Derecho en el Mapa**:
  `void reactToPosition(player_t faction, int pos_x, int pos_y);`
  Recorrer los elementos seleccionables y, en caso de estar marcados como seleccionados, solicitarles reaccionar a esa posición.

Board

Atributos

```
int dim_x;
int dim_y;
std::vector<std::vector<Cell>> cells;
std::map<int,std::unique_ptr<Selectable>> & elements;
```

Contiene la matriz de elementos de la clase **Celda** que constituyen el Tablero. El tablero delega en cada celda la información específica de cada posición. Además, contiene una referencia a el mapa de seleccionables por motivos que se veran a continuación.

Métodos

El tablero contiene métodos de consulta sobre una posición que son utilizados por los elementos seleccionables, y cuya funcionalidad se delega a la celda en esa posición.

```
bool canDeposit(int x, int y, player_t faction);
bool isOccupied(int x, int y);
bool canHarvest(int x, int y);
bool canTraverse(int x, int y);
bool canSlowDown(int x, int y);
```

Así como métodos de consulta sobre información global del mapa.

```
std::vector<Position> get_traversable_neighbors_of(Position pos, size_t distance);
size_t get_distance_between(Position pos1, Position pos2);
```

El tablero coordina indirectamente la acción entre elementos seleccionables de distintas facciones. Cada elemento seleccionable, como veremos, tiene un ID único asociado. Cada celda almacena el ID del seleccionable que ocupa esa posición, de modo que el tablero puede referenciar a ese elemento a través del mapa. Así, por ejemplo, el método:

```
void dealDamage(int x, int y, int damage);
```

inflige cierto daño 'damage' sobre el elemento que se encuentre en la posición (x,y). Una unidad Harkonnen hace uso del tablero para saber si (x,y) es una posición enemiga, y luego nuevamente para infligir daño sobre esa posición.

Cell

Atributos

```
std::unique_ptr<Terrain> terrain;
Position position;
int element_ID;
```

Cada celda tiene una posición que puede estar ocupada por un elemento (en cuyo caso element_ID es el de aquel elemento), lo bién desocupada (en cuyo caso element_ID = -1). Además, tiene un tipo de **Terreno** que informa las características geográficas de la posición de la celda sobrecargando los distintos métodos de consulta.

Métodos

La celda dispone de métodos de consulta:

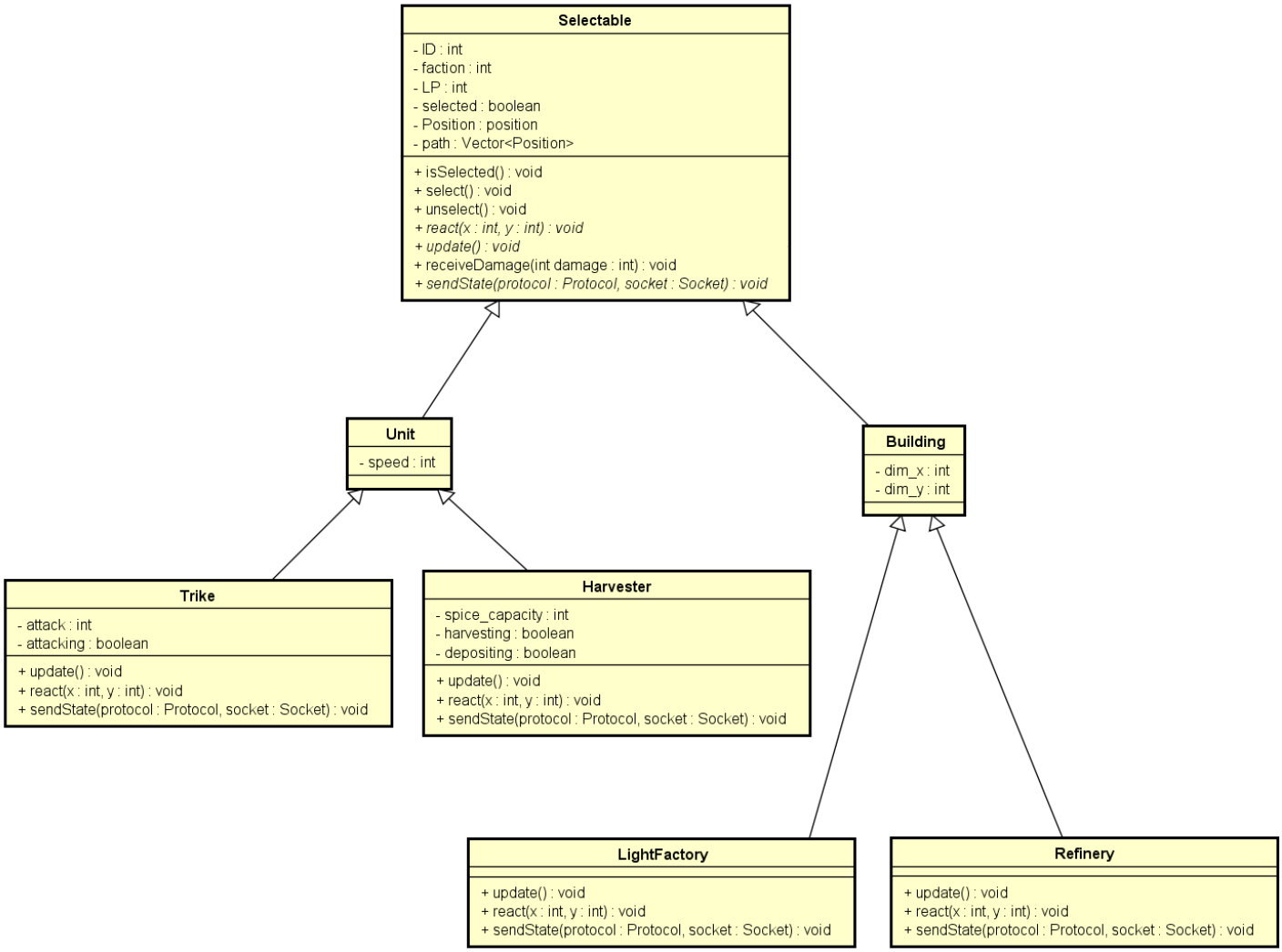
```
bool isOccupied();
bool canTraverse();
bool canHarvest();
bool canBuild();
bool canSlowDown();
```

cuya respuesta cambia en función del tipo de terreno que albergue. Por otro lado, dispone de métodos que alteran su estado.

```
void setTerrain(cell_t terrain);
void occupy(int ID);
void disoccupy();
size_t extractSpice();
```

Cuando, por ejemplo, una unidad cambia su posición, informa de esto al tablero, que procede a desocupar la celda donde esta se ubicaba previamente, y a ocupar la nueva posición.

Seleccionables



Descripción General

Cada unidad y/o edificio en el mapa es un 'elemento seleccionable' con un ID único. A partir de este se heredan las clases **Unit** y **Building** y asi mismo de estas se heredan distintos tipos de unidad y edificio (en el diagrama UML solo se muestran dos de cada uno a modo de ejemplo). El objetivo de esta jerarquía es lograr polimorfismo en tiempo de ejecución. Los métodos:

```
virtual void react(int x, int y, Board& board);
virtual void update(Board& board);
```

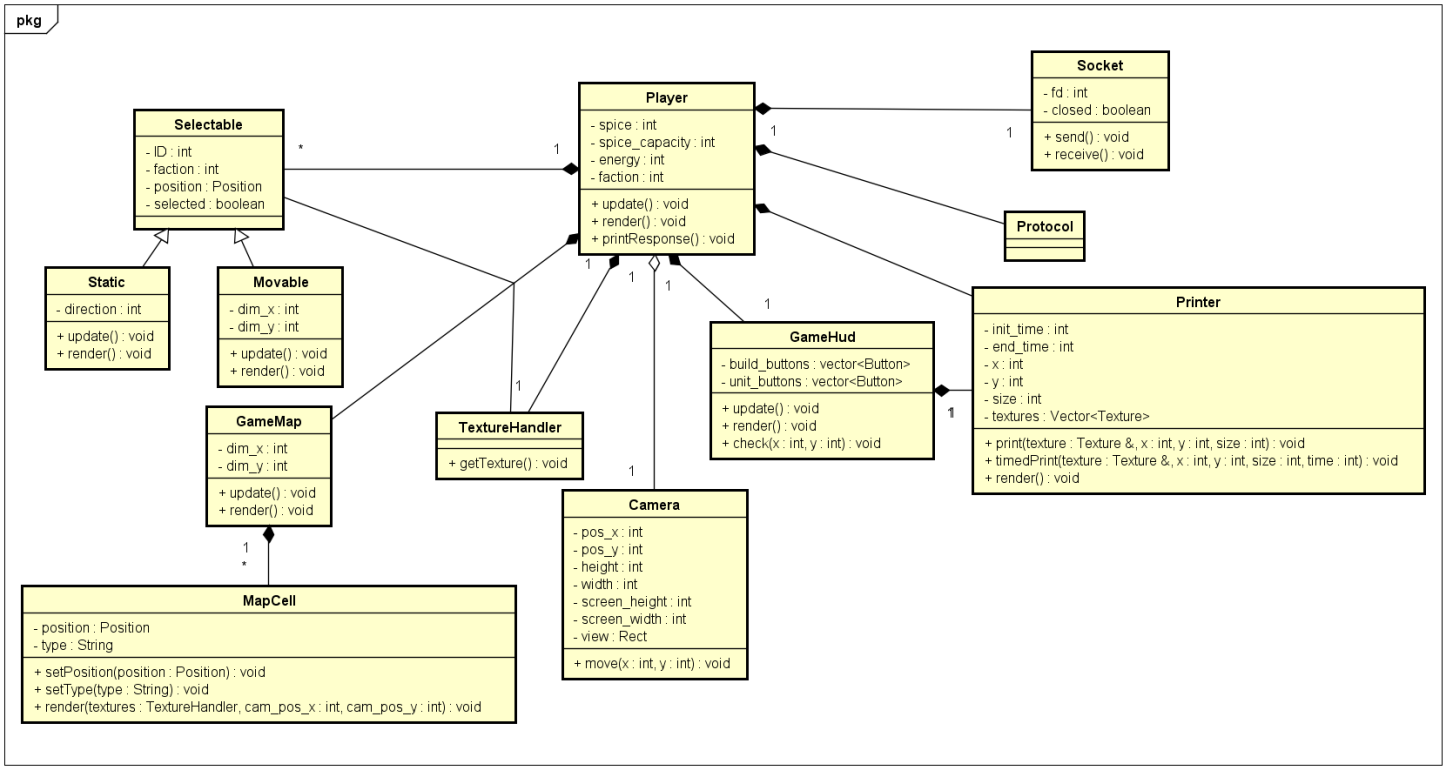
se sobrecargan para cada sub-clase de esta jerarquía. De este modo, si GameResources recibe la instrucción de reaccionar a una posición, lo único que debe hacer es:

```
void GameResources::reactToPosition(player_t faction,int pos_x,int pos_y){
    for (auto& e : this->elements)
        if(e.second->getFaction() == faction){
            if (e.second->isSelected()){
                e.second->react(pos_x,pos_y,this->board);
}
```

Y dejar que cada unidad o edificio decida su respuesta en función de la información del tablero. Luego, cuando el juego se actualiza:

```
void GameResources::update(){
    for (auto& e : this->elements)
        e.second->update(this->board);
}
```

Programa Cliente



Descripción General

El programa cliente interactua directamente con las bibliotecas **SDL2pp** y **Qt**. Es un modelo de cliente *simple* en el sentido de que practicamente toda la funcionalidad de juego queda encapsulada del lado del servidor. El cliente solo recibe el estado del juego y actualiza lo que debe renderizar en pantalla.

Player

Atributos

```
Protocol protocol;
Socket socket;
```

Las clases protocolo y socket se utilizan para el manejo de la comunicación.

```
player_t faction;
```

Faccion para este jugador.

```
size_t spice;
size_t c_spice;
int energy;
```

Los valores de capacidad y energía se actualizan desde el servidor y se muestran en pantalla.

```
std::map<int, std::unique_ptr<CSelectable>> elements;
```

El Cliente tiene su propio arreglo de elementos seleccionables 'simétrico' al del servidor. Esta no es la clase **Selectable** sino la clase **CSelectable**. Los seleccionables del cliente actualizan sus parámetros en función de los seleccionables en el servidor y renderizarán la imágen correspondiente en función de estos parámetros.

```
std::queue<std::vector<int>> mouse_events;
```

Cola que almacena los elementos del mouse procesados como una instrucción que es pasada por protocolo al Servidor.

```
MouseHandler mouse;
```

Clase 'Helper' para procesar los eventos del mouse

```
TextureHandler & textures;
```

Texturas precargadas al inicializar el programa y guardadas en la clase TextureHandler.

```
AudioPlayer audio;
```

Clase para manejo de audio.

```
CPrinter printer;
```

Clase que se encarga de imprimir los response_t recividos por parte del servidor por pantalla.

```
GameHud hud;
```

Clase para el manejo del Hud.

```
GameMap map;
```

Mapa del juego.

```
SDL2pp::Window& game_window;
SDL2pp::Renderer& game_renderer;
```

Referencias a la ventana y renderer SDL2 previamente inicializados.

```
Camera & camera;
```

Cámara del jugador.

Métodos

```
void play();
```

Loop de juego principal que se encarga de leer los eventos del mouse y procesarlos a modo de instrucción que se envía por protocolo al jugador.

```
void update();
```

Lectura de los valores enviados por el Servidor y actualización de los elementos del Cliente.

```
void render();
```

Renderización de todos los elementos del cliente en la Ventana.

GameMap

Atributos:

```
int dim_x;
int dim_y;
std::vector<std::vector<MapCell>> map_cells;
```

Cada celda del lado del Servidor tiene asociada una **MapCell** que no es más que un contenedor de una textura en forma de Tile. GameMap es un contenedor de estas **MapCell**, y se encarga de actualizarlas y renderizarlas en cada iteración del ciclo de juego.

Métodos:

```
void updateCell(SDL2pp::Renderer& renderer,int pos_x, int pos_y,int spice);
void render(SDL2pp::Renderer& renderer,TextureHandler & texture, int cam_pos_x, int cam_pos_y);
```

CSelectable / CMovable / CStatic

Estas son clases contenedoras de aquellos atributos relevantes existentes del lado del servidor que deberían mostrarse por pantalla. Disponen únicamente de los métodos `update(...)` y `render(...)` . En el primero se pasan a el elemento los valores actualizados recibidos de parte del servidor. En el segundo se les solicita renderizarse por pantalla. Así, a la hora de renderizar cada elemento, la clase Player simplemente recorre cada elemento de su arreglo de punteros a CStatic y le solicita renderizarse.

```
for (auto& e : this->elements)
    e.second->render(this->game_renderer,this->camera.pos_x,this->camera.pos_y);
```

Dado que el programa Cliente no implementa casi ningúna funcionalidad más allá de la de recibir valores del servidor, actualizarlos y renderizarlos, se juzgo innecesario definir heredaras de CMovable o CStatic, ya que el tipo de unidad o edificio solo es relevante para solicitar qué imágen renderizar de parte de el `TextureHandler` . **CMovable** y **CStatic** reciben el tipo de unidad/edificio en su constructor, y luego este es de utilidad para referenciar la textura a renderizar.

```
void CStatic::render(SDL2pp::Renderer& renderer, int cam_pos_x, int cam_pos_y){

int xpos = this->position.x*TILE_SIZE-cam_pos_x;
int ypos = this->position.y*TILE_SIZE-cam_pos_y;
int xdim = this->dim_x*TILE_SIZE;
int ydim = this->dim_y*TILE_SIZE;

renderer.Copy(
    this->textures.getTexture(this->type,(player_t)this->faction)
    .SetColorMod(this->color.r,this->color.g,this->color.b),
    SDL2pp::NullOpt,
    SDL2pp::Rect(xpos,ypos,xdim,ydim)
);

}
```

GameHud

Clase helper que renderiza y actualiza el Hud. Sus atributos son contenedores de botones:

```
std::vector<Button> build_buttons;
std::vector<Button> unit_buttons;
```

Cada boton tiene una ID asignada, de modo durante el loop de juego se haga un click en el Hud en alguno de sus botones, los métodos:

```
int checkUnit(int& x, int& y);
int checkBuild(int& x, int& y);
```

Informen a **CPlayer** en que boton se está clickeando.

CPrinter

Esta es una clase Helper para la impresión de las respuestas enviadas por el servidor. **CPlayer** define el método.

```
void print(std::string toprint,
std::string fontpath,
int x,
int y,
int size,
SDL_Color color,size_t time);
```

Que hace uso de esta clase para renderizar un texto de cierta fuente, tamaño y color en la posición de la pantalla designada.

AudioPlayer

Clase helper utilizada por **CPlayer** que es básicamente un contenedor de efectos de sonido. Sus atributos son:

```
std::map<sfx_t,Mix_Chunk*> sound_effects;
std::map<music_t,Mix_Music*> background_songs;
```

Y estos efectos de sonido son buscados y ejecutados por los métodos.

```
void play(sfx_t sound_effect);
void play(music_t music);
void stopMusic();
```

TextureHandler

Clase Helper en donde se cargan todas las texturas al inicializarse el cliente. Las estructuras se cargan en distintos mapas de la forma:

```
...
std::map<game_status_t,SDL2pp::Texture> game_status_textures;
std::map<building_t,std::map<player_t,SDL2pp::Texture>> building_textures;
std::map<std::string,SDL2pp::Texture> cell_textures;
...
```

Para luego ser referenciadas por métodos:

```
...
SDL2pp::Texture & getGameStatus(game_status_t game_status);
SDL2pp::Texture & getTexture(building_t building, player_t faction);
SDL2pp::Texture & getCell(std::string & type);
...
```

En cualquier parte del programa.

Camera

Clase Helper que designa que porción de la ventana va a renderizarse y mostrarse al cliente. Todos los elementos del juego se renderizarán respecto a la cámara.

Atributos:

```
int pos_x;
int pos_y;
int height;
int width;
int screen_height;
int screen_width;
SDL2pp::Rect view;
```

Métodos:

```
void move(int x,int y);
```