Diseño Preeliminar/Simplificado

Diseño simple que omite requisitos y funcionalidades que se consideran no esenciales en el corto plazo (se pretende que sea extensible como para que estas últimas puedan incorporarse posteriormente con facilidad).
Las cosas que se posponen en este diseño son:

- Detalles del protocolo y la lógica de comunicación entre Cliente y Servidor.
- Mejoras/upgrades para las Unidades y/o edificios
- Tipos de Armamentos y sus Animaciones Particulares (simplemente que cada unidad de ataque tenga un atributo 'poder' o algo así, por ahora)
- Mensajes de voz, sonidos, etc... (son todos event-handlers del cliente para un cambio de estado).
- Diferenciacion de casas (Harkonnen, Atreides, Ordos) para los tipos de Unidades y Edificios.
- El jugador puede demoler sus edificios permitiéndole recuperar una fracción del dinero invertido en su construcción y reducir el gasto de energía si la situación lo amerita.
- Gusanos

Separación de Responsabilidades entre Cliente y Servidor (Idea)

Las clases del cliente **no** tienen comportamiento funcional, solo estado. Su comportamiento se reduce a informar al usuario este estado. Son contenedores para los datos que se estan actualizando en el servidor que deben mostrarse por pantalla, ya sea directamente (ej. la cantidad de Energía o Especia) o indirectamente a través de una imágen o animación (ej. La barra de vida de cada unidad).

- **Ejemplo:** cada **Unidad** que existe en el Servidor también existe en el Cliente, pero en el cliente una unidad solo tendría los atributos **posición** y **LP**, por ejemplo, y un método `Unit::render(Renderer & render)` que asigna una textura/animación a ese valor de LP y a esa posición y las muestra por pantalla.

Las responsabilidades del **Cliente** se limitan a:

- traducir los eventos del mouse o teclado en instrucciones concretas que se pasan en forma de bytestream a través de un socket a el **Servidor**.
- Interpretar la respuesta del servidor y actualizar el estado de las variables.

Clase Jugador

Propiedades (~atributos)

Especia

Parámetro de la clase jugador. Habrá un **size_t especia** que es la cantidad total de especia y que es la sumatoria de lo que tienen las refinerías y una **size_t capacidad** que es la sumatoria de las capacidades de las refinerías. No hay necesidad de estar siempre recalculando. Cuando ocurre un evento que deba modificar este valor (ej. se destruye una nueva refinería, o se agrega especia a una refinería), este valor se actualiza automáticamente. Lo mismo para la energía y las trampas de viento.

```
size_t especia;
size_t capacidad;
```

Energía

Parámetro de la clase jugador. Habrá una **int energía** que es la cantidad total de energía y que no debe superar la capacidad y que se calcula como la sumatoria de la energía para todas las trampas de viento. No hay necesidad de estar siempre recalculando. Cuando ocurre un evento que deba modificar este valor (ej. se crea o destruye una nueva trampa de viento), este valor se actualiza automáticamente. Lo mismo para la capacidad y las refinerías.

```
int energia;
```

Eficiencia

Coeficiente que afecta (hasta donde entiendo), la velocidad con la que se construyen nuevas unidades y edificios. Lo ponemos en 1 por defecto y lo reducimos proporcionalmente a la **deuda energética** (el valor de la energía cuando esta es negativa).

```
double eficiencia;
```

Unidades y Edificios

El jugador guarda las unidades y edificios en un arreglo o en un mapa. Corresponden a la clase **Jugador** y no al **Tablero** por que este último es un recurso compartido de todos los jugadores mientras que cada unidad y edificio corresponde a un jugador. El **Tablero** puede ser un **Singleton** compartido entre todos los jugadores cuyos métodos son estáticos. O bién cada jugador puede tener

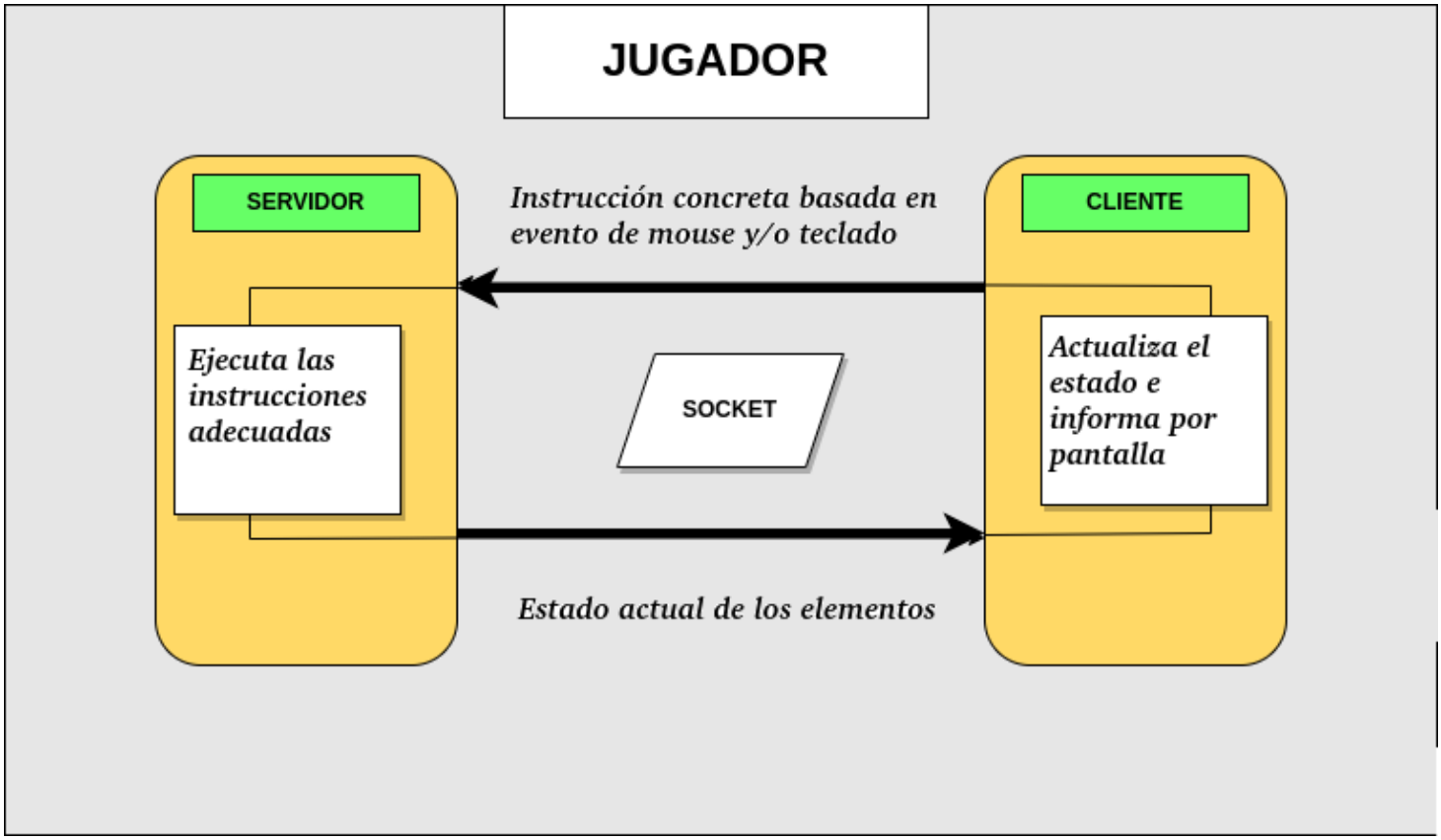
una referencia al mismo tablero. Probablemente esta clase **Tablero** sea el recurso a proteger por el **Monitor**.

Comportamientos (~métodos):

El **Jugador** en el **Servidor** tiene acceso a un **Socket** que lo conecta con el **Jugador** del **Cliente**. Este último comunica los eventos de mouse y teclado disparados por el usuario en forma de instrucción. Por ejemplo:

- **Crear Edificio**
- **Crear Unidad**
- **Selección del Mapa**
- **Click Izquierdo en el Mapa**
- **Click Derecho en el Mapa**
- **Mouse Levantado (default)**

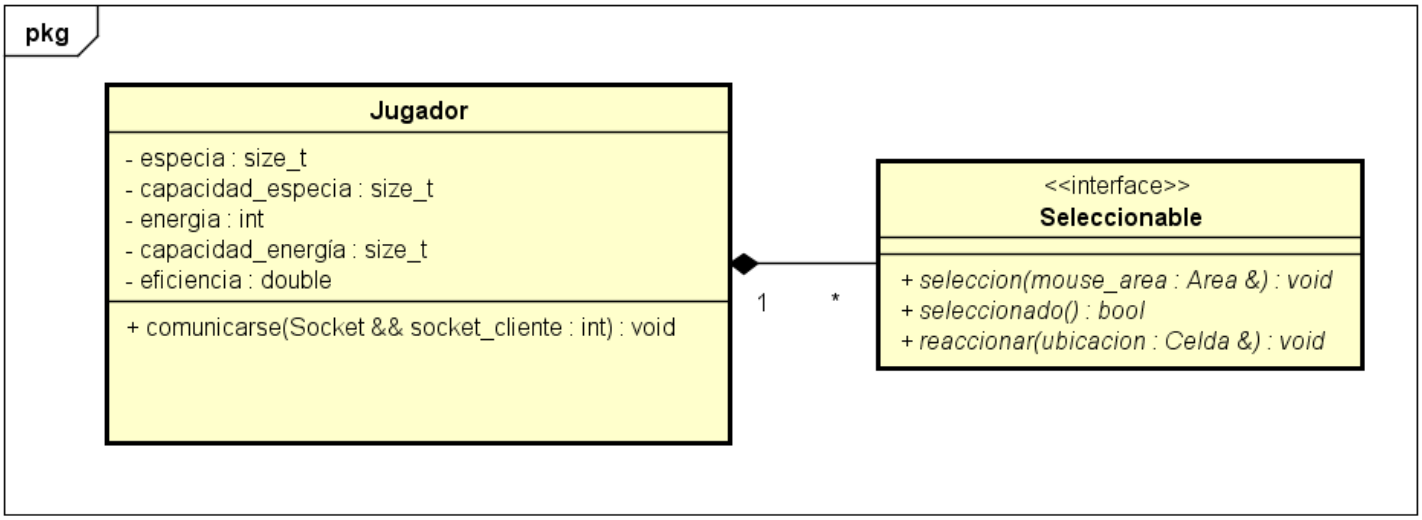
  .



Para el manejo de la respuesta a estos eventos por parte del **Jugador** en el servidor, conviene definir una Interfaz **Seleccionable** que es implementada tanto por las unidades como los edificios, y que tiene los métodos:

```
void seleccion(Area & mouse_area) // marca como seleccionado si esta incluido en el area
bool seleccionado() // retorna si esta seleccionado actualmente
void reaccionar(Celda & ubicacion) // reaccionar a lo que se clickeo
```

El objetivo de definir una interfaz que englobe a todo tipo de unidad es que permite que el **Jugador** pueda almacenar todos los elementos del mapa en una misma colección y delegar polimórficamente la implementación de las respuestas a los eventos del mouse a cada uno.



Adicionalmente, podemos incorporar otros métodos como:

```
Descripcion descripcion()
std::vector<Posiciones> obtenerPosiciones()
```

Donde **Descripcion** es una estructura cuyos campos dan información de el elemento **Seleccionable**. Así, podemos construir una seudo **base de datos** de nuestros elementos seleccionables. Cuando pasamos información al cliente, lo que hacemos es obtener el tamaño de nuestro arreglo de seleccionables y lo pasamos por el **Socket**. Luego enviamos una a una las descripciones para cada unidad.

```
Sv ---...<pos_y2><pos_x2><LP2><ID2><pos_y1><pos_x1><LP1><ID1><total>---> Cl
```

El cliente hara tantas lecturas como elementos haya del otro lado, procesará estas descripciones y actualizará estos valores para sus clases, que se ocuparan de mostrar estos cambios por pantalla.

```
for (size_t i = 0; i < total_read; i++)
{
    protocol.read(&ID,&LP,&pos_x,&pos_y)
    elements.at(ID).update(LP,pos_x,pos_y).render(renderer)
}
```

A continuación, especificamos como el servidor debe manejarse la respuesta a cada instrucción del cliente.

Caso crear edificio

Cuando el mouse del usuario hace click en la posición de la pantalla correspondiente a la creación de un edificio, el Cliente dispara un evento que solicita la posición de el mapa donde este quiere colocarse. Al clickear, se dispara un evento que envia al Servidor el tipo de unidad y la posición donde se desea colocar.

```
Sv <------<tipo>(8)<pos_x>(16)<pos_y>(16)-----Cl
```

Este responde afirmativamente o negativamente segun la unidad pueda crearse o no (debe estar creandose sobre celdas tipo **Roca** y estas deben además estar desocupadas).

```
Sv -------<status>(8)------->Cl
```

En caso afirmativo el servidor marca las celdas como ocupadas y guarda la unidad entre sus Seleccionables. El cliente por su parte agrega esta unidad a sus Seleccionables, sustrayendo la cantidad de especia y energía correspondiente, y la llama a renderizarse en la pantalla.
En caso negativo, ninguno de los dos hace nada.

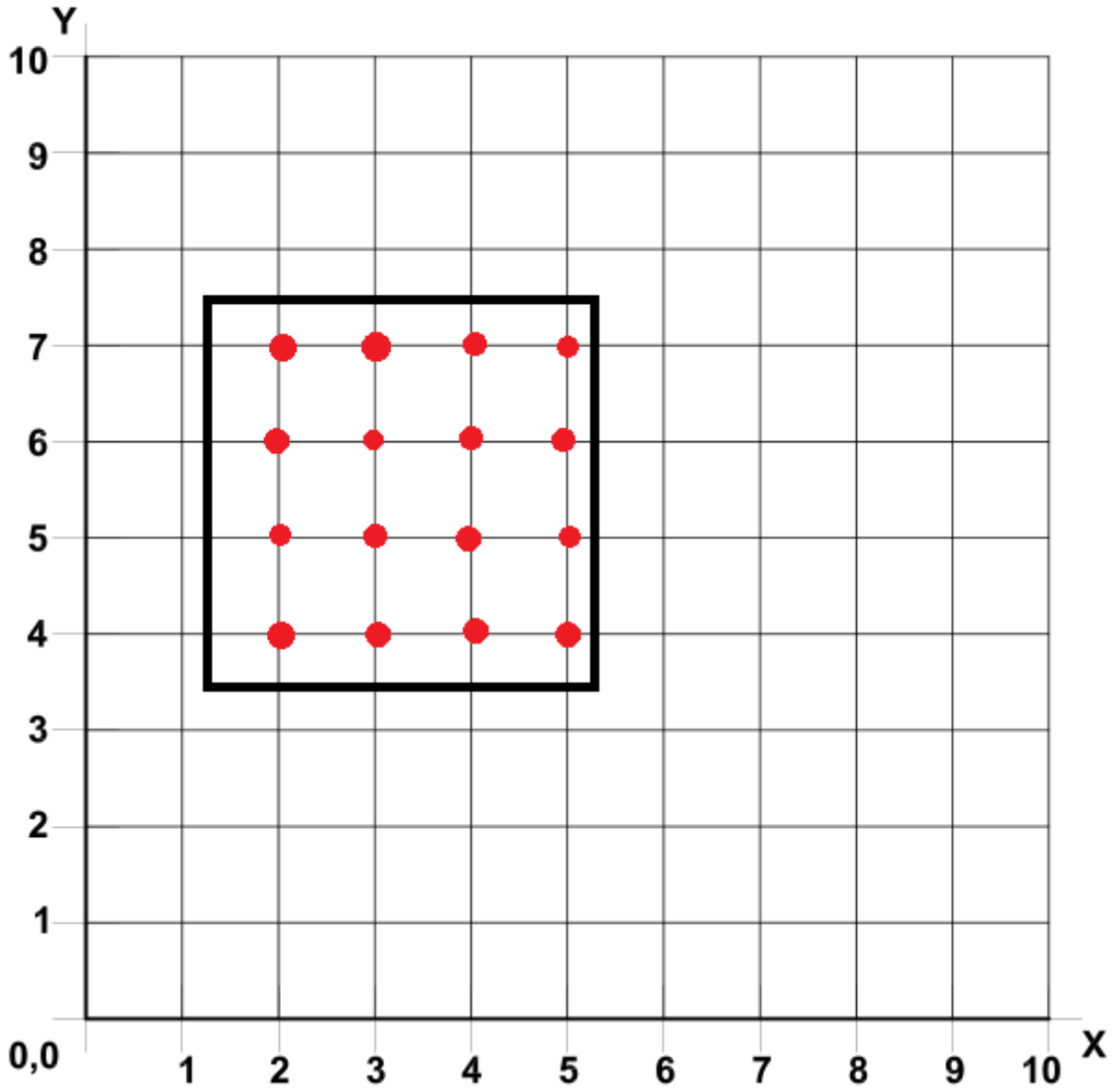
Caso crear unidad

Cuando el mouse del usuario hace click en la posición de la pantalla correspondiente a la creación de una unidad...

1. ¿Cómo debe ser la lógica de creación de unidades? ¿Salen de el mismo edificio?

Caso Seleccion:

El cliente se encarga de traducir el evento del mouse a una estructura **Area**, que es esencialmente un contenedor de los puntos $\{x_{min}, x_{max}, y_{min}, y_{max}\}$ del rectángulo formado por el mouse.



El jugador consulta la posición de cada uno de sus edificios y unidades y verifica que esten contenidos dentro del área o no. Aquellos que lo estén se marcan como seleccionados.

```
for (auto & e : elements){
    e.second.unselect();
    if(e.second.isWithin(selection))
        e.second.select();
}
```

Caso Click Izquierdo:

Se de-seleccionan todos los edificios y unidades y se seleccionan solo los que esten en esa posición.

Caso Click Derecho:

El jugador recorre los elementos (Unidades y Edificios) marcados como *seleccionados* y llama para cada uno el método:

```
reaccionar(Celda & posicion);
```

Este es un método polimórfico que se redefine para cada tipo de la clase **Unidad** y **Edificio**. Esto es así por que cada unidad reacciona de forma diferente de acuerdo al contenido de la información provista por la **Celda**.
Alternativa: La lógica de selección se maneja del lado del cliente. Al haber un click derecho el cliente envía los seleccionados al Servidor y este maneja las respuestas.

información: *¿Puede desplazarse hacia ella?*
caso: La celda ***no*** es de tipo **Precipicio**.
respuesta: Todas las unidades llaman al método:

```
moverse(Posicion & destino)
```

para la posición de la Celda.
Si es de tipo **Precipicio**, las unidades no hacen nada.

información: *¿Puede extraerse especia?*
caso: La celda es de tipo **Arena** y tiene especia para extraer.
respuesta: las unidades de la clase **Cosechadora** no implementarán el método para moverse y en su lugar llamarán a:

```
cosechar(Posicion & destino)
```

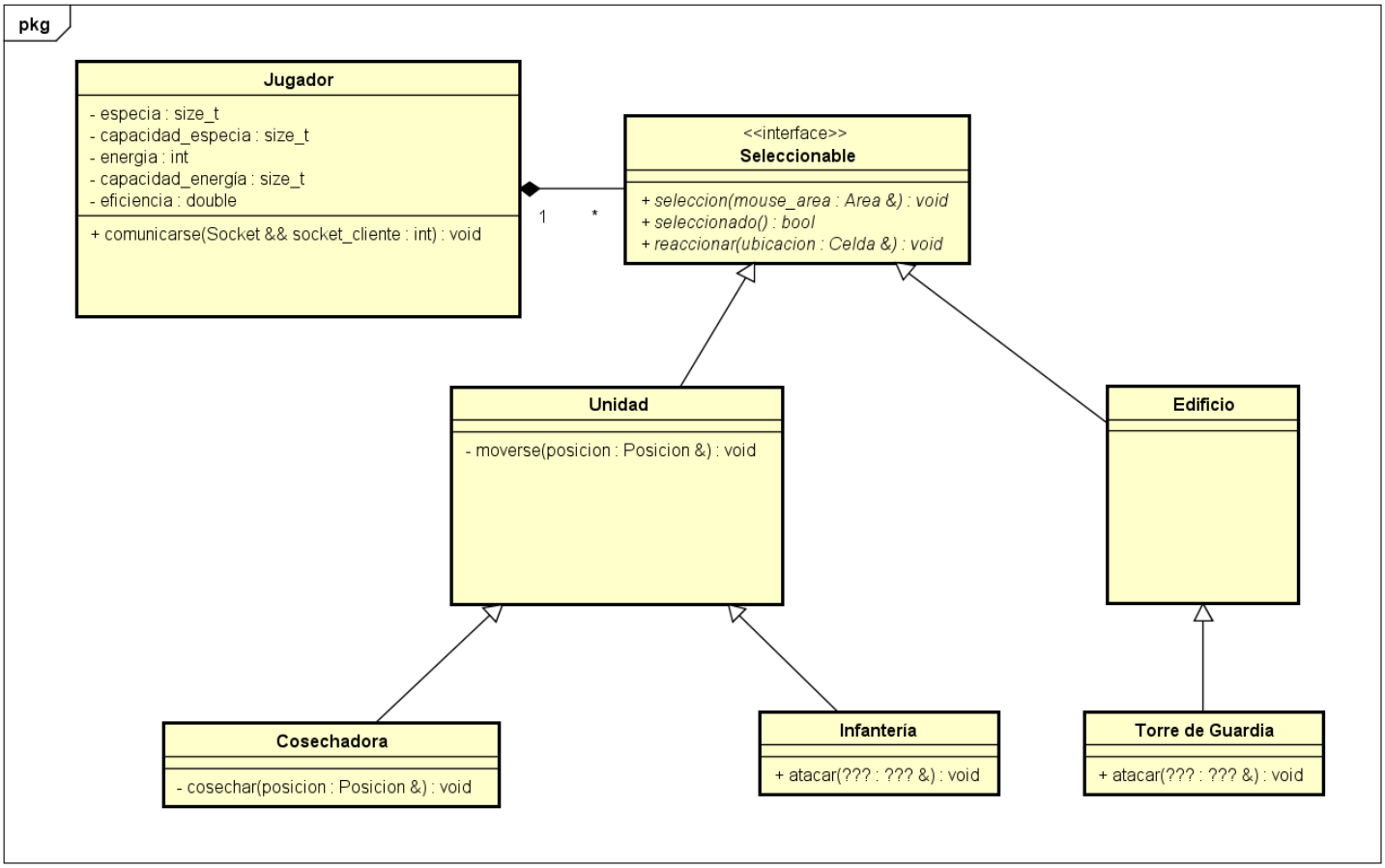
el resto de las unidades movibles no hacen este chequeo y solo se mueven a esta posición.
Adicionalmente, el metodo `reaccionar()` de las refinerías puede, por ejemplo, seleccionar a esta posición como la posición por defecto a la que se moveran sus cosechadoras al crearse.

información: *¿Puede atacarse?*
caso: La celda esta ocupada por una unidad enemiga.

respuesta: Las unidades o edificios que sean de combate no implementan el método para moverse y en su lugar llaman a.

```
atacar(??? & ???);
```

2. ¿Cómo debe ser la lógica de combate? ¿Cómo puede una unidad del Jugador 1 hacer un seguimiento de otra del Jugador 2 si estan en threads diferentes?



Tablero

Contenedor de celdas (básicamente).

Propiedades (~ atributos):

```
std::vector<Celda> celdas;
size_t dim_x;
size_t dim_y;
```

Comportamientos (~ métodos):

```
bool construir(Posicion & posicion,size_t u_dim_x, size_t u_dim_y)
```

Recibe una **Posicion** para la unidad y sus dimensiones. A partir de estas y la posición pasada deriva un conjunto de Posiciones donde se pretende construir. Consulta a las celdas que estan en esas posiciones si se puede construir en ellas y devuelve **true** solo en caso de que todas le respondan afirmativamente, marcando a aquellas celdas como **ocupadas**. El jugador por su parte recibe este **true** y agrega esta unidad al contenedor correspondiente. De lo contrario (caso **false**), la descarta y notifica de esto al cliente (este puede reaccionar ya sea con un mensaje al usuario,un audio que le diga que no puede construirse en esa area, una imágen de la unidad pero en rojito, o todo al mismo tiempo).

```
bool obtenerCelda(Posicion & posicion)
```

devuelve la celda en esa posición. Este método sera utilizado por el **Jugador** al recibir una posición por parte del mouse del cliente. Obtendrá la celda en esa posición y se la pasará a todas sus unidades seleccionadas pidiendoles reaccionar.

Celdas

5 tipos de terreno → 5 tipos de celda

3. ¿Cómo se maneja el tema de la visibilidad de las celdas del tablero si esta debe ser diferente para distintos jugadores?

Para diferenciar los tipos, definimos un atributo de la clase **Celda** que es en principio una interface implementada por cada tipo (esto es, usamos el patrón **Strategy** o **State**). Esta interface puede llamarse **Terreno** o si se quiere ser estricto con la nomenclatura podemos llamarla **Reaccionable**. Esta tendria métodos:

```
bool esAtravesable();
```

para la **Roca**, en caso de que se haya construido o no un edificio sobre ella. El **Precipicio** es no atravesable por defecto y los otros 3 tipos restantes son atravesables por defecto.

```
bool esExtractible();
```

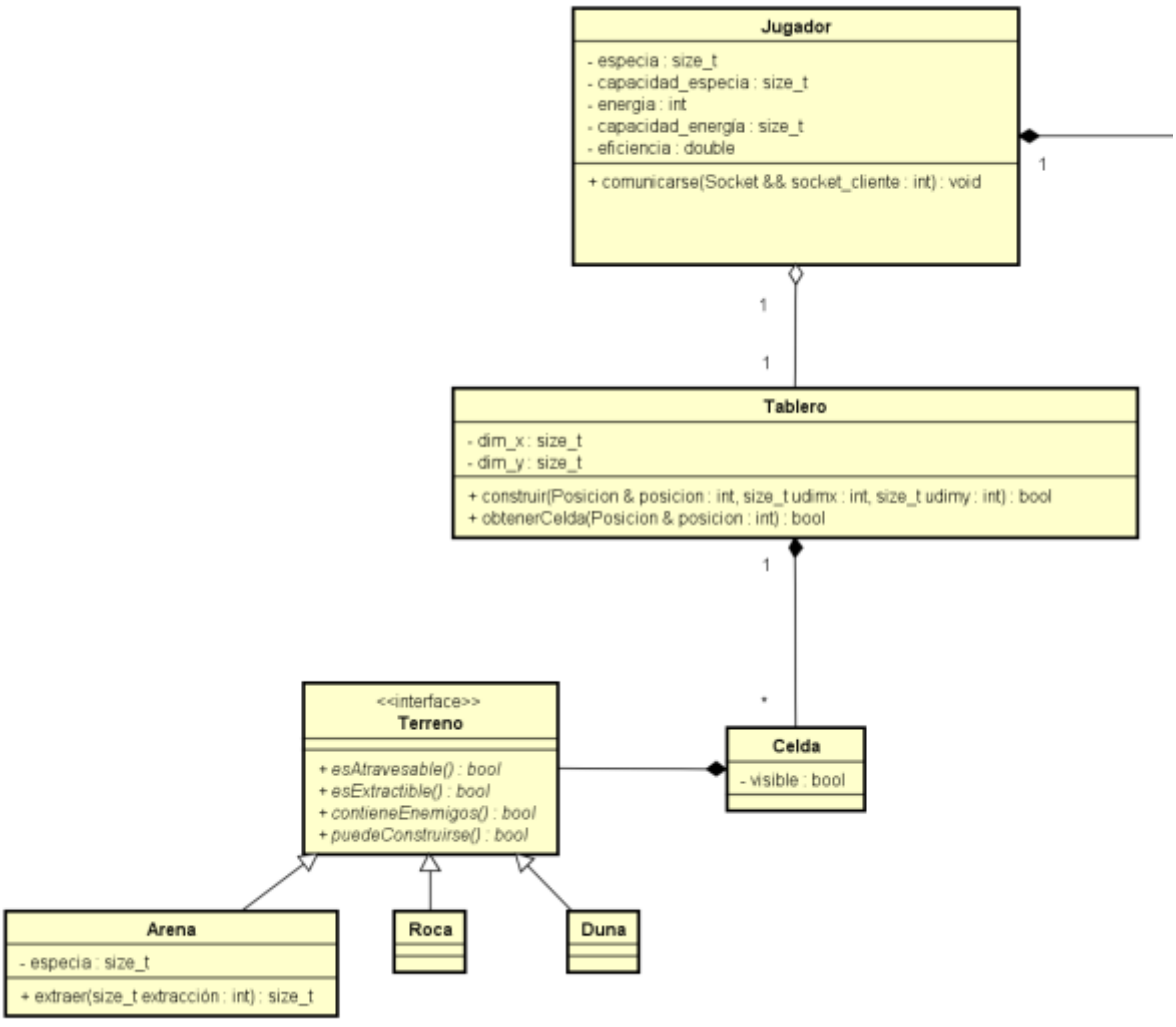
devolvera **true** solo para el tipo **Arena** y en la medida en que la cantidad de especia sea mayor a cero.

```
bool puedeConstruirse();
```

Solo sera verdadero para la clase **Roca** y en la medida en que sea atravesable.

```
bool contieneEnemigos();
```

O algo así, no se cómo sería la lógica de combate



Observaciones:

1. Cualquier funcionalidad asociada a este tipo es solo relevante desde el punto de vista del **Servidor**. El **Cliente** también puede aplicar el patron Strategy pero este solo va a definir el tipo de textura que se va a poner en la renderización.
2. La **especia** es en principio un **size_t** para las celdas de tipo **Arena**.

Unidades

Cosas en común

Propiedades (~ atributos) comunes:

1. **Vida:**

   ```
   size_t LP
   ```

2. **Velocidad:**

   ```
   size_t speed
   ```

3. **Armadura:**

   ```
   size_t armor
   ```

Comportamientos (~ métodos) comunes:

1. **Proceso de selección con el Mouse:**

Discutido arriba

2. **Proceso de movimiento:**

   ```
   moverse(Posicion & destino)
   ```

Ejecuta el algoritmo **A-star** desde su posición actual hasta la indicada. Este algoritmo devuelve, como sabemos, un arreglo de puntos. La unidad recorre estos puntos actualizando su posición cada cierto tiempo que va a estar determinado por su atributo **velocidad**. Cada vez que su posición cambia de estado, dispara un **event handler** que notifica al cliente (ej. le envia por un socket su posición inicial y su posición actual). El cliente dispara la animación correspondiente. Por ejemplo, si fue de (1,1) a (1,2), dispara la animación *arriba*; si fue de (1,1) a (2,2), dispara la animación **diagonal derecha arriba** y así.

Infantería

De momento, solo hacer que varien sus 3 atributos

Trike

De momento, solo hacer que varien sus 3 atributos

Tanque

De momento, solo hacer que varien sus 3 atributos

Cosechadoras

Tipo de unidad que además de los atributos comunes de la clase **Unidad** tiene el atributo adicional **size_t capacidad** y puede o no también tener un atributo **size_t eficiencia** que es un coeficiente que define la velocidad de extracción.

Proceso de extracción:
a.
b. Cuando la **capacidad** se llena y la cosechadora termina de extraer, esta informa al servidor pasandole su Posición actual. El servidor le devuelve la posición de la refinería más cercana (consulta a sus refinerías sus posiciones y una vez obtenidas utiliza el algoritmo **Position find_nearest(Position origin, std:: vector<Position> destinations)** y le pasa la posición obtenida a la cosechadora, la cual utiliza el algoritmo **A-star** para caminar hacia esa refinería)

Problema: *¿Cómo implementar esto sin tener que consultar tipos?*

Solucion preeliminar: *Recorrer las unidades llamando al método* `Descripcion descripcion()` *y compare* `desc.nombre == "refineria"` *y para las que lo verifiquen llamar a* `.obtenerPosiciones()` *.*

Refinerias

Tipo de edificio que tiene asociada una **size_t especia** con un límite prefijado de 5000 que podemos o no agregar como atributo.

Edificios

Cosas en común

Propiedades (~atributos) comunes:

1. **Puntos**: Resistencia de la estructura
2. **Energia**: Valor energético que consume
3. **Costo**: En especia, para construirlo
4. **rango:** distancia (expresada en celdas/posiciones) a la redonda donde otro edificio **no** puede construirse.

5. **Visibilidad**: cuantas celdas se hacen visibles para un rango alrededor de esa estructura.
6. **dim_x:** dimensión en X
7. **dim_y:** dimensión en Y
8. **posición** de la celda en la esquina superior izquierda

Observaciones:

1. Los edificios solo pueden construirse sobre terreno rocoso (celdas de tipo **Roca**)

Comportamientos (~métodos) comunes:

1. **selección con el Mouse:**
   Tengo que mirar gameplays para saber que propiedades tienen al seleccionarlos
2. **construcción:**
   Al clickear, se retornan todas las posiciones que ocuparía la construcción. El tablero va a preguntar a todas las celdas correspondientes si son del tipo **Roca** y si estan desocupadas y solo en ese caso va a aprobar la construcción.
3. **creación de Unidades:**
   No me queda claro de donde tienen que salir las unidades

Trampa de Aire

Refinería

Cuartel