```
#ifndef Z_TPGRUPAL_THREAD_H
#define Z_TPGRUPAL_THREAD_H


#include <thread>

class Thread {
protected:
    std::thread thread;

public:
    void start();

    virtual void run() = 0;

    void join();
};


#endif //Z_TPGRUPAL_THREAD_H
```

```
#include "Thread.h"

void Thread::start() {
    thread = std::thread(&Thread::run, this);
}

void Thread::join() {
    this->thread.join();
}
```

```cpp
#include "socket.h"
#include "messenger.h"
#include <mutex>
#include <iostream>
#include "clientReceiverTest.h"

int main() {
    Socket s("127.0.0.1", 8000);
    Messenger m(s);
    std::mutex mutex;
    clientReceiverTest client(m,mutex);
    client.start();

    while(true) {
        std::cout << "Enter message: " << std::endl << ">> ";
        std::string msg;
        std::getline(std::cin, msg);
        m.sendMessage(msg);
    }
}
```

```cpp
#ifndef TP3TALLER_COMMON_SOCKET_H
#define TP3TALLER_COMMON_SOCKET_H

#include <string>
#include "socketError.h"

#define LISTEN_BACKLOG 10 // Amt. of connections to have in the accept backlog

// Socket class. Wraps functionality of glibc's socket functions.
class Socket {
    int fd;

public:
    /* Server constructor. Creates a socket, binds and listens to the specified
     * port. */
    explicit Socket(int port);

    /* Client constructor. Creates a socket and attempts to connect to the
     * specified address/port. Raises exception if the connection fails. */
    Socket(const char *addr, int port);

    ~Socket();

    // Returns a new client.
    Socket accept_client();

    // Sends/recieves len bytes of data
    ssize_t send(const char *msg, unsigned int len);
    ssize_t receive(char *dest, size_t len);

    // Wrapper for socket shutdown/close
    void shutdown();
    void close(); // Effectively makes the socket object useless

    bool is_valid();
    // Move constructor
    Socket(Socket∧ other);

private:
    Socket();

    Socket(Socket&) = delete;
    void operator=(Socket&) = delete;
};

#endif //TP3TALLER_COMMON_SOCKET_H
```

```
//
// Created by rodian on 22/05/17.
//

#ifndef Z_TPGRUPAL_SOCKETERROR_H
#define Z_TPGRUPAL_SOCKETERROR_H

#include <iostream>
#include <cstring>

////////////////////////
// SocketError Class to warn
// of an error on the socket
////////////////////////

class SocketError : public std::exception {
private:
    char buffer[124];

public:
    explicit SocketError(const char* message, ...) noexcept;

    // Returns the error message
    virtual const char* what() const noexcept;
};

#endif //Z_TPGRUPAL_SOCKETERROR_H
```

```
//
// Created by rodian on 22/05/17.
//

#include "socketError.h"

SocketError::SocketError(const char *message, ...) noexcept {
    strncpy(buffer, message, strlen(message));
}

const char* SocketError::what() const noexcept {
    return buffer;
}
```

```cpp
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <cstring>
#include <utility>
#include <iostream>
#include <string>
#include "socket.h"

Socket::Socket(int port) {
    fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) {
        throw SocketError("Couldn’t create a socket!\n");
    }

    struct sockaddr_in srv;
    memset(&srv, 0, sizeof(srv));
    srv.sin_family = AF_INET;
    srv.sin_addr.s_addr = htonl(INADDR_ANY);
    srv.sin_port = htons((uint16_t) port);

    int error = bind(fd, (struct sockaddr *) &srv, sizeof(srv));
    if (error) {
        throw SocketError("Error binding socket on creation! "
                                      "Most likely port already in use");
    }

    listen(fd, LISTEN_BACKLOG);
}

Socket::Socket(const char *addr, int port) {
    fd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in srv;
    srv.sin_family = AF_INET;
    srv.sin_port = htons((uint16_t)port);
    srv.sin_addr.s_addr = inet_addr(addr);

    socklen_t len = (socklen_t)sizeof(struct sockaddr);
    int error = connect(fd, (struct sockaddr *) &srv, len);

    if (error) {
        throw SocketError("Error connecting to server!");
    }
}


Socket Socket::accept_client() {
    struct sockaddr_in client;
    socklen_t clilen = (socklen_t) sizeof(struct sockaddr_in);

    int client_fd = accept(fd, (struct sockaddr *) &client, &clilen);
    if (client_fd < 0 ∨ fd < 0) {
        throw SocketError("Socket disconnected");
    }

    Socket client_socket;
    client_socket.fd = client_fd;
    return client_socket;
}

Socket::Socket() {
}

Socket::Socket(Socket∧ other) {
    fd = other.fd;
```

```cpp
    other.fd = −1; // "Deactivates" other
}

ssize_t Socket::send(const char *msg, unsigned int len) {
    size_t total_bytes = 0;
    ssize_t sent = 1;

    // Sends msg until it’s complete OR socket_send returns 0 (connection
    // closed)
    while (total_bytes < len ∧ sent) {
        sent = ::send(fd, msg + total_bytes, len − total_bytes,
                      MSG_NOSIGNAL);
        if (sent < 0) {
            return −1;
        }
        total_bytes += sent;
    }

    return total_bytes;
}

ssize_t Socket::receive(char *dest, size_t len) {
    ssize_t received = 1;
    size_t total_bytes = 0;

    // Writes to dest until it’s complete OR socket_recv returns 0 (connection
    // closed)
    while (total_bytes < len ∧ received) {
        received = recv(fd, dest + total_bytes, len − total_bytes,
                        MSG_NOSIGNAL);
        if (received < 0) {
            return −1;
        }
        total_bytes += received;
    }
    return total_bytes;
}


Socket::~Socket() {
    if (fd > 0) {
        close();
    }
}

void Socket::shutdown() {
    ::shutdown(fd, SHUT_RDWR);
    fd = −1;
}

void Socket::close() {
    ::close(fd);
    fd = −1;
}

bool Socket::is_valid() {
    return fd > 0;
}
```

```cpp
//
// Created by rodian on 22/05/17.
//

#ifndef Z_TPGRUPAL_MESSENGER_H
#define Z_TPGRUPAL_MESSENGER_H

#include "socket.h"
#include <iostream>
#include <string>
////////////////////////
// Messenger Class meant to use sockets
// to send messages between Client and
// Server using a specific protocol.
// Send lenght of message first, then the message.
////////////////////////
class Messenger{
private:
    Socket socket;
    bool connected;

public:
    // Recieves a unique socket to send
    // and recieves messages from
    explicit Messenger(Socket& socket);

    // Recieves a Message from the remote
    // connected socket.
    // Returns the message on a string
    std::string recieveMessage();

    // Sends a message to the remote socket
    // Recieves the message on a string
    void sendMessage(const std::string &message);

    // Shuts down the socket for read and write
    void shutdown();

    // If the sockets are still connected returns true
    // otherwise false.
    bool isConnected();

    ~Messenger();
};


#endif //Z_TPGRUPAL_MESSENGER_H
```

```cpp
#include <netinet/in.h>
#include <string>
#include "socket.h"
#include "messenger.h"

Messenger::Messenger(Socket& socket) : socket(std::move(socket)),
                                       connected(true) {}

#define MSG_SIZE 1024

std::string Messenger::recieveMessage() {
    // Receive length first, then the message
    uint32_t len = 0;
    socket.receive((char*) &len, sizeof(len));
    len = ntohl(len);
    char* buf = new char[len];
    ssize_t sent = socket.receive(buf, len);
    if (sent ≤ 0) {
        throw(SocketError("Socket closed"));
    }
    std::string result(buf);

    delete[] buf;
    return result;
}

void Messenger::sendMessage(const std::string &message) {
    if (this→connected) {
        uint32_t len = (uint32_t) message.size() + 1;
        // Send length first, then the message
        uint32_t network_len = htonl(len);
        socket.send((char *) &network_len, sizeof(network_len));
        socket.send(message.c_str(), len);
    }
}

void Messenger::shutdown() {
    socket.shutdown();
}

bool Messenger::isConnected() {
    return socket.is_valid();
}

Messenger::~Messenger() {
}
```

jun 27, 17 14:46 **clientReceiverTest.h** Page 1/1

```
//
// Created by rodian on 15/06/17.
//

#ifndef Z_TPGRUPAL_CLIENTRECEIVERTEST_H
#define Z_TPGRUPAL_CLIENTRECEIVERTEST_H


#include <mutex>
#include "messenger.h"
#include "Thread.h"

class clientReceiverTest: public Thread {
private:
    std::mutex &m;
    Messenger &messenger;
    bool listen;
public:
    clientReceiverTest(Messenger &messenger, std::mutex& m);

    void run();
};


#endif //Z_TPGRUPAL_CLIENTRECEIVERTEST_H
```

jun 27, 17 14:46 **clientReceiverTest.cpp** Page 1/1

```
//
// Created by rodian on 15/06/17.
//

#include "clientReceiverTest.h"

clientReceiverTest::clientReceiverTest(Messenger &messenger, std::mutex &m) :
     m(m),messenger(messenger), listen(true) {}

void clientReceiverTest::run() {
    while (listen) {
        std::string msg = messenger.recieveMessage();

//        m.lock();
        std::cerr << "Answer: " << msg << std::endl;
//        m.unlock();
    }
}
```

```
//
// Created by rodian on 21/05/17.
//

#ifndef Z_TPGRUPAL_WEAPON_H
#define Z_TPGRUPAL_WEAPON_H

#include <iostream>
#include <vector>
#include "size.h"
#include "Occupant.h"
#include "bullet.h"

class Weapon {
private:
    std::string type;
    int damage, w_speed;
    bool explosive;
    Size w_size;
    Bullet bullet;

public:
    Weapon(std::string& type, int damage, int w_speed, bool explosive,
                                                     Size w_size);

    void setNewTarget(Occupant* target);

    Bullet* shotTarget(Occupant* target);

    bool isTheAttackExplosive();

    std::vector<Position>& getBulletRoad();

    Size getBulletSize() const;

    void recalculateRoadToTarget();

    void movePosition(int x,int y);
};


#endif //Z_TPGRUPAL_WEAPON_H
```

```
//
// Created by rodian on 21/05/17.
//

#include "weapon.h"

Weapon::Weapon(std::string& type, int dmg, int w_speed,
               bool explosive, Size w_size) : type(type), damage(dmg),
                w_speed(w_speed), explosive(explosive), w_size(w_size),
                bullet(type,dmg,w_speed,w_size) {}

Bullet* Weapon::shotTarget(Occupant* target) {
    Bullet* shoted_bullet = new Bullet(type,damage,w_speed,w_size,target);
    shoted_bullet→shotTarget(target);
    return shoted_bullet;
}

bool Weapon::isTheAttackExplosive() {
    return explosive;
}

void Weapon::setNewTarget(Occupant* target) {
    this→bullet.shotTarget(target);
}

std::vector<Position>& Weapon::getBulletRoad() {
    return this→bullet.getRoad();
}

Size Weapon::getBulletSize() const {
    return this→w_size;
}

void Weapon::recalculateRoadToTarget() {
    this→bullet.calculateRoadToTarget();
}

void Weapon::movePosition(int x, int y) {
    w_size.moveTo(x,y);
    this→bullet.setStartLocation(x,y);
}

```

```cpp
//
// Created by rodian on 22/05/17.
//

#ifndef Z_TPGRUPAL_UNITMOLD_H
#define Z_TPGRUPAL_UNITMOLD_H


#include "unit.h"

class UnitMold {
private:
    int tec_level, life, range, width, height, unit_speed, fire_rate,
            creation_time, creation_quantity;
    std::string type,weapon_type;

public:
    UnitMold(int tec_level, int life, int range, int width, int height,
             int unit_speed, int fire_rate, int creation_time,
             int creation_quantity, std::string &type,
             std::string &weapon_type);

    Unit *createUnit(int id, Size u_size, Map& map,
                     Weapon &weapon);

    // Returns the technology level needed to create Tough unit
    int getTechnologyLevel() const;

    // Returns creation time in seconds
    int getCreationTime();

    // Returns the type of unit that this mold creates
    std::string getTypeOfUnit() const;

    std::string getWeaponType() const;

    Size getUnitSize();

    int getCreationQuantity() const;

    int getFireRate() const;

    int getLife() const;


};


#endif //Z_TPGRUPAL_UNITMOLD_H
```

```cpp
//
// Created by rodian on 22/05/17.
//

#include "unitMold.h"

UnitMold::UnitMold(int tec_level, int life, int range, int width, int height,
                   int unit_speed, int fire_rate, int time, int creation_quantit
y,
                   std::string &type, std::string &weapon_type) :
tec_level(tec_level), life(life), range(range), width(width), height(height),
unit_speed(unit_speed), fire_rate(fire_rate), creation_time(time),
creation_quantity(creation_quantity), type(type),weapon_type(weapon_type) {}

Unit *UnitMold::createUnit(int id, Size u_size, Map& map,
                           Weapon &weapon) {
    Position u_pos = u_size.getPosition();
    Compass* compass = new Compass(map, u_size,id,unit_speed);
    // get closest valid position from fabric
    Position valid_pos = compass->getAValidPositionForDestiny(u_pos);
    u_size.moveTo(valid_pos.getX(),valid_pos.getY());
    int x_range = valid_pos.getX() - range;
    int y_range = valid_pos.getY() - range;
    int w_range = range * 2 + width;
    int h_range = range * 2 + height;
    Size unit_range(x_range,y_range,w_range,h_range);


    Unit* new_unit = new Unit(id, life, type, unit_speed, u_size, unit_range,
                              compass, weapon, fire_rate);
    return new_unit;
}

int UnitMold::getTechnologyLevel() const {
    return tec_level;
}

int UnitMold::getCreationTime() {
    return creation_time;
}

std::string UnitMold::getTypeOfUnit() const {
    return type;
}

Size UnitMold::getUnitSize() {
    return Size(0, 0, width, height);
}

int UnitMold::getCreationQuantity() const {
    return creation_quantity;
}

std::string UnitMold::getWeaponType() const {
    return this->weapon_type;
}

int UnitMold::getFireRate() const {
    return fire_rate;
}

int UnitMold::getLife() const {
    return life;
}
```

```
//
// Created by rodian on 18/05/17.
//

#ifndef Z_TPGRUPAL_UNIT_H
#define Z_TPGRUPAL_UNIT_H

#include "Occupant.h"
#include "compass.h"
#include "teamable.h"
#include "bullet.h"
#include "weapon.h"

#define ATKSTATE "atk"
#define MOVESTATE "mv"
#define STANDINGSTATE "std"
#define GRABBINGSTATE "grb"
#define FLAGTYPE "flag"
#define GRUNTTYPE "grunt"


class Unit: public Occupant {
private:
    Compass* compass;
    Weapon weapon;
    int unit_speed, fire_rate, fire_count;
    std::string state,action;
    Size range, grab_range;
    std::vector<Position> road;
    Occupant* target;
    Teamable* grab_target;
    std::vector<Bullet*> bullets;
    bool got_target, mount_vehicule;

public:
    Unit(int id, int life, std::string type, int unit_speed, Size size,
             Size range, Compass* compass, Weapon &weapon, int fire_rate);

    // This method is the one who makes the unit make the action that he must
    // do depending on his state
    void makeAction();

    // Calculates the minimum road to destiny (x,y)
    void calculateRoadTo(int x, int y);

    // Returns the current position of the unit
    Position getCurrentPosition() const;

    // Returns "atk" if is attacking, "mv" if is moving, "std" if
    // is standing still
    std::string getActionState() const;

    void grab();

    void setTargetToAttack(Occupant* target);

    void setTargetToGrab(Teamable* object, std::string type);

    bool doYouHaveAnyBullets();

    std::vector<Bullet*> collectBullets();

    bool checkIfTargetIsOnRange();

    // The bullet will hit if there is no Occupant in the middle.
    // except for bridges
```

```
    bool checkIfBulletWillHit(std::vector<Position>& b_road, Size& b_size);

    void getOnRangeOf(int x, int y);

    bool checkIfAlreadyOnMyWay(int x, int y);

    Size getNextPosition(int steps);

    void recalculateMyStartPosition();

    ~Unit();

private:
    // Indicates the Unit to make the next step on the road.
    // Make sure of use the calculateRoadTo method before this one.
    void move();

    // this method creates the bullets to attack a certain target
    void attack();

    bool onRangeToGrabTarget();
};


#endif //Z_TPGRUPAL_UNIT_H
```

```
//
// Created by rodian on 18/05/17.
//

#include "unit.h"
#define NEUTRAL "Neutral"
Unit::Unit(int id, int life, std::string type, int unit_speed, Size size,
           Size range, Compass* compass, Weapon &weapon, int fire_rate) :
        Occupant(id, life, type, size), compass(compass), weapon(weapon),
        unit_speed(unit_speed),fire_rate(fire_rate),fire_count(0),
        state(STANDINGSTATE),action(STANDINGSTATE), range(range),
        grab_range(size.getPosition().getX() − 1,size.getPosition().getY() − 1,
                   size.getWidth() + 2, size.getHeight() + 2),
        target(this),grab_target(this),got_target(false),
        mount_vehicule(false){
    compass→changeUnitId(id);
    compass→buildNodeMap();
}

void Unit::makeAction() {
    if (this→state ≡ STANDINGSTATE) {
        if (this→team ≠ NEUTRAL) {
            if (¬got_target) {
                // Check for enemies around you. If so, state = ATKSTATE
                this→changed = false;
                target = compass→checkForEnemiesOnRange
                                                    (*(Occupant *) this, range);
                if (target→getId() ≠ this→id) {
                    got_target = true;
                }
            } else {
                if (target→areYouAlive() ∧ checkIfTargetIsOnRange()) {
                    attack();
                    this→action = ATKSTATE;
                } else {
                    got_target = false;
                    this→action = STANDINGSTATE;
                    this→changed = true;
                }
            }
        }
    }
    if (this→state ≡ MOVESTATE) {
        this→move();
        if (road.empty()) {
            this→state = STANDINGSTATE;
            this→action = STANDINGSTATE;
            this→changed = true;
        }
    }
    if (this→state ≡ ATKSTATE) {
        if (target→areYouAlive()) {
            if (checkIfTargetIsOnRange()) {
                if (¬road.empty())
                    road.clear();
                attack();
            } else {
                // If target is not on range move till it is
                // calculate road to target
                if (road.empty()) {
                    Position trg_pos = target→getPosition();
                    getOnRangeOf(trg_pos.getX(), trg_pos.getY());
                }
                move();
            }
        } else {
```

```
            this→state = STANDINGSTATE;
            this→action = STANDINGSTATE;
            this→changed = true;
        }
    }
    if (this→state ≡ GRABBINGSTATE) {
        if (this→action ≠ MOVESTATE) {
            Position target_pos = grab_target→getPosition();
            if (¬road.empty())
                road.clear();
            Position actual = obj_size.getPosition();
            road = compass→getFastestWay(actual, target_pos);
            this→action = MOVESTATE;
        } else {
            if (onRangeToGrabTarget()) {
                grab();
            } else if (road.empty()) {
                this→state = STANDINGSTATE;
                this→action = STANDINGSTATE;
                this→changed = true;
                mount_vehicule = false;
            } else if (¬road.empty()) {
                move();
            }
        }
    }
}

void Unit::calculateRoadTo(int x, int y) {
    if (¬checkIfAlreadyOnMyWay(x,y)) {
        this→state = MOVESTATE;
        Position destination(x, y);
        Position actual = obj_size.getPosition();
        road = compass→getFastestWay(actual, destination);
    }
}

void Unit::getOnRangeOf(int x, int y) {
    Position destination(x,y);
    Position actual = obj_size.getPosition();
    road = compass→getFastestWay(actual,destination);
    Position new_dest = road.back();
    if (new_dest.getX() ≡ actual.getX() ∧ new_dest.getY() ≡ actual.getY()) {
        this→state = STANDINGSTATE;
        this→action = STANDINGSTATE;
    }
}

void Unit::move() {
    int distance = unit_speed;
    int steps = 0;
    bool crash = false;
    compass→clearCompass();

    while (¬road.empty() ∧ steps ≤ distance ∧ ¬crash){
        Position pos = road.back();
        Size next_pos(pos.getX(),pos.getY(),
                      obj_size.getWidth(),obj_size.getHeight());
        if (compass→canIWalkToThisPosition(next_pos)) {
            double t_factor = compass→getTerrainFactorOn(
                                                    pos.getX(),pos.getY());
            // move unit position, range and weapon
            this→obj_size.moveTo(pos.getX(),pos.getY());
            this→weapon.movePosition(pos.getX(),pos.getY());
            int x_range = pos.getX() −
                    (range.getWidth() − obj_size.getWidth()) / 2;
            int y_range = pos.getY() −
                    (range.getHeight() − obj_size.getHeight()) / 2;
            this→range.moveTo(x_range,y_range);
            this→grab_range.moveTo(pos.getX() − 1,pos.getY() − 1);
            this→changed = true;
            this→action = MOVESTATE;
            road.pop_back();

            // increase or decrease distance til steps are more than unit speed
            if (steps < 1) {
                distance = (int) (t_factor * distance);
            } else if (steps < 1 ∧ unit_speed > 4) {
                distance = (int) (t_factor *distance *
                                                (1−(damage_recv/life_points)));
            }
            ++steps;
        } else {
            crash = true;
        }
    }
    if (crash) {
        Position destiny = road.front();
        Position actual = obj_size.getPosition();
        road = compass→getFastestWay(actual,destiny);
        this→action = STANDINGSTATE;
        this→changed = true;
    }
}

void Unit::attack() {
    if (fire_count ≡ 0 ∨ fire_count ≡ fire_rate) {
        fire_count = 0;
        // make a shot
        bullets.push_back(weapon.shotTarget(target));
        this→action = ATKSTATE;
        this→changed = true;
        fire_count = 0;
    }
    fire_count += 1;
}

std::string Unit::getActionState() const {
    return this→action;
}

Position Unit::getCurrentPosition() const {
    return this→obj_size.getPosition();
}

void Unit::setTargetToGrab(Teamable *object, std::string type) {
    // if its a flag any unit can grab it
    if (type ≡ FLAGTYPE) {
        grab_target = object;
        this→state = GRABBINGSTATE;
    } else if (this→type ≡ GRUNTTYPE ∧ object→getTeam() ≡ NEUTRAL
            ∧ compass→checkIfItIsGrabbable(type) ∧ type ≠ FLAGTYPE) {
        // Only Grunt robots can drive
        // If is not a flag, is a vehicle
        grab_target = object;
        this→state = GRABBINGSTATE;
        mount_vehicule = true;
    }
}

void Unit::grab() {
    this→state = STANDINGSTATE;
```

```
    this→action = STANDINGSTATE;
    this→changed = true;
    grab_target→changeTeam(this→team);
    if (mount_vehicule) {
        this→damage_recv = this→life_points;
        mount_vehicule = false;
    }
}

void Unit::setTargetToAttack(Occupant* target) {
    std::string type = target→getType();
    if (¬compass→checkIfItIsABuilding(type)) {
        this→state = ATKSTATE;
        this→target = target;
        // clean bullets on weapon when a new target is set
        this→weapon.setNewTarget(target);
    } else {
        if (weapon.isTheAttackExplosive()) {
            this→state = ATKSTATE;
            this→target = target;
            // clean bullets on weapon when a new target is set
            this→weapon.setNewTarget(target);
        }
    }
}

std::vector<Bullet*> Unit::collectBullets() {
    std::vector<Bullet*> tmp = bullets;
    bullets.clear();
    return tmp;
}

bool Unit::checkIfTargetIsOnRange() {
    bool on_range = true;
    Size trg_size = target→getSize();
    if (¬range.isThereACollision(trg_size))
        on_range = false;
    else {
        weapon.recalculateRoadToTarget();
        std::vector<Position>& bullet_road = weapon.getBulletRoad();
        Size b_size = weapon.getBulletSize();
        if (¬checkIfBulletWillHit(bullet_road,b_size))
            on_range = false;
    }

    return on_range;
}

bool Unit::checkIfBulletWillHit(std::vector<Position>& b_road, Size &b_size) {
    bool will_hit = true;
    for (auto x: b_road) {
        b_size.moveTo(x.getX(),x.getY());
        if (¬compass→canBulletWalkToThisPosition(b_size,*this,*target))
            will_hit = false;
    }
    return will_hit;
}

bool Unit::doYouHaveAnyBullets() {
    return (¬bullets.empty());
}

bool Unit::checkIfAlreadyOnMyWay(int x, int y) {
    bool on_my_way = false;
    if (state ≡ MOVESTATE) {
        Position destiny = road.front();
```

```cpp
        if (destiny.getX() ≡ x ∧ destiny.getY() ≡ y)
            on_my_way = true;
    }
    return on_my_way;
}

Size Unit::getNextPosition(int steps) {
    Position dest = road.front();
    Position pos = road.back();

    while ((dest.getX() ≠ pos.getX() ∨ dest.getY() ≠ pos.getY())
           ∧ steps ≠ 0) {
        road.pop_back();
        pos = road.back();
        −−steps;
    }

    return Size(pos.getX(),pos.getY(),obj_size.getWidth(),obj_size.getHeight());
}

bool Unit::onRangeToGrabTarget() {
    Size trg_size = grab_target→getSize();
    return grab_range.isThereACollision(trg_size);
}

void Unit::recalculateMyStartPosition() {
    Position actual = getPosition();
    Position valid_pos = compass→getAValidPositionForDestiny(actual);
    this→obj_size.moveTo(valid_pos.getX(),valid_pos.getY());
    this→range.moveTo(valid_pos.getX(),valid_pos.getY());
    this→weapon.movePosition(valid_pos.getX(),valid_pos.getY());
}

Unit::~Unit() {
    target = nullptr;
    grab_target = nullptr;
    for(auto& b: bullets) {
        delete(b);
    }
    delete(compass);
}
```

```cpp
//
// Created by rodian on 10/06/17.
//

#ifndef Z_TPGRUPAL_TERRITORY_H
#define Z_TPGRUPAL_TERRITORY_H


#include <vector>
#include "factory.h"
#include "teamable.h"

class Territory: public Teamable {
private:
    std::map<int,Factory*> factories;
    Teamable flag;
    Size territory_size;
    int id;

public:
    Territory(const std::map<int, Factory *> &factories, Position flag_position,
                  Size territory_size, int id);

    void grabFlag(std::string& new_team);

    int getTechLevel();

    void changeFactoriesTechLevel(int tech_level);

    Teamable* getFlag();

    std::map<int,Factory*>& getFactories();

    bool doesTerritorysOwnerChanged();

    int getId();

    ~Territory();
};


#endif //Z_TPGRUPAL_TERRITORY_H
```

```
//
// Created by rodian on 10/06/17.
//

#include "territory.h"
#define FLAGWIDTH 2
#define FLAGHEIGHT 3
Territory::Territory(const std::map<int, Factory *> &factories, Position flag,
                     Size size, int id) :
Teamable("Neutral",territory_size),factories(factories) ,
flag("Neutral",Size(flag.getX(),flag.getY(),FLAGWIDTH,FLAGHEIGHT)),
territory_size(size), id(id){}

void Territory::grabFlag(std::string& new_team) {
    changed = true;
    this→flag.changeTeam(new_team);
    this→changeTeam(new_team);
    for (auto fac: factories) {
        fac.second→changeTeam(new_team);
    }
}

int Territory::getTechLevel() {
    return (int)factories.size();
}

void Territory::changeFactoriesTechLevel(int tech_level) {
    for (auto fac: factories) {
        fac.second→changeTechLevel(tech_level);
    }
}

std::map<int, Factory*> &Territory::getFactories() {
    return factories;
}

bool Territory::doesTerritorysOwnerChanged() {
    if (this→flag.getTeam() ≠ this→team) {
        std::string new_team = flag.getTeam();
        grabFlag(new_team);
    }
    bool tmp = changed;
    changed = false;
    return tmp;
}

int Territory::getId() {
    return id;
}

Teamable *Territory::getFlag() {
    return &flag;
}

Territory::~Territory() {
    for(auto& f: factories) {
        delete(f.second);
    }
    factories.clear();
}


```

```
#ifndef Z_TPGRUPAL_TERRAIN_H
#define Z_TPGRUPAL_TERRAIN_H

#include <iostream>
#include <string>

// Class Terrain to be set on every cell
class Terrain {
    private:
        std::string kind;
        double factor;

    public:
        // Terrain is of constant kind and factor for moving over it
        Terrain(std::string& kind, int factor );

        // Returns string to the kind of terrain
        std::string getKind() const;

        // Returns the moving factor
        double getFactor() const;

        ~Terrain();
};


#endif //Z_TPGRUPAL_TERRAIN_H
```

```
//
// Created by rodian on 13/05/17.
//

#include "Terrain.h"

Terrain::Terrain(std::string &kind, int factor) :
                        kind(kind), factor(factor) {}

std::string Terrain::getKind() const{
    return this→kind;
}

double Terrain::getFactor() const {
    return this→factor;
}

Terrain::~Terrain() {}



```

```
//
// Created by rodian on 12/06/17.
//

#ifndef Z_TPGRUPAL_TEAM_H
#define Z_TPGRUPAL_TEAM_H


#include <vector>
#include "playerInfo.h"

class Team {
private:
    std::vector<PlayerInfo> players;
    int team_id;
public:
    Team(std::vector<PlayerInfo>& players,int team_id);

    bool doesTeamLose();

    std::vector<PlayerInfo>& getPlayersInfo();
};


#endif //Z_TPGRUPAL_TEAM_H
```

```
//
// Created by rodian on 12/06/17.
//

#include "team.h"

Team::Team(std::vector<PlayerInfo>& players, int team_id) :
        players(players), team_id(team_id) {}

bool Team::doesTeamLose() {
    bool lose = true;
    // if any of the fortress of the team is alive, they didn’t lose
    for (auto& p: players) {
        if (p.checkIfFortressLives()) {
            lose = false;
        }
        if (¬p.areYouStillConected()) {
            lose = true;
        }
    }
    return lose;
}

std::vector<PlayerInfo> &Team::getPlayersInfo() {
    return this→players;
}
```

```
//
// Created by rodian on 22/05/17.
//

#ifndef Z_TPGRUPAL_TEAMABLE_H
#define Z_TPGRUPAL_TEAMABLE_H

#include <iostream>
#include "size.h"

class Teamable {
protected:
    std::string team;
    Size obj_size;
    bool changed;

public:
    Teamable(std::string team, Size obj_size);

    // Builds a Teamable with "Neutral" team
    Teamable(Size size);

    // Changes the team of the object
    void changeTeam(std::string team);

    // Returns the actual team of the object
    std::string getTeam() const;

    // Returns the central position of the object
    Position getPosition() const;

    // Returns true if is there a collision with the object
    bool isThereACollision(Size& size);

    Size getSize() const;

    bool haveYouChanged();
};


#endif //Z_TPGRUPAL_TEAMABLE_H
```

jun 27, 17 14:46 **teamable.cpp** Page 1/1

```cpp
//
// Created by rodian on 22/05/17.
//

#include "teamable.h"

Teamable::Teamable(std::string team, Size size) : team(team), obj_size(size),
                                                    changed(false){}

Teamable::Teamable(Size size) : team("Neutral"), obj_size(size),
                                changed(false) {}

void Teamable::changeTeam(std::string team) {
    this→changed = true;
    this→team = team;
}

std::string Teamable::getTeam() const {
    return this→team;
}

Position Teamable::getPosition() const {
    return obj_size.getPosition();
}

bool Teamable::isThereACollision(Size& other) {
    return obj_size.isThereACollision(other);
}

Size Teamable::getSize() const {
    return this→obj_size;
}

bool Teamable::haveYouChanged() {
    bool tmp = changed;
    changed = false;
    return tmp;
}

```

jun 27, 17 14:46 **size.h** Page 1/1

```cpp
//
// Created by rodian on 17/05/17.
//

#ifndef Z_TPGRUPAL_SIZE_H
#define Z_TPGRUPAL_SIZE_H

#include "position.h"

//Class Size to represent the space on the map that an Occupant is standing on
class Size {
private:
    // (x,y) are the coordinates of the top−left corner of the object
    // all Occupants on game are Four−sided
    Position position;
    int width, height;

public:
    // width and lenght must be even numbers
    Size(int x, int y, int width, int height);

    // Returns the x position on the map
    Position getPosition() const;

    // Returns the width of the Object of this size
    int getWidth() const;

    // Returns the lenght of the Object of this size
    int getHeight() const;

    // Changes the position of the Object of this size to a new (x,y)
    void moveTo(int x, int y);

    // Returns True is this Size collisions with other.
    // Meant to check collision between different objects.
    // To fully be sure there is no collison, use this method on the size of
    // the other object as well
    bool isThereACollision(Size& other);

    // If other is partly or completely outside returns true.
    // If other is all inside this returns false.
    bool areYouHalfOutSide(Size& other);

    // Returns true if the position (x,y) received is inside this Size
    bool areYouOnThisPoint(int x, int y);

    ~Size();

    // Writes on max and min the maximum value on ’x’ coordinate that this size
    // has and the minimum value on min
    void calculateMaxAndMinForX(int& max, int& min);

    // Writes on max and min the maximum value on ’y’ coordinate that this size
    // has and the minimum value on min
    void calculateMaxAndMinForY(int& max, int& min);
};


#endif //Z_TPGRUPAL_SIZE_H
```

```cpp
//
// Created by rodian on 17/05/17.
//

#include "size.h"

Size::Size(int x, int y, int width, int height) : position(x,y),
                                               width(width), height(height){}

Position Size::getPosition() const {
    return this→position;
}

int Size::getWidth() const {
    return width;
}

int Size::getHeight() const {
    return height;
}

bool Size::isThereACollision(Size &other) {
    int x_max, x_min, y_max, y_min;
    bool collision = false;
    this→calculateMaxAndMinForX(x_max, x_min);
    this→calculateMaxAndMinForY(y_max, y_min);

    if (other.areYouOnThisPoint(x_max, y_max) ∨
        other.areYouOnThisPoint(x_max, y_min) ∨
        other.areYouOnThisPoint(x_min, y_max) ∨
        other.areYouOnThisPoint(x_min, y_min)) {
        collision = true;
    }

    other.calculateMaxAndMinForX(x_max, x_min);
    other.calculateMaxAndMinForY(y_max, y_min);
    if (this→areYouOnThisPoint(x_max, y_max) ∨
            this→areYouOnThisPoint(x_max, y_min) ∨
            this→areYouOnThisPoint(x_min, y_max) ∨
            this→areYouOnThisPoint(x_min, y_min)) {
        collision = true;
    }

   return collision;
}


bool Size::areYouHalfOutSide(Size &other) {
    int x_max, x_min, y_max, y_min;
    other.calculateMaxAndMinForX(x_max, x_min);
    other.calculateMaxAndMinForY(y_max, y_min);

    // If a point is not inside returns true
    return (¬this→areYouOnThisPoint(x_max, y_max) ∨
            ¬this→areYouOnThisPoint(x_max, y_min) ∨
            ¬this→areYouOnThisPoint(x_min, y_max) ∨
            ¬this→areYouOnThisPoint(x_min, y_min));
}

bool Size::areYouOnThisPoint(int x_pos, int y_pos) {
    int x_max, x_min, y_max, y_min;
    this→calculateMaxAndMinForX(x_max, x_min);
    this→calculateMaxAndMinForY(y_max, y_min);

    return ((x_pos < x_max) ∧ (x_pos ≥ x_min) ∧
            (y_pos < y_max) ∧ (y_pos ≥ y_min));
```

```cpp
}

void Size::calculateMaxAndMinForX(int &max, int &min) {
    max = position.getX() + width;
    min = position.getX();
}

void Size::calculateMaxAndMinForY(int &max, int &min) {
    max = position.getY() + height;
    min = position.getY();
}

void Size::moveTo(int x, int y) {
    position.moveTo(x,y);
}

Size::~Size() {}

```

```
//
// Created by rodian on 27/05/17.
//

#ifndef Z_TPGRUPAL_SERVER_H
#define Z_TPGRUPAL_SERVER_H


#include "../common/socket.h"
#include "../common/messenger.h"
#include "../common/Thread.h"
#include "menu.h"

class Server: public Thread  {
private:
    int port;
    Socket socket;
    bool running;
    Menu& menu;

public:
    // Recieves the arguments to build the Control Unit
    explicit Server(unsigned int port, Menu &menu);

    // use to start the process of of accepting clients
    void run();

    // Shuts down the accepter Socket and stops run()
    void stop();

    ~Server();
};


#endif //Z_TPGRUPAL_SERVER_H
```

```
//
// Created by rodian on 27/05/17.
//

#include "server.h"
#include <sstream>

Server::Server(unsigned int port, Menu &menu) : socket(port),
                                                running(true),
                                                menu(menu),
                                                port(port) {}

void Server::run() {
    try {
        int i = 0;
        while(this→running) {
            Socket new_client = this→socket.accept_client();
            Messenger* messenger = new Messenger(new_client);
            std::string id_new_player;
            id_new_player = "Player" + i;
            bool added = menu.addPlayer(messenger, menu,id_new_player);
            while (¬added) {
                std::string new_player;
                ++i;
                new_player = "Player" + std::to_string(i);
                added = menu.addPlayer(messenger, menu,id_new_player);
            }

        }
    } catch (SocketError& e) {
        std::string error = e.what();
        std::cerr << error << std::endl;
    }
}

void Server::stop() {
    this→running = false;
    this→socket.shutdown();
}

Server::~Server() {}
```

```
//
// Created by rodian on 18/05/17.
//

#ifndef Z_TPGRUPAL_POSITION_H
#define Z_TPGRUPAL_POSITION_H

// Position class to keep the coordinates of the map
class Position {
private:
    int x, y;
public:
    Position(int x, int y);

    // Returns the x coordinate
    int getX() const;

    // Returns the y coordinate
    int getY() const;

    // Changes the position to the (x,y) received
    void moveTo(int x, int y);
};


#endif //Z_TPGRUPAL_POSITION_H
```

```
//
// Created by rodian on 18/05/17.
//

#include "position.h"

Position::Position(int x, int y) : x(x), y(y) {}

int Position::getX() const {
    return this->x;
}

int Position::getY() const {
    return this->y;
}

void Position::moveTo(int x, int y) {
    this->x = x;
    this->y = y;
}
```

```cpp
//
// Created by rodian on 12/06/17.
//

#ifndef Z_TPGRUPAL_PLAYERINFO_H
#define Z_TPGRUPAL_PLAYERINFO_H


#include <messenger.h>
#include "factory.h"
#include "territory.h"

class PlayerInfo {
private:
    std::string id;
    Factory* fortress;
    int tech_level;
    Messenger* player_messenger;
    std::vector<Territory*> territories;

public:
    PlayerInfo(std::string id ,Factory* fortress);

    PlayerInfo(std::string id);

    std::string getPlayerId() const;

    bool checkIfFortressLives();

    int getTechLevel();

    void increaseTechLevel();

    void decreaseTechLevel();

    Factory* getFortress();

    void addMessenger(Messenger* player_messenger);

    Messenger* getMessenger();

    void addTerritory(Territory* territory);

    void recalculateTechLevel();

    void eliminateThisTerritory(Territory* territory);

    bool areYouStillConected();

    void addFortress(Factory* fortress);

    ~PlayerInfo();
};


#endif //Z_TPGRUPAL_PLAYERINFO_H
```

```cpp
//
// Created by rodian on 12/06/17.
//

#include "playerInfo.h"

PlayerInfo::PlayerInfo(std::string id, Factory *fortress) :
        id(id), fortress(fortress), tech_level(1) {}


PlayerInfo::PlayerInfo(std::string id) : id(id), tech_level(0) {}

std::string PlayerInfo::getPlayerId() const {
    return id;
}

bool PlayerInfo::checkIfFortressLives() {
    return fortress→areYouAlive();
}

int PlayerInfo::getTechLevel() {
    return tech_level;
}

void PlayerInfo::increaseTechLevel() {
    this→tech_level += 1;
}

void PlayerInfo::decreaseTechLevel() {
    this→tech_level -= 1;
}

Factory *PlayerInfo::getFortress() {
    return fortress;
}

void PlayerInfo::addMessenger(Messenger *messenger) {
    this→player_messenger = messenger;
}

Messenger *PlayerInfo::getMessenger() {
    return this→player_messenger;
}

void PlayerInfo::addTerritory(Territory *territory) {
    territories.push_back(territory);
    recalculateTechLevel();
}

void PlayerInfo::recalculateTechLevel() {
    tech_level = 0;
    for (auto& t: territories) {
        tech_level += t→getTechLevel();
    }

    for (auto& t: territories) {
        t→changeFactoriesTechLevel(tech_level);
    }
}

void PlayerInfo::eliminateThisTerritory(Territory* territory) {
    std::vector<Territory*>::iterator it = territories.begin();
    for (; it ≠ territories.end();) {
        if (territory→getId() ≡ (*it)→getId()) {
            it = territories.erase(it);
        } else {
            ++it;
        }
    }
    recalculateTechLevel();
}

PlayerInfo::~PlayerInfo() {
    player_messenger = nullptr;
}

bool PlayerInfo::areYouStillConected() {
    return player_messenger→isConnected();
}

void PlayerInfo::addFortress(Factory *fortress) {
    this→fortress = fortress;
}

```

```
//
// Created by rodian on 29/05/17.
//

#ifndef Z_TPGRUPAL_PLAYER_H
#define Z_TPGRUPAL_PLAYER_H

#include <iostream>
#include "../common/messenger.h"
#include "../common/Thread.h"
#include "menu.h"
#include "lobby.h"

class CommandMonitor;
class ControlUnit;
class Lobby;
class Menu;
class Game;

class Player: public Thread {
private:
    Messenger* messenger;
    std::string id;
    int color;
    bool conected,on_menu,on_lobby ,playing,ready;
    CommandMonitor* commands;
    ControlUnit* control;
    Lobby* lobby;
    Menu& menu;

public:
    Player(Messenger *messenger, Menu &menu, std::string& id);

    void run();

    void updateInfo(std::string& info);

    void addLobby(Lobby* lobby);

    void addControlUnit(ControlUnit* control, CommandMonitor* commands);

    Messenger* getMessenger();

    void shutDown();

    std::string getId() const;

    void getInGame();

    bool areYouReady();

    void resetReady();

    bool areYouInLobby();

    ~Player();

private:
    void processMenuCommands(std::string& cmd);

    void processLobbyCommands(std::string& cmd);

    std::string getNextData(std::string& line);
};


```

```
#endif //Z_TPGRUPAL_PLAYER_H
```

```
//
// Created by rodian on 29/05/17.
//

#include <sstream>
#include "player.h"

#define RETURNTOMENU "returntomenu"
#define OK "ok"

Player::Player(Messenger *msg, Menu &menu, std::string& id) :
        messenger(msg),id(id), conected(true),on_menu(true),on_lobby(false),
        playing(false),menu(menu),ready(false) {}

void Player::run() {
    try {
        while (messenger→isConnected()) {
            std::string new_cmd = messenger→recieveMessage();

            std::cerr << "Player " << id << " ejecuta " << new_cmd << std::endl;
            if (on_menu) {
                processMenuCommands(new_cmd);
            } else if (on_lobby) {
                processLobbyCommands(new_cmd);
            } else if (new_cmd ≡ RETURNTOMENU) {
                this→playing = false;
                this→on_menu = true;
                messenger→sendMessage(this→menu.getLobbiesInfo());
            } else if (playing) {
                commands→addCommand(this→id, new_cmd, control);
            }
        }
    } catch(SocketError e) {
        conected = false;
        if (on_lobby ∨ playing) {
            lobby→disconectPlayer(this);
        }
    }
}

void Player::updateInfo(std::string &info) {
    messenger→sendMessage(info);
}

void Player::addControlUnit(ControlUnit *control, CommandMonitor* commands) {
    this→playing = true;
    this→control = control;
    this→commands = commands;
}

Messenger *Player::getMessenger() {
    return messenger;
}

void Player::addLobby(Lobby* lobby) {
    this→lobby = lobby;
    on_lobby = true;
}

void Player::processMenuCommands(std::string &full_cmd) {
    std::string cmd = getNextData(full_cmd);
    if (cmd ≡ "createlobby") {
        this→menu.createNewLobby(this);
        on_menu = false;
    }  else if (cmd ≡ "joinlobby") {
        std::string lobby_id = getNextData(full_cmd);
        int id = std::stoi(lobby_id);
        if (this→menu.addToLobby(id,this))
            on_menu = false;
    }  else if (cmd ≡ "lobbyinfo") {
        messenger→sendMessage(this→menu.getLobbiesInfo());
    }   else if (cmd ≡ "changename") {
        std::string new_name = getNextData(full_cmd);
        std::string ans = this→menu.changeName(new_name);
        if (ans ≡ "ok")
            this→id = new_name;
        messenger→sendMessage(ans);
    } else {
        messenger→sendMessage("Invalid cmd");
    }
}

void Player::processLobbyCommands(std::string &full_cmd) {
    std::string cmd = getNextData(full_cmd);
    if (cmd ≡ "startgame") {
        std::string map = getNextData(full_cmd);
        this→lobby→startGame(map);
    } else if (cmd ≡ "ready") {
        this→ready = true;
        this→lobby→ready();
    }  else if (cmd ≡ "unready") {
        this→ready = false;
        this→lobby→unReady();
    }  else if (cmd ≡ "exitlobby") {
        this→ready = false;
        this→lobby→unReady();
        this→on_lobby = false;
        this→on_menu = true;
        this→lobby→exitLobby(this);
    } else if (cmd ≡ "mapsinfo") {
        messenger→sendMessage(lobby→get_loaded_maps());
    } else {
        messenger→sendMessage("Invalid cmd");
    }
}

void Player::shutDown() {
    conected = false;
    messenger→shutdown();
}

std::string Player::getId() const {
    return id;
}

std::string Player::getNextData(std::string& line) {
    std::size_t found = line.find('-');
    std::string data = line.substr(0,found);
    line.erase(0,found+1);
    return data;
}

void Player::getInGame() {
    // Notify the client the game is starting
    std::stringstream msg;
    msg << "startgame-" << OK;
    messenger→sendMessage(msg.str());
    this→on_lobby = false;
    this→playing = true;
}

bool Player::areYouReady() {
```

```
    return this→ready;
}

Player::~Player() {
    commands = nullptr;
    control = nullptr;
    lobby = nullptr;
}

void Player::resetReady() {
    this→ready = false;
}

bool Player::areYouInLobby() {
    return on_lobby;
}
```

```cpp
#ifndef Z_TPGRUPAL_OCCUPANT_H
#define Z_TPGRUPAL_OCCUPANT_H

#include <iostream>
#include "size.h"
#include "teamable.h"


// Class Occupant so any object knows where is on the map
class Occupant: public Teamable {
protected:
    int id, life_points, damage_recv;
//    Size occ_size;
    std::string type;
    bool disappear;

public:
    // Constructor for Occupant on a specific position saved in Size and it’s id
    Occupant(int id, int life, std::string type, Size occ_size);

    // Meant to create a null Occupant
    Occupant();

    // Returns the id of the object
    std::string getType() const;

    void reduceLifeBy(int dmg);

    bool areYouAlive();

//    bool isThereACollision(Size& size);

    int getLifeLeft();

    int getId();

    void mustDisappear();

    bool doYouNeedToDisappear();

    ~Occupant();
};


#endif //Z_TPGRUPAL_OCCUPANT_H
```

```cpp
//
// Created by rodian on 13/05/17.
//

#include "Occupant.h"

Occupant::Occupant(int id, int life, std::string type, Size position) :
       Teamable(position), id(id), life_points(life), type(type)/*,
       occ_size(position)*/, damage_recv(0), disappear(false) {}

Occupant::Occupant() :
        Teamable(Size(3,3,3,3)), id(-1), life_points(0), type("nullOccupant")/*,
       occ_size(position)*/, damage_recv(0) {}

std::string Occupant::getType() const {
    return this->type;
}

void Occupant::reduceLifeBy(int dmg) {
    this->damage_recv += dmg;
    this->changed = true;
}

bool Occupant::areYouAlive() {
    return ((life_points - damage_recv) > 0);
}

//bool Occupant::isThereACollision(Size &other) {
//    return occ_size.isThereACollision(other);
//}

int Occupant::getLifeLeft() {
    if (life_points - damage_recv > 0) {
        return life_points - damage_recv;
    } else {
        return 0;
    }
}

Occupant::~Occupant() {}

int Occupant::getId() {
    return id;
}

bool Occupant::doYouNeedToDisappear() {
    return disappear;
}

void Occupant::mustDisappear() {
    disappear = true;
}

```

```
//
// Created by rodian on 15/05/17.
//

#ifndef Z_TPGRUPAL_NODE_H
#define Z_TPGRUPAL_NODE_H

#include "cell.h"
#include "size.h"
#include <iostream>
#include <vector>

// Class Node meant to be use in the calculation of A* algorithm
class Node {
private:
    int h_value, g_value;
    Node* parent;
    bool was_visited;
    Size size;

public:
    Node(int x, int y, int width, int lenght);

    void setHValue(int h);

    void setGValue(int g, int terrain_factor);

    int getGValue();

    int getFValue() const;

    int getHvalue() const;

    void setNewParent(Node* parent);

    Node* getParent() const;

    Position getPosition() const;

    Size getSize() const;

    bool beenSeen() const;

    int getFValueIfGWere(int g, int terrain_factor);

    void clean();

    ~Node();
};

#endif //Z_TPGRUPAL_NODE_H
```

```
//
// Created by rodian on 15/05/17.
//

#include "node.h"

Node::Node(int x, int y, int width, int lenght) : h_value(0), g_value(0),
                        was_visited(false), size(x, y, width, lenght) {}

void Node::setHValue(int h) {
    this→h_value = h;
}

void Node::setGValue(int g, int terrain_factor) {
    this→g_value = (g + 4)+terrain_factor*terrain_factor;
    this→was_visited = true;
}

int Node::getGValue() {
    return this→g_value;
}

int Node::getFValue() const {
    return (h_value + g_value);
}

int Node::getHvalue() const {
    return this→h_value;
}

void Node::setNewParent(Node *parent) {
    this→parent = parent;
}

Node* Node::getParent() const {
    return this→parent;
}

Position Node::getPosition() const{
    return size.getPosition();
}

Size Node::getSize() const {
    return this→size;
}

bool Node::beenSeen() const {
    return this→was_visited;
}

int Node::getFValueIfGWere(int g, int terrain_factor) {
    return (h_value + (g + 4)*terrain_factor);
}

void Node::clean() {
    this→h_value = 0;
    this→g_value = 0;
    this→was_visited = false;
}

Node::~Node() {
    this→parent = nullptr;
}
```

```cpp
//
// Created by rodian on 29/05/17.
//

#ifndef Z_TPGRUPAL_MENU_H
#define Z_TPGRUPAL_MENU_H


#include <vector>
#include <iostream>
#include <mutex>
#include "player.h"
class Player;
class Lobby;

class Menu {
private:
    std::vector<Player*> players;
    std::mutex m;
    int lobby_counter;
    std::vector<Lobby*> lobbies;
    std::string& config;
public:
    Menu(std::string& config);

    bool addPlayer(Messenger* msgr, Menu& menu, std::string player_id);

    void createNewLobby(Player* player);

    std::string getLobbiesInfo();

    std::string changeName(std::string &new_name);

    bool addToLobby(int id_lobby, Player *player);

    void shutDown();

    void disconectPlayer( Player *player);

    ~Menu();
};


#endif //Z_TPGRUPAL_MENU_H
```

```cpp
//
// Created by rodian on 29/05/17.
//

#include <sstream>
#include <string>
#include "menu.h"

#define ERROR_MSG "joinlobby−error"
#define OK_MSG "joinlobby−ok"

Menu::Menu(std::string& config) : lobby_counter(0), config(config) {}

bool Menu::addPlayer(Messenger *msgr, Menu& menu, std::string player_id) {
    Lock l(m);
    // check desconected players
    std::vector<Player *>::iterator p = players.begin();
    for (;p ≠ players.end();) {
        if (¬(*p)→getMessenger()→isConnected()) {
            (*p)→shutDown();
            (*p)→join();
            delete((*p));
            p = players.erase(p);
        } else {
            ++p;
        }
    }

    for(Player* p : players) {
        if (p→getId() ≡ player_id) {
            return false;
        }
    }
    this→players.push_back(new Player(msgr, menu, player_id));
    this→players.back()→start();
    return true;
}

void Menu::createNewLobby(Player* player) {
    Lock l(m);
    Lobby* new_lobby = new Lobby(lobby_counter++, config,m);
    lobbies.emplace_back(new_lobby);
    lobbies.back()→addPlayer(player);
    player→addLobby(new_lobby);
    player→getMessenger()→sendMessage(OK_MSG);
}

std::string Menu::getLobbiesInfo() {
    Lock l(m);
    std::string info = "lobbyinfo−";
    std::vector<Lobby *>::iterator it = lobbies.begin();
    for (;it ≠ lobbies.end();) {
        if (¬(*it)→haveGameFinished()) {
            info += std::to_string((*it)→get_id()) + "−";
            ++it;
        } else {
            (*it)→shutDown();
            delete(*it);
            it = lobbies.erase(it);
            −−lobby_counter;
            // check desconected players
            std::vector<Player *>::iterator p = players.begin();
            for (;p ≠ players.end();) {
                if (¬(*p)→getMessenger()→isConnected()) {
                    (*p)→shutDown();
                    (*p)→join();
                    delete((*p));
                    p = players.erase(p);
                } else {
                    ++p;
                }
            }
        }
    }
    return info;
}

bool Menu::addToLobby(int id_lobby, Player* player) {
    Lock l(m);
    for (Lobby* lobby : lobbies) {
        if (lobby→get_id() ≡ id_lobby) {
            if (lobby→addPlayer(player)) {
                player→addLobby(lobby);
                player→getMessenger()→sendMessage(OK_MSG);
                return true;
            } else {
                player→getMessenger()→sendMessage(ERROR_MSG);
                return false;
            }
        }
    }
    player→getMessenger()→sendMessage(ERROR_MSG);
    return false;
}

void Menu::shutDown() {
    for(auto p: players) {
        p→shutDown();
        p→join();
        delete(p);
    }

    for(auto l: lobbies) {
        l→shutDown();
        delete(l);
    }
}

Menu::~Menu() {}

std::string Menu::changeName(std::string &new_name) {
    Lock l(m);
    for(Player* p : players) {
        if (p→getId() ≡ new_name) {
            return "error";
        }
    }
    return "ok";
}

void Menu::disconectPlayer(Player *player) {
    Lock l(m);
    std::vector<Player*>::iterator it = players.begin();
    for(;it ≠ players.end();++it) {
        if ((*it)→getId() ≡ player→getId()) {
            (*it)→shutDown();
            (*it)→join();
            delete((*it));
            players.erase(it);
            break;
        }
    }
```

```
}

```

```cpp
#ifndef Z_TPGRUPAL_MAPLOADER_H
#define Z_TPGRUPAL_MAPLOADER_H


#include <vector>
#include <pugixml.hpp>
#include <memory>
#include "map.h"
#include "Occupant.h"
#include "cell.h"
#include "unit.h"
#include "factory.h"
#include "territory.h"

class MapLoader {
    std::vector<std::vector<Cell>> map;
    std::vector<Occupant *> occupants;
    std::vector<Unit> units;
    std::string map_string;
    std::string &config;
    std::vector<UnitMold*> unit_mold;
    std::vector<UnitMold*> vehicle_mold;
    std::map<std::string, Weapon> weapons;
    std::shared_ptr<Map> game_map;
    std::vector<Factory*> forts;

    std::vector<Territory*> territories;
    int internal_positions;
public:
    MapLoader(std::string path, std::string& config);

    ~MapLoader();

    std::vector<Occupant*> getOccupants();

    std::vector<Unit> getUnits();

    std::shared_ptr<Map> get_map();

    std::vector<Factory*> get_forts();

    std::vector<Territory*> get_territories();

private:
    void load_structs(const pugi::xml_node &root, const pugi::xml_node &cfg);

    Factory* create_factory(int id, int hp, std::string& type, Size size);

    void load_unit_molds(pugi::xml_node node);
    void load_weapons(pugi::xml_node weapons);
    void create_map();
    void load_territories(const pugi::xml_node &structs_cfg,
                          const pugi::xml_node &root, int id_counter);

    void create_territory(int hp, const pugi::xml_node &territory,
                          int &id_counter,
                          std::map<int, Factory *> &factories_in_territory);

    void load_vehicle_molds(const pugi::xml_node &vehicles);

    void load_mold(std::vector<UnitMold*>& mold,
                   const pugi::xml_node& source);
};


#endif //Z_TPGRUPAL_MAPLOADER_H
```

```cpp
#include <map>
#include "MapLoader.h"
#include <pugixml.hpp>
#include <sstream>


const std::map<std::string, int> terrain_factor {
        {std::string("Tierra"), int(1)},
        {std::string("Agua"), int(7/10)},
        {std::string("Carretera"), int(15/10)},
        {std::string("Lava"), int(1000)}
};

MapLoader::MapLoader(std::string path, std::string& config) : config(config) {
    pugi::xml_document doc;
    pugi::xml_parse_result result = doc.load_file(path.c_str());
    if (¬result) {
        std::cout << "ERROR LOADING MAP: " << path << ": " <<
                  result.description() << std::endl;
    }
    std::stringstream stream;
    doc.save(stream);
    map_string = stream.str();
    // Get root node
    pugi::xml_node root = doc.child("Map");
    pugi::xml_node map_node = root.child("Terrain");

    pugi::xml_document cfg;
    cfg.load_file(config.c_str());
    pugi::xml_node cfg_node = cfg.child("Config");

    internal_positions = std::stoi(cfg_node.child("Cells").
            attribute("internal_positions").value());

    // Iterate over every row
    unsigned int coord_y = 0;
    auto row = map_node.children().begin();
    for (; row ≠ map_node.children().end(); ++row) {
        unsigned int coord_x = 0;
        // Iterate over every row creating cells
        auto cell = row→children().begin();
        for (; cell ≠ row→children().end(); ++cell) {
            if (map.size() ≤ (coord_x)) {
                map.push_back(std::vector<Cell>());
            }

            std::string terrain = cell→attribute("terrain").value();
            std::string structure = cell→attribute("struct").value();

            int factor = terrain_factor.find(terrain)→second;

            // Create a new cell and push it to the row
            map.at(coord_x).emplace_back(coord_x * internal_positions, coord_y,
                                         internal_positions, internal_positions,
                                         terrain, factor);
            coord_x++;
        }
        // Push the whole row to the map
        coord_y += internal_positions;
    }



    create_map();
    load_unit_molds(cfg_node.child("Units"));
    load_vehicle_molds(cfg_node.child("Vehicles"));
    load_weapons(cfg_node.child("Weapons"));
    load_structs(root, cfg_node.child("Structs"));
}

void MapLoader::load_structs(const pugi::xml_node &root,
                             const pugi::xml_node &cfg) {

    int id_counter = 0;
    pugi::xml_node structs = root.child("Structures");
    pugi::xml_node structure_cfg = cfg.find_child_by_attribute("type", "Rock");
    int size_x = std::stoi(structure_cfg.attribute("size_x").value());
    int size_y = std::stoi(structure_cfg.attribute("size_y").value());
    int hp = std::stoi(structure_cfg.attribute("hp").value());

    std::string type = structure_cfg.attribute("type").value();
    for(auto& rock : structs) {
        int x = std::stoi(rock.attribute("x").value()) *
            internal_positions;
        int y = std::stoi(rock.attribute("y").value()) *
            internal_positions;
        Occupant* f = new Occupant(id_counter++, hp, type,
                                   Size(x, y, size_x, size_y));
        occupants.push_back(f);
    }

    load_territories(cfg, root.child("Territories"), id_counter);

}

void MapLoader::create_map() {
    int width = (int) map.at(0).size() * internal_positions;
    int height = (int) map.size() * internal_positions;
    int x = 0;
    int y = 0;
    game_map = std::shared_ptr<Map>(new Map(x, y, width, height, map,
                                            &occupants, map_string));
}

MapLoader::~MapLoader() {
}

std::vector<Occupant*> MapLoader::getOccupants() {
    return this→occupants;
}

std::vector<Unit> MapLoader::getUnits() {
    return this→units;
}

Factory *MapLoader::create_factory(int id, int hp, std::string &type, Size size)
 {
    std::vector<UnitMold*> both = unit_mold;
    for (UnitMold* vehicle : vehicle_mold) {
        both.push_back(vehicle);
    }
    return (new Factory(id, hp, type, size, both, game_map, weapons));
}

void MapLoader::load_unit_molds(pugi::xml_node units) {
    load_mold(unit_mold, units);
}

void MapLoader::load_weapons(pugi::xml_node weapons) {
    for(auto& unit : weapons.children()) {
        std::string type = unit.attribute("type").value();
        int width = std::stoi(unit.attribute("size_x").value());
```

```cpp
        int height = std::stoi(unit.attribute("size_y").value());
        int speed = std::stoi(unit.attribute("speed").value());
        int damage = std::stoi(unit.attribute("damage").value());
        std::string exp = unit.attribute("explosive").value();
        bool explosive = false;
        if (exp ≡ "yes")
            explosive = true;

        Size s(0, 0, width, height);
        this→weapons.emplace(type, Weapon(type, damage, speed, explosive, s));
    }
}

std::shared_ptr<Map> MapLoader::get_map() {
    return game_map;
}

void MapLoader::load_territories(const pugi::xml_node &structs_cfg,
                                 const pugi::xml_node &root, int id_counter) {
    pugi::xml_node factory_cfg = structs_cfg.
            find_child_by_attribute("type", "Factory");

    int size_x = std::stoi(factory_cfg.attribute("size_x").value());
    int size_y = std::stoi(factory_cfg.attribute("size_y").value());
    int hp = std::stoi(factory_cfg.attribute("hp").value());
    std::string type = factory_cfg.attribute("type").value();

    for(auto& territory : root.children()) {
        std::map<int, Factory*> factories_in_territory;
        for(auto& factory : territory.children()) {
            std::vector<UnitMold*> mold = unit_mold;
            std::string name = factory.name();
            if (name ≡ "VehicleFactory") {
                mold = vehicle_mold;
            }

            int x = std::stoi(factory.attribute("x").value()) *
                internal_positions;
            int y = std::stoi(factory.attribute("y").value()) *
                internal_positions;
            Size s(x, y, size_x, size_y);
            Factory* f = new Factory(id_counter, hp, type, s, mold, game_map, we
apons);
            factories_in_territory[id_counter] = f;
            ++id_counter;
        }
        create_territory(hp, territory, id_counter, factories_in_territory);
    }
}

void MapLoader::create_territory(int hp, const pugi::xml_node &territory,
                                 int &id_counter,
                                 std::map<int, Factory *> &factories_in_territor
y) {
    std::string name = territory.name();
    int x = std::stoi(territory.attribute("center_x").value()) *
            internal_positions;
    int y = std::stoi(territory.attribute("center_y").value()) *
            internal_positions;
    /* If it's a fort we also create a factory */
    if (name ≡ "Fort") {
        Factory* f = create_factory(id_counter, hp, name,
                                    Size(x, y, 20, 20));
        factories_in_territory[id_counter] = f;
        forts.push_back(f);
        ++id_counter;
    }

    int min_x = std::stoi(territory.attribute("min_x").value());
    int min_y = std::stoi(territory.attribute("min_y").value());
    int max_x = std::stoi(territory.attribute("max_x").value());
    int max_y = std::stoi(territory.attribute("max_y").value());
    int width =  (min_x + max_x) / 2;
    int height = (min_y + max_y) / 2;
    Size flag(x, y, width, height);
    Position flag_position(x, y);
    Territory* t = new Territory(factories_in_territory, flag_position, flag,
    ++id_counter);
    territories.emplace_back(t);
}

std::vector<Factory *> MapLoader::get_forts() {
    return forts;
}

std::vector<Territory *> MapLoader::get_territories() {
    return territories;
}

void MapLoader::load_vehicle_molds(const pugi::xml_node &vehicles) {
    load_mold(vehicle_mold, vehicles);
}

void MapLoader::load_mold(std::vector<UnitMold *>& mold,
                          const pugi::xml_node &source) {
    for (auto& unit : source.children()) {
        std::string type = unit.attribute("type").value();
        std::string weapon = unit.attribute("weapon").value();
        int width = std::stoi(unit.attribute("size_x").value());
        int height = std::stoi(unit.attribute("size_y").value());
        int hp = std::stoi(unit.attribute("hp").value());
        int fire_rate = std::stoi(unit.attribute("fire_rate").value());
        int range = std::stoi(unit.attribute("range").value());
        int speed = std::stoi(unit.attribute("speed").value());
        int time = std::stoi(unit.attribute("time").value());
        int quantity = std::stoi(unit.attribute("quantity").value());
        int tech_level = std::stoi(unit.attribute("tech_level").value());
        mold.push_back(
                new UnitMold(tech_level, hp, range, width, height, speed,
                             fire_rate, time, quantity, type, weapon));
    }
}
```

```cpp
//
// Created by rodian on 18/05/17.
//

#ifndef Z_TPGRUPAL_MAP_H
#define Z_TPGRUPAL_MAP_H


#include <vector>
#include <map>
#include "cell.h"
#include "Occupant.h"
// later written
//class Compass;
class Unit;

class Map {
private:
    std::vector<std::vector<Cell>> terrain_map;
    Size map_size;
    std::vector<Occupant*>* all_occupants;
    std::string xml;
    std::map<std::string,std::string> types;
    /*std::vector<Unit>& all_units;*/

public:
    // Map receives the center position (x,y) and dimensions width and height
    Map(int x, int y, int width, int height,
        std::vector<std::vector<Cell>>& terrain_map,
        std::vector<Occupant*>* all_occupants,
        std::string& xml);

    // Recieves the coordinates (x,y) and returns the terrain factor on that
    // position on the map.
    double getTerrainFactorOn(int x, int y);

    // Returns the name of the type of Terrain
    std::string getTerrainType(int x, int y);

    // Returns true if the points are empty
    bool areThisPointsEmpty(Size &size, int id);

    // Returns true if points are empty or it is the Occupant
    bool areThisPointsEmpty(Size& size, Occupant &shooter, Occupant& occupant);

    // Recieves the size of an object on the position that wants to be walk
    // Returns true if the object fits and can step to that position
    bool canIWalkToThisPosition(Size &size, int id);

    // Recieves the size of an object on the position that wants to be walk
    // Returns true if the object fits and can step to that position ignoring
    // the occupant from parameter
    bool canBulletWalkToThisPosition(Size& size, Occupant &shooter,
                                               Occupant& target);

    // Returns the width of the map
    int getWidth();

    // Returns the Heigth of the map
    int getHeigth();

    bool doesThisPositionExist(int x, int y);

    bool isThereLava(Size& other_size);

    bool thereIsABridge(Size& other_size);

    std::string& get_map();

    std::vector<Occupant*>& getOccupants();

    void updateOccupants(std::vector<Occupant*>* all_occupants);

    Occupant* checkForEnemiesOn(Size& range, Occupant& unit);

    bool tellIfItIsGrabbable(std::string& type);

    bool tellIfItIsBuilding(std::string& type);

private:
    void buildTypeMap();
};


#endif //Z_TPGRUPAL_MAP_H
```

```cpp
//
// Created by rodian on 18/05/17.
//

#include "map.h"
#define BUILDING "Building"
#define NATURE "Nature"
#define UNIT "Unit"
#define VEHICLE "Vehicule"
#define FLAG "flag"

Map::Map(int x, int y, int width, int height,
   std::vector<std::vector<Cell>>& terrain_map,
         std::vector<Occupant*>* occupants,
         std::string& xml) : map_size(x,y,width,height),
    terrain_map(terrain_map), all_occupants(occupants), xml(xml) {
    this→buildTypeMap();
}

double Map::getTerrainFactorOn(int x, int y) {
    int w_cell = terrain_map[0][0].getWidthOfCell();
    int x_pos = x / w_cell;
    int y_pos = y / w_cell;

    return terrain_map[x_pos][y_pos].getMovementFactor();
}

std::string Map::getTerrainType(int x, int y) {
    int w_cell = terrain_map[0][0].getWidthOfCell();
    int x_pos = x / w_cell;
    int y_pos = y / w_cell;

    return terrain_map[x_pos][y_pos].getTerrainType();
}

bool Map::areThisPointsEmpty(Size &size, int id) {
    bool no_collision = true;
    for(auto x: *all_occupants) {
        if(x→getId() ≠ id
           ∧ x→isThereACollision(size) ∧ x→getType() ≠ "Bridge") {
            no_collision = false;
            break;
        }
    }
    return no_collision;
}

bool Map::areThisPointsEmpty(Size &size, Occupant &shooter, Occupant &occupant)
{
    bool no_collision = true;
    for(auto x: *all_occupants) {
        if(x→getId() ≠ occupant.getId() ∧ x→isThereACollision(size)
           ∧ x→getType() ≠ "Bridge"  ∧ x→getId() ≠ shooter.getId()) {
            no_collision = false;
            break;
        }
    }
    return no_collision;
}

int Map::getWidth() {
    return map_size.getWidth();
}

int Map::getHeigth() {
    return map_size.getHeight();
```

```cpp
}

bool Map::canIWalkToThisPosition(Size &other_size, int id) {
    bool you_can = true;

    // if the object is stepping out of the map
    if (map_size.areYouHalfOutSide(other_size))
        you_can = false;
    // if the object is stepping into lava
    if (isThereLava(other_size)) {
        you_can = false;
        if (thereIsABridge(other_size))
            you_can = true;
    }
    if (¬areThisPointsEmpty(other_size, id)) {
        you_can = false;
    }

    return (you_can);
}

bool Map::canBulletWalkToThisPosition(Size &other_size, Occupant &shooter,
                                      Occupant &target) {
    bool you_can = true;

    // if the object is stepping out of the map
    if (map_size.areYouHalfOutSide(other_size))
        you_can = false;

    if (¬areThisPointsEmpty(other_size,shooter,target)) {
        you_can = false;
    }

    return (you_can);
}

bool Map::doesThisPositionExist(int x, int y) {
    return map_size.areYouOnThisPoint(x,y);
}

bool Map::isThereLava(Size& other_size) {
    int x_max, x_min, y_max, y_min;
    other_size.calculateMaxAndMinForX(x_max, x_min);
    other_size.calculateMaxAndMinForY(y_max, y_min);

    int w_cell = terrain_map[0][0].getWidthOfCell();
    // Check if any of the corners are stepping into lava
    for (int y = y_min; y ≤ y_max; ++y) {
        for (int x = x_min; x ≤ x_max; ++x) {
            if (doesThisPositionExist(x,y)) {
                // Calculate the cell that holds this position
                int x_pos = x / w_cell;
                int y_pos = y / w_cell;

                if (terrain_map[x_pos][y_pos].getTerrainType() ≡ "Lava") {
                    return terrain_map[x_pos][y_pos].isThereACollision(
                            other_size);
                }
            }
        }
    }

    return false;
}

bool Map::thereIsABridge(Size& other_size) {
```

```cpp
    bool bridge = false;
    for(auto x: *all_occupants) {
        if(x→isThereACollision(other_size) ∧ x→getType() ≡ "Bridge") {
            bridge = true;
            break;
        }
    }
    return bridge;
}

std::string &Map::get_map() {
    return xml;
}

std::vector<Occupant *> &Map::getOccupants() {
    return *this→all_occupants;
}

void Map::updateOccupants(std::vector<Occupant *> *all_occupants) {
    this→all_occupants = all_occupants;
}

Occupant* Map::checkForEnemiesOn(Size &range, Occupant& unit) {
    for(auto x: *all_occupants) {
        if(x→getId() ≠ unit.getId() ∧ x→isThereACollision(range)
           ∧ (types[x→getType()] ≡ UNIT ∨ types[x→getType()] ≡ VEHICLE)
           ∧ x→getTeam() ≠ "Neutral" ∧ unit.getTeam() ≠ x→getTeam()) {
            return x;
        }
    }
    return &unit;
}

void Map::buildTypeMap() {
    types.insert(std::pair<std::string,std::string>("Fort",BUILDING));
    types.insert(std::pair<std::string,std::string>
                         ("vehiculeFactory",BUILDING));
    types.insert(std::pair<std::string,std::string>("robotFactory",BUILDING));
    types.insert(std::pair<std::string,std::string>("Factory",BUILDING));
    types.insert(std::pair<std::string,std::string>("Rock",NATURE));
    types.insert(std::pair<std::string,std::string>("iceblock",NATURE));
    types.insert(std::pair<std::string,std::string>("grunt",UNIT));
    types.insert(std::pair<std::string,std::string>("Psycho",UNIT));
    types.insert(std::pair<std::string,std::string>("Tough",UNIT));
    types.insert(std::pair<std::string,std::string>("Pyro",UNIT));
    types.insert(std::pair<std::string,std::string>("Sniper",UNIT));
    types.insert(std::pair<std::string,std::string>("laser",UNIT));
    types.insert(std::pair<std::string,std::string>("jeep",VEHICLE));
    types.insert(std::pair<std::string,std::string>("MediumTank",VEHICLE));
    types.insert(std::pair<std::string,std::string>("LightTank",VEHICLE));
    types.insert(std::pair<std::string,std::string>("HeavyTank",VEHICLE));
    types.insert(std::pair<std::string,std::string>("MML",VEHICLE));
    types.insert(std::pair<std::string,std::string>("flag",FLAG));
}

bool Map::tellIfItIsGrabbable(std::string& type) {
    std::string tmp = type;
    return types[tmp] ≡ VEHICLE;
}

bool Map::tellIfItIsBuilding(std::string &type) {
    std::string tmp = type;
    return types[tmp] ≡ BUILDING;
}

```

```
#include <iostream>
#include <pugixml.hpp>
#include "server.h"


unsigned int load_port(const char* cfg_file_path) {
    pugi::xml_document doc;
    pugi::xml_parse_result result = doc.load_file(cfg_file_path);
    if (¬result) {
        std::cout << "Error reading cfg file: " << result.description()
                  << std::endl;
        return 0;
    }
    pugi::xml_node port_node = doc.child("Config").child("Port");
    std::string port = port_node.attribute("port").value();
    if (¬port.size()) {
        std::cout << "Error reading port from cfg file!" << std::endl;
        return 0;
    }
    int a = std::stoi(port);
    return (unsigned int) a;
}

int main (int argc, char **argv) {
    if (argc < 2) {
        std::cout << "Usage: ./Z_Server <cfg file>" << std::endl;
        return 1;
    }

    std::string cfg = argv[1];
    if (¬cfg.size()) {
        std::cout << "Error loading cfg file" << std::endl;
        return 1;
    }
    unsigned int port = load_port(argv[1]);
    if (¬port) {
        return 1;
    }

    try {
        Menu menu(cfg);

        Server server_accepter(port, menu);
        server_accepter.start();
        char exit = 'a';

        while (exit ≠ 'q') {
            std::cin>>exit;
        }

        server_accepter.stop();
        server_accepter.join();

        menu.shutDown();

        return 0;
    } catch (SocketError& e) {
        std::cout<< e.what();
        return 1;
    }
}

```

```
//
// Created by rodian on 29/05/17.
//

#ifndef Z_TPGRUPAL_LOBBY_H
#define Z_TPGRUPAL_LOBBY_H


#include "player.h"
#include "game.h"
#include "MapLoader.h"
class Game;
class Player;

class Lobby {
private:
    int lobby_id;
    bool all_ready, game_started;
    std::vector<Player*> players;
    Game* game;
    std::vector<std::vector<std::string>> teams;
    std::map<std::string, std::string> maps;
    std::mutex &m;
    std::string& config;
public:
    Lobby(int id, std::string& config, std::mutex &m);

    bool addPlayer(Player* player);

    void startGame(const std::string& map_name);

    void ready();

    std::vector<std::string> get_player_names();

    void unReady();

    int get_id();

    void exitLobby(Player* player);

    std::string get_loaded_maps();

    void load_maps();

    void shutDown();

    bool haveGameFinished();

    void disconectPlayer( Player *player);

    ~Lobby();
};


#endif //Z_TPGRUPAL_LOBBY_H
```

```
//
// Created by rodian on 29/05/17.
//

#include <sstream>
#include "lobby.h"

Lobby::Lobby(int id, std::string& config, std::mutex &m) : lobby_id(id),
                                                            config(config),
                                                            all_ready(false),
                                                            game_started(false),
                                                            m(m){
    load_maps();
}

void Lobby::startGame(const std::string& map_name) {
    Lock l(m);
    if(all_ready) {
        //start game
        game_started = true;
        auto path_it = maps.find(map_name);
        if (path_it ≠ maps.end()) {
            std::string path = path_it→second;

            // build teams
            std::vector<Team> teams_info;
            for (int i = 0; i < teams.size(); ++i) {
                std::vector<PlayerInfo> playersInfo;
                for (int j = 0; j < teams[i].size(); ++j) {
                    PlayerInfo new_player(teams[i][j]);
                    for (auto p: players) {
                        if (p→getId() ≡ teams[i][j]) {
                            new_player.addMessenger(p→getMessenger());
                            playersInfo.push_back(new_player);
                        }
                    }
                }
                if (¬playersInfo.empty()) {
                    Team new_team(playersInfo, i);
                    teams_info.push_back(new_team);
                }
            }

            for (auto p: players) {
                p→getInGame();
            }

            game = new Game(path, config, teams_info, players);
            game→start();
        }
    }
}

void Lobby::ready() {
    Lock l(m);
    if (players.size() ≥ 2) {
        bool any_not_ready = false;
        for (auto p: players) {
            if (¬p→areYouReady()) {
                any_not_ready = true;
            }
        }
        if (¬any_not_ready)
            all_ready = true;
    }
}
```

```

bool Lobby::addPlayer(Player* player) {
    bool added = false;
    if (¬game_started) {
        if (players.size() < 4) {
            players.push_back(player);
            added = true;
        }

        std::string names_cmd = "names−";
        for (std::string name : get_player_names()) {
            names_cmd += name + "−";
        }

        for (Player *p : players) {
            p→getMessenger()→sendMessage(names_cmd);

        }
        teams.push_back(std::vector<std::string>());
        teams.back().push_back(player→getId());
        return added;
    } else {
        return added;
    }
}

std::vector<std::string> Lobby::get_player_names() {
    std::vector<std::string> names;
    for (Player* p : players) {
        names.push_back(p→getId());
    }
    return names;
}

int Lobby::get_id() {
    return lobby_id;
}

void Lobby::unReady() {
    Lock l(m);
    all_ready = false;
}

void Lobby::exitLobby(Player *player) {
    Lock l(m);
    std::vector<Player *>::iterator it = players.begin();
    for (; it ≠ players.end(); ++it) {
        if ((*it)→getId() ≡ player→getId()) {
            players.erase(it);
            break;
        }
    }

    for (auto& t: teams) {
        std::vector<std::string>::iterator ito = t.begin();
        for (;ito ≠ t.end(); ++ito) {
            if (*ito ≡ player→getId()) {
                t.erase(ito);
                break;
            }
        }
    }

    std::string names_cmd = "names−";
    for (std::string name : get_player_names()) {
        names_cmd += name + "−";
```

```
    }

    for(Player* p : players) {
        p→getMessenger()→sendMessage(names_cmd);
    }
}

std::string Lobby::get_loaded_maps() {
    std::stringstream s;
    s << "mapsinfo−";
    for (auto map : maps) {
        s << map.first << "−";
    }
    return s.str();
}


void Lobby::load_maps() {
    pugi::xml_document doc;
    doc.load_file(config.c_str());
    pugi::xml_node cfg_root = doc.child("Config");
    pugi::xml_node maps_node = cfg_root.child("Maps");
    for (pugi::xml_node map : maps_node.children()) {
        maps[map.attribute("name").value()] = map.attribute("path").value();
    }
}

void Lobby::shutDown() {
    game→shutDownGame();
    game→join();
}

Lobby::~Lobby() {}

bool Lobby::haveGameFinished() {
    if (game_started) {
        return game→gameHaveFinished();
    } else {
        return false;
    }
}

void Lobby::disconectPlayer(Player *player) {
    std::vector<Player*>::iterator it = players.begin();
    for(;it ≠ players.end();++it) {
        if ((*it)→getId() ≡ player→getId()) {
            if (game_started)
                this→game→disconectPlayer(player→getId());
            players.erase(it);
            break;
        }
    }
}
```

```
//
// Created by rodian on 29/05/17.
//

#ifndef Z_TPGRUPAL_GAME_H
#define Z_TPGRUPAL_GAME_H


#include "map.h"
#include "controlUnit.h"
#include "player.h"
#include "commandMonitor.h"

class ControlUnit;
class Player;

class Game: public Thread {
private:
    std::mutex m;
    CommandMonitor commands;
    std::map<int,Unit*> all_units;
    std::vector<Player*> players;
    std::vector<Occupant*> all_occupants;
    std::vector<Territory*> territories;
    std::vector<Team> teams;
    ControlUnit* control;
    std::shared_ptr<Map> map;
    std::string path, config;
    std::map<std::string,std::string> types;
    std::vector<UnitMold*> unit_molds;
    bool finished;

public:
//    Game(std::vector<Player *> players, std::vector<Messenger *> msgr,
//             std::shared_ptr<Map> map, std::map<int, Unit *> units,
//             std::vector<Team>& teams_info, std::vector<Occupant *> occupants,
//             std::vector<Territory *> &territories);

    Game(std::string path, std::string &config, std::vector<Team> &teams_info,
             std::vector<Player *> &players);

    void run();

    void shutDownGame();

    void sendMapInfo(ControlUnit &control);

    void sendOccupantsInfo();

    bool gameHaveFinished();

    void disconectPlayer(std::string player);

    ~Game();
private:
    void analyseOccupantsInfo(std::string& info);

    void buildTypeMap();

    void sincronizeOccupants();

    void sendTerritoryInfo();

    void createStartingUnits();

    void buildMap();
```

```

    std::vector<Messenger*> getMessengers();
};


#endif //Z_TPGRUPAL_GAME_H
```

```
//
// Created by rodian on 29/05/17.
//

#include <sstream>
#include <string>
#include "game.h"

Game::Game(std::string path, std::string &config, std::vector<Team> &teams_info,
           std::vector<Player *> &players) :
        commands(m),teams(teams_info), path(path), config(config),
        players(players), finished(false) {}

void Game::run() {
    this→buildMap();
    this→sincronizeOccupants();
    std::vector<Messenger*> messengers = getMessengers();
    control = new ControlUnit(messengers,all_units,all_occupants,
                        teams,commands,territories);
    this→sendMapInfo(*control);
    this→buildTypeMap();
    this→sendTerritoryInfo();
    this→sendOccupantsInfo();
    control→run();

    for (auto& m: unit_molds) {
        delete(m);
    }
    finished = true;
}

void Game::buildMap() {
    MapLoader maploader(path,config);
    map = maploader.get_map();

    all_occupants = map→getOccupants();

    // add a Fortress to each player
    std::vector<Factory*> forts = maploader.get_forts();
    unit_molds = forts.back()→getMolds();
    for (auto& t: teams) {
        std::vector<PlayerInfo>& playersInfo = t.getPlayersInfo();
        for (auto& p: playersInfo) {
            Factory* fortress = forts.back();
            fortress→changeTeam(p.getPlayerId());
            // set changed boolean to false
            fortress→haveYouChanged();
            p.addFortress(fortress);
            all_occupants.push_back((Occupant*) fortress);
            forts.pop_back();
        }
    }
    territories = maploader.get_territories();
}

void Game::shutDownGame() {
    if (¬finished)
        control→finishGame();
}

void Game::sendOccupantsInfo() {
    std::string info;
    this→analyseOccupantsInfo(info);
    for(auto& player : players) {
        player→getMessenger()→sendMessage(info);
    }
```

```
}

void Game::sendMapInfo(ControlUnit &control) {
    std::string& map_str = map.get()→get_map();
    for(auto& player : players) {
        std::string msg = "loadmap-" + map_str;
        player→getMessenger()→sendMessage(msg);
        player→addControlUnit(&control, &commands);
    }
}

void Game::analyseOccupantsInfo(std::string& info) {
    for (auto o: all_occupants) {
        if (types[o→getType()] ≡ "Nature") {
            info += "addnature-";
        } else if (types[o→getType()] ≡ "Building") {
            info += "addbuilding-";
        } else if (types[o→getType()] ≡ "Unit") {
            info += "addunit-";
        }
        info += std::to_string(o→getId()) + "-";
        Position pos = o→getPosition();
        info += std::to_string(pos.getX()) + "-";
        info += std::to_string(pos.getY()) + "-";
        info += o→getType() + "-";
        info += o→getTeam() + "-";
        info += std::to_string(o→getLifeLeft()) + "|";  // life left == max hp
    }
}

void Game::buildTypeMap() {
    types.insert(std::pair<std::string,std::string>("Fort","Building"));
    types.insert(std::pair<std::string,std::string>
                         ("vehiculeFactory","Building"));
    types.insert(std::pair<std::string,std::string>("robotFactory","Building"));
    types.insert(std::pair<std::string,std::string>("Factory","Building"));
    types.insert(std::pair<std::string,std::string>("Rock","Nature"));
    types.insert(std::pair<std::string,std::string>("iceblock","Nature"));
    types.insert(std::pair<std::string,std::string>("grunt","Unit"));
    types.insert(std::pair<std::string,std::string>("Psycho","Unit"));
    types.insert(std::pair<std::string,std::string>("Tough","Unit"));
    types.insert(std::pair<std::string,std::string>("Pyro","Unit"));
    types.insert(std::pair<std::string,std::string>("Sniper","Unit"));
    types.insert(std::pair<std::string,std::string>("laser","Unit"));
    types.insert(std::pair<std::string,std::string>("jeep","Unit"));
    types.insert(std::pair<std::string,std::string>("MediumTank","Unit"));
    types.insert(std::pair<std::string,std::string>("LightTank","Unit"));
    types.insert(std::pair<std::string,std::string>("HeavyTank","Unit"));
    types.insert(std::pair<std::string,std::string>("MML","Unit"));
}

void Game::sincronizeOccupants() {
    for (auto& t: territories) {
        std::map<int, Factory*>& factories = t→getFactories();
        for (auto& f: factories) {
            (f.second)→resetSelectedUnit();
            all_occupants.push_back((Occupant*)(f.second));
        }
        for (auto& team: teams) {
            std::vector<PlayerInfo>& players = team.getPlayersInfo();
            for (auto& p : players) {
                Factory* fortress = p.getFortress();
                for (auto& f: factories) {
                    if (fortress→getId() ≡ f.second→getId()) {
                        std::string new_team = p.getPlayerId();
                        t→grabFlag(new_team);
                        p.addTerritory(t);
                    }
                }
            }
        }
    }
    map→updateOccupants(&all_occupants);
    this→createStartingUnits();
}

void Game::sendTerritoryInfo() {
    std::stringstream info;
    for (auto& t: territories) {
        info << "updateterritory−";
        info << std::to_string(t→getId()) << "−"  << t→getTeam() << "−";
        Position flag = t→getFlag()→getPosition();
        info << std::to_string(flag.getX()) << "−" <<
                std::to_string(flag.getY()) << "|";
    }
    for(auto& player : players) {
        player→getMessenger()→sendMessage(info.str());
    }
}

void Game::createStartingUnits() {
    for (auto& team: teams) {
        std::vector<PlayerInfo>& players = team.getPlayersInfo();
        for (auto& p : players) {
            Factory* fortress = p.getFortress();
            int id_counter = (int)(territories.size() + all_occupants.size());
            fortress→createStartingUnits(id_counter);
            std::vector<Unit *> tmp = fortress→getUnits();
            for (auto &u: tmp) {
                u→recalculateMyStartPosition();
                all_units[u→getId()] = u;
                all_occupants.push_back((Occupant*)u);
                // set changed boolean to false
                u→haveYouChanged();
            }
        }
    }
}

std::vector<Messenger *> Game::getMessengers() {
    std::vector<Messenger *> messengers;
    for (auto& p: players) {
        messengers.push_back(p→getMessenger());
    }
    return messengers;
}

Game::~Game() {
    delete (this→control);
}

bool Game::gameHaveFinished() {
    return finished;
}

void Game::disconectPlayer(std::string id_player) {
    for (auto& t: teams) {
        std::vector<PlayerInfo>& players_info = t.getPlayersInfo();
        std::vector<PlayerInfo>::iterator it = players_info.begin();
        for (; it ≠ players_info.end(); ++it) {
            if ((*it).getPlayerId() ≡ id_player) {
                players_info.erase(it);
```

```
                break;
            }
        }
    }
    std::vector<Player*>::iterator it = players.begin();
    for(;it ≠ players.end();++it) {
        if ((*it)→getId() ≡ id_player) {
            players.erase(it);
            break;
        }
    }
}
```

```
//
// Created by rodian on 22/05/17.
//

#ifndef Z_TPGRUPAL_FACTORY_H
#define Z_TPGRUPAL_FACTORY_H
#include "unitMold.h"
#include <memory>
class Factory: public Occupant {
    bool running;
    int tech_level,time_counter;
    std::vector<UnitMold*> units;
    std::vector<UnitMold*>::iterator mold;
    std::vector<Unit*> new_units;
    std::shared_ptr<Map> map;
    std::map<std::string, Weapon> weapons;


public:
    Factory(int id, int life, std::string& type, Size position,
                std::vector<UnitMold*>& units, std::shared_ptr<Map> map,
                std::map<std::string, Weapon> &weapons);

    // starts the creation of the selected unit
    void build(int& id_counter);

    void startBuilding(const std::string& player_id);

    // Returns the creational time of the selected Unit
    int getSelectedUnitTime();

    // Returns the type of the unit that now is selected
    UnitMold* nextUnit();

    UnitMold* previousUnit();

    UnitMold * getSelectedUnit();

    void changeTechLevel(int tech_level);

    int getCreationSpeed();

    bool haveNewUnits();

    std::vector<Unit*> getUnits();

    void resetSelectedUnit();

    void createStartingUnits(int &id_counter);

    std::vector<UnitMold*> getMolds();

    ~Factory();
};


#endif //Z_TPGRUPAL_FACTORY_H
```

```
//
// Created by rodian on 22/05/17.
//

#include <memory>
#include "factory.h"
#define ID 3
Factory::Factory(int id, int life, std::string& type, Size position,
                 std::vector<UnitMold*>& units, std::shared_ptr<Map> map,
                 std::map<std::string, Weapon> &weapons) :
Occupant(id, life,type, position), running(false),time_counter(0), units(units),
map(map), weapons(weapons), tech_level(1){
    mold = units.begin();
}

void Factory::build(int& id_counter) {
    if (time_counter ≥ (*mold)→getCreationTime()) {
        Size u_size = (*mold)→getUnitSize();
        Position factory_pos = this→obj_size.getPosition();
        u_size.moveTo(factory_pos.getX(),factory_pos.getY());
        Weapon u_weapon = weapons.at((*mold)→getWeaponType());

        for (int i = 0; i < (*mold)→getCreationQuantity(); ++i) {
            Unit* new_unit = (*mold)→createUnit(
                    id_counter,u_size,*map,u_weapon);
            new_unit→changeTeam(this→team);
            new_units.push_back(new_unit);
            ++id_counter;
        }
        this→changed = true;
        time_counter = 0;
    } else if (running ∧ time_counter < (*mold)→getCreationTime()) {
        time_counter += 1 + tech_level;
        this→changed = true;
    } else {
        this→changed = false;
    }
}

int Factory::getSelectedUnitTime() {
    return (*mold)→getCreationTime();
}

UnitMold* Factory::nextUnit() {
    int i = 0;
    this→running = false;
    time_counter = 0;
    while (i ≡ 0 ∨ (*mold)→getTechnologyLevel() > this→tech_level) {
        ++mold;
        if (mold ≡ units.end())
            mold = units.begin();
        ++i;
    }
    return *mold;
}

void Factory::changeTechLevel(int tech_level) {
    this→tech_level = tech_level;
}

void Factory::startBuilding(const std::string &player_id) {
    if (player_id ≡ this→getTeam())
        running = true;
}

int Factory::getCreationSpeed() {
```

```cpp
    return (((*mold)→getCreationTime() − time_counter) / (1 + tech_level));
}

bool Factory::haveNewUnits() {
    return (¬this→new_units.empty());
}

std::vector<Unit*> Factory::getUnits() {
    std::vector<Unit*> tmp = new_units;
    new_units.clear();
    return tmp;
}

UnitMold * Factory::getSelectedUnit() {
    return *mold;
}

UnitMold* Factory::previousUnit() {
    int i = 0;
    this→running = false;
    time_counter = 0;
    while (i ≡ 0 ∨ (*mold)→getTechnologyLevel() > this→tech_level) {
        if (mold ≡ units.begin()) {
            mold = units.end();
        }
        −−mold;
        ++i;
    }
    return *mold;
}

void Factory::resetSelectedUnit() {
    mold = units.begin();
}

void Factory::createStartingUnits(int &id_counter) {
    time_counter = (*mold)→getCreationTime();
    this→build(id_counter);
}

Factory::~Factory() {}

std::vector<UnitMold *> Factory::getMolds() {
    return units;
}

```

```cpp
//
// Created by rodian on 22/05/17.
//

#ifndef Z_TPGRUPAL_CONTROUNIT_H
#define Z_TPGRUPAL_CONTROUNIT_H

#include <iostream>
#include <chrono>
#include <thread>
#include "unit.h"
#include "../common/Lock.h"
#include "../common/messenger.h"
#include "factory.h"
#include "territory.h"
#include "team.h"
#include "command.h"
#include "commandMonitor.h"

class Command;
class CommandMonitor;

class ControlUnit {
private:
    std::map<int,Unit*>& all_units;
    std::vector<Territory*>& territories;
    std::vector<Occupant*>& all_occupants;
    std::vector<Messenger*> players;
    CommandMonitor &commands;
    std::mutex m;
    bool winning;
    std::vector<Team>& teams;
    std::vector<Bullet*> all_bullets;
    std::vector<int> changed_units;
    std::vector<Occupant> changed_occupants;
    std::vector<Factory> changed_factories;
    std::vector<Unit*> eliminated_units;
    int objects_counter;

public:
    ControlUnit(std::vector<Messenger *> &new_players,
                std::map<int, Unit *> &all_units,
                std::vector<Occupant *> &all_occupants,
                std::vector<Team> &teams, CommandMonitor &commands,
                std::vector<Territory *>& territories);

    // Method to start checking commands from players
    void run();

    void sleepFor(std::chrono::duration<double> sec);

    // Meant to make every unit make a micro action on the Tic
    void unitsMakeMicroAction();

    // Checks if any Occupant is dead. If so, it will remove it from the game
    void checkAllLivingOccupants();

    void moveAllBullets();

    // Command move unit. Meant to give the order to the unit to start moving
    // to de (x,y) position
    void cmdMoveUnit(const std::string& id_player, int id, int x, int y);

    void cmdAttack(const std::string& attacker_team, int id_unit, int target);

    void cmdGrab(const std::string& id_player, int id_unit, int target);

    void cmdFactoryCreate(const std::string& player_id, int id_factory);

    void cmdFactoryNext(const std::string& player_id, int id_factory);

    void cmdFactoryPrev(const std::string& player_id, int id_factory);

    void cmdFactoryCurrent(const std::string& player_id, int id_factory);

    void finishGame();

private:
    // Process all commands on commands vector and leaves the vector empty
    void executeCommands();

    void sendUpdateMessage();

    void sendMessageTo(const std::string& player_id, std::string& msg);

    std::string getUpdateInfo();

    std::string getInfoFromUnit(Unit& unit);

    std::string getInfoFromOccupant(Occupant& Occupant);

    std::string getInfoFromBullets(Bullet& bullet);

    std::string getInfoFromFactories(Factory& factory);

    std::string getInfoFromUnitMold(UnitMold& mold, double time);

    std::string getInfoForAddUnit(Unit& unit);

    void makeTerritoriesChecks();

    void makeFactoryChecks();

    void checkForWinner();

    void sendFinnalMessage();

    void getTime(int& minutes, int& seconds, double time);

    void freeMemory();
};


#endif //Z_TPGRUPAL_CONTROUNIT_H
```

```cpp
//
// Created by rodian on 22/05/17.
//

#include "controlUnit.h"
#define WAIT 0.3
#define FLAG "flag"
#define NATURE "Rock"

ControlUnit::ControlUnit(std::vector<Messenger *> &new_players,
                         std::map<int, Unit *> &all_units,
                         std::vector<Occupant *> &occupants,
                         std::vector<Team> &teams, CommandMonitor &commands,
                         std::vector<Territory *>& territories) :
    all_units(all_units), territories(territories),
    all_occupants(occupants), players(new_players), commands(commands),
    winning(false), teams(teams) {
}

void ControlUnit::run() {
    objects_counter = (int)all_occupants.size();
    while(¬winning) {
        std::chrono::duration<double> t3(WAIT);

        auto t1 = std::chrono::high_resolution_clock::now();

        // execute commands
        executeCommands();

        // do stuff
        this→moveAllBullets();
        this→unitsMakeMicroAction();
        this→makeTerritoriesChecks();
        this→checkAllLivingOccupants();

        //send update message
        this→sendUpdateMessage();

        this→checkForWinner();

        auto t2 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> time_span = t3 − (t2 − t1);
        sleepFor(time_span);
        changed_units.clear();
        changed_occupants.clear();
        changed_factories.clear();
        for (auto& u: eliminated_units) {
            delete(u);
        }
        eliminated_units.clear();
    }
    // send victory or defeated message
    this→sendFinnalMessage();
    this→freeMemory();
}

void ControlUnit::sleepFor(std::chrono::duration<double> msec) {
    std::this_thread::sleep_for((msec));
}

void ControlUnit::unitsMakeMicroAction() {
    // erase units with life 0
    std::vector<int> units_id;
    for (auto& x: all_units) {
        Unit& unit = *x.second;
        if (unit.doYouNeedToDisappear()) {
            units_id.push_back(x.first);
        }
    }
    for (auto& id: units_id) {
        eliminated_units.push_back(all_units[id]);
        all_units.erase(id);
    }

    // units alive make micro action
    for (auto& x: all_units){
        Unit& unit = *x.second;
        // check if someone changed the unit
        bool was_changed = false;
        if (unit.haveYouChanged()) {
            changed_units.push_back(unit.getId());
            was_changed = true;
        }
        unit.makeAction();
        // check if the unit changed
        if (unit.haveYouChanged() ∧ ¬was_changed) {
            changed_units.push_back(unit.getId());
        }
        if (¬unit.areYouAlive()) {
            unit.mustDisappear();
        } else if (unit.doYouHaveAnyBullets()) {
            std::vector<Bullet *> tmp = unit.collectBullets();
            for (auto& b: tmp) {
                b→setCorrectId(objects_counter);
                ++objects_counter;
            }
            all_bullets.insert(all_bullets.end(), tmp.begin(), tmp.end());
        }
    }
}


void ControlUnit::checkAllLivingOccupants() {
    std::vector<Occupant*>::iterator it = all_occupants.begin();
    for(;it ≠ all_occupants.end();){
        if((*it)→doYouNeedToDisappear()) {
            //erase it from map
            if ((*it)→getType() ≡ NATURE) {
                delete((*it));
            }
            it = all_occupants.erase(it);
            // if building put ruins
        } else {
            if((*it)→haveYouChanged())
                changed_occupants.push_back(*(*it));
            if(¬(*it)→areYouAlive()) {
                (*it)→mustDisappear();
            }
            ++it;
        }
    }
}

void ControlUnit::makeTerritoriesChecks() {
    for (auto& t: territories) {
        if (t→doesTerritorysOwnerChanged()) {
            std::string info = "updateterritory−";
            info += std::to_string(t→getId()) + "−" + t→getTeam() + "−";
            Position flag = t→getFlag()→getPosition();
            info += std::to_string(flag.getX()) + "−" +
                    std::to_string(flag.getY());
            for (auto& team: teams) {
```

```
                std::vector<PlayerInfo>& players = team.getPlayersInfo();
                for (auto& p : players) {
                    // the last owner must eliminate the territory from his
                    // vector
                    p.eliminateThisTerritory(t);

                    // the new owner must add it
                    if (t→getTeam() ≡ p.getPlayerId()) {
                        p.addTerritory(t);
                    }
                }
            }
            for (auto y: players) {
                y→sendMessage(info);
            }
        }
    }
    makeFactoryChecks();
}

void ControlUnit::makeFactoryChecks() {
    for (auto t: territories) {
        std::map<int,Factory*>& factories = t→getFactories();
        auto it = factories.begin();
        // vector to know witch factories erase
        std::vector<int> factories_id;
        for (; it ≠ factories.end();) {
            Factory *f = it→second;
            bool was_changed = false;
            if (f→haveYouChanged()) {
                changed_factories.push_back(*f);
                was_changed = true;
            }
            if (f→areYouAlive()) {
                f→build(objects_counter);
                // check if the factory changed
                if (f→haveYouChanged() ∧ ¬was_changed) {
                    changed_factories.push_back(*f);
                }
                if (f→haveNewUnits()) {
                    std::vector<Unit *> tmp = f→getUnits();
                    std::string msg = "";
                    for (auto &u: tmp) {
                        u→recalculateMyStartPosition();
                        all_units[u→getId()] = u;
                        all_occupants.push_back((Occupant*)u);
                        // set changed boolean to false
                        u→haveYouChanged();
                        msg += "addunit−";
                        msg += getInfoForAddUnit(*u);
                    }
                    for (auto y: players) {
                        y→sendMessage(msg);
                    }
                }
            }
            ++it;
        }
        for (auto& fact: factories_id) {
            factories.erase(fact);
        }
    }
}

void ControlUnit::cmdMoveUnit(const std::string& id_player,int id, int x,
                              int y) {
```

```
    std::map<int,Unit*>::iterator it;
    it = all_units.find(id);
    if ((*it→second).getTeam() ≡ id_player)
        (*it→second).calculateRoadTo(x,y);
}

void ControlUnit::cmdAttack(const std::string& attacker_team, int id_unit,
                            int target) {
    std::map<int,Unit*>::iterator it;
    it = all_units.find(id_unit);
    if ((*it→second).getTeam() ≡ attacker_team) {
        for (auto z: all_occupants) {
            if (z→getId() ≡ target) {
                if (z→getTeam() ≠ attacker_team) {
                    (*it→second).setTargetToAttack(z);
                    break;
                }
            }
        }
    }
}

void ControlUnit::cmdGrab(const std::string &id_player, int id_unit,
                          int target) {
    std::map<int,Unit*>::iterator it;
    it = all_units.find(id_unit);
    Unit& unit = (*it→second);
    bool found = false;
    if (unit.getTeam() ≡ id_player) {
        for (auto t: territories) {
            if (t→getId() ≡ target) {
                unit.setTargetToGrab(t→getFlag(),FLAG);
                found = true;
            }
        }
        if (¬found) {
            for (auto& z: all_occupants) {
                if (z→getId() ≡ target) {
                    unit.setTargetToGrab(z, z→getType());
                }
            }
        }
    }
}

void ControlUnit::cmdFactoryCreate(const std::string& player_id,
                                   int id_factory) {
    for (auto t: territories) {
        std::map<int, Factory *> &factories = t→getFactories();
        for (auto& f: factories) {
            if (f.first ≡ id_factory ∧ f.second→areYouAlive()) {
                f.second→startBuilding(player_id);
            }
        }
    }
}

void ControlUnit::cmdFactoryNext(const std::string &player_id, int id_factory) {
    std::string info = "";
    for (auto t: territories) {
        std::map<int, Factory *> &factories = t→getFactories();
        for (auto& f: factories) {
            if (f.first ≡ id_factory ∧ f.second→getTeam() ≡ player_id
                                      ∧ f.second→areYouAlive()) {
                UnitMold* mold = f.second→nextUnit();
                info += "factorystats−";
```

```
                int creation_time = f.second→getCreationSpeed();
                info += getInfoFromUnitMold(*mold,creation_time);
                break;
            }
        }
    }
    sendMessageTo(player_id,info);
}

void ControlUnit::cmdFactoryPrev(const std::string &player_id, int id_factory) {
    std::string info = "";
    for (auto t: territories) {
        std::map<int, Factory *> &factories = t→getFactories();
        for (auto& f: factories) {
            if (f.first ≡ id_factory ∧ f.second→getTeam() ≡ player_id
                                      ∧ f.second→areYouAlive()) {
                UnitMold* mold = f.second→previousUnit();
                info += "factorystats−";
                int creation_time = f.second→getCreationSpeed();
                info += getInfoFromUnitMold(*mold,creation_time);
                break;
            }
        }
    }
    sendMessageTo(player_id,info);
}


void ControlUnit::cmdFactoryCurrent(const std::string &player_id,
                                    int id_factory) {
    std::string info = "";
    for (auto t: territories) {
        std::map<int, Factory *> &factories = t→getFactories();
        for (auto& f: factories) {
            if (f.first ≡ id_factory ∧ f.second→areYouAlive()) {
                UnitMold* mold = f.second→getSelectedUnit();
                info += "factorystats−";
                int creation_time = f.second→getCreationSpeed();
                info += getInfoFromUnitMold(*mold,creation_time);
                break;
            }
        }
    }
    sendMessageTo(player_id,info);
}


void ControlUnit::executeCommands() {
    std::vector<Command> commands_copy;
    commands.copyCommands(commands_copy);

    // Execute command
    for (auto cmd: commands_copy) {
        cmd();
    }
}

void ControlUnit::sendMessageTo(const std::string& player_id,
                                std::string& msg) {
    bool found = false;
    for (auto& t: teams) {
        std::vector<PlayerInfo>& plyrs = t.getPlayersInfo();
        for (auto& p: plyrs) {
            if (p.getPlayerId() ≡ player_id) {
                p.getMessenger()→sendMessage(msg);
                found = true;
```

```cpp
                break;
            }
        }
        if (found)
            break;
    }
}

void ControlUnit::sendUpdateMessage() {
    std::string info = getUpdateInfo();
    if (¬info.size()) {
        return;
    }
    for (auto y: players) {
        y→sendMessage(info);
    }
}

std::string ControlUnit::getUpdateInfo() {
    std::string  update_msg = "";
    for (auto z: changed_units) {
        update_msg += "updateunit−";
        update_msg += getInfoFromUnit(*all_units.at(z));
    }

    for (auto y: changed_occupants) {
        update_msg += "updateoccupant−";
        update_msg += getInfoFromOccupant(y);
    }

    for (auto& f: changed_factories) {
        update_msg += "updatefactory−";
        update_msg += getInfoFromFactories(f);
    }

//    for (auto b: all_bullets) {
//        update_msg += "updatebullet−";
//        update_msg += getInfoFromBullets(*b) ;
//    }

    return update_msg;
}

std::string ControlUnit::getInfoFromUnit(Unit &unit) {
    std::string info = "";
    info += std::to_string(unit.getId()) + "−";
    info += unit.getActionState() + "−";
    info += std::to_string(unit.getCurrentPosition().getX()) + "−";
    info += std::to_string(unit.getCurrentPosition().getY()) + "−";
    info += std::to_string(unit.getLifeLeft()) + "−";
    info += unit.getTeam() + "|";
    return info;
}

std::string ControlUnit::getInfoFromOccupant(Occupant& Occupant) {
    std::string info = "";
    info += std::to_string(Occupant.getId()) + "−";
    info += std::to_string(Occupant.getPosition().getX()) + "−";
    info += std::to_string(Occupant.getPosition().getY()) + "−";
    info += std::to_string(Occupant.getLifeLeft()) + "|";
    return info;
}

std::string ControlUnit::getInfoFromFactories(Factory &factory) {
    std::string info = "";
    info += std::to_string(factory.getId()) + "−";
```

```cpp
    // This is the time needed before the next unit is build in seconds
    double time = WAIT * factory.getCreationSpeed();
    int min = 0, sec = 0;
    getTime(min, sec, time);
    info += std::to_string(min)+ "−" +std::to_string(sec) + "−";
    info += std::to_string(factory.getLifeLeft()) + "−";
    info += factory.getTeam() + "|";
    return info;
}

std::string ControlUnit::getInfoFromBullets(Bullet &bullet) {
    std::string info = "";
    info +=  bullet.getType() + "−";
    info += std::to_string(bullet.getId()) + "−";
    info += std::to_string(bullet.getPosition().getX()) + "−";
    info += std::to_string(bullet.getPosition().getY()) + "|";
    return info;
}

std::string ControlUnit::getInfoFromUnitMold(UnitMold &mold,
                                             double creation_time) {
    std::string info = "";
    info += mold.getTypeOfUnit() + "−";
    info += std::to_string(mold.getFireRate()) + "−";
    double time = WAIT * creation_time;
    int min = 0, sec = 0;
    getTime(min, sec, time);
    info += std::to_string(min)+ "−" +std::to_string(sec) + "−";
    info += std::to_string(mold.getLife()) + "|";
    return info;
}

std::string ControlUnit::getInfoForAddUnit(Unit &unit) {
    std::string info = "";
    info += std::to_string(unit.getId()) + "−";
    Position pos = unit.getPosition();
    info += std::to_string(pos.getX()) + "−";
    info += std::to_string(pos.getY()) + "−";
    info += unit.getType() + "−";
    info += unit.getTeam() + "−";
    info += std::to_string(unit.getLifeLeft()) + "|";
    return info;
}

void ControlUnit::moveAllBullets() {
    std::vector<Bullet*>::iterator it = all_bullets.begin();
    for (; it ≠ all_bullets.end();) {
        (*it)→move();
        if ((*it)→doYouHaveToDisapear()) {
            delete((*it));
            it = all_bullets.erase(it);
        } else {
            if ((*it)→didHit())
                (*it)→disapear();
            ++it;
        }
    }
}


void ControlUnit::checkForWinner() {
    int teams_alive = 0;
    for (auto t: teams) {
        if (¬t.doesTeamLose()) {
            teams_alive += 1;
        } else {
            std::vector<PlayerInfo>& losers = t.getPlayersInfo();
            for (auto& w: losers) {
                w.getMessenger()→sendMessage("loseryousuck");
            }
        }
    }

    if (teams_alive ≡ 1) {
        winning = true;
    }
}

void ControlUnit::sendFinnalMessage() {
    std::string winner = "winner−";
    for (auto& t: teams) {
        if (¬t.doesTeamLose()) {
            std::vector<PlayerInfo>& winners = t.getPlayersInfo();
            for (auto& w: winners) {
                w.getMessenger()→sendMessage("winner");
            }
        } else {
            std::vector<PlayerInfo>& losers = t.getPlayersInfo();
            for (auto& w: losers) {
                w.getMessenger()→sendMessage("loseryousuck");
            }
        }
    }
}

void ControlUnit::getTime(int &minutes, int &seconds, double time) {
    double min = time/60;
    minutes = (int)min;
    double sec = min − minutes;
    sec = sec * 60;
    seconds = (int) sec;
}

void ControlUnit::freeMemory() {
    // free memory
    std::vector<Occupant*>::iterator it = all_occupants.begin();
    for (;it ≠ all_occupants.end();){
        if ((*it)→getType() ≡ NATURE) {
            delete((*it));
        }
        it = all_occupants.erase(it);
    }

    for (auto& u: all_units) {
        delete(u.second);
    }
    all_units.clear();

    for (auto& t: territories) {
        delete(t);
    }

    for (auto& b: all_bullets) {
        delete(b);
    }
    all_bullets.clear();
}

void ControlUnit::finishGame() {
    winning = true;
}

```

```
//
// Created by rodian on 14/05/17.
//

#ifndef Z_TPGRUPAL_COMPASS_H
#define Z_TPGRUPAL_COMPASS_H

#include <iostream>
#include <vector>
#include <map>
#include "cell.h"
#include "node.h"
#include "map.h"

class Map;

// class Compass so every moving unit knows the fastest way to destiny
class Compass {
private:
    Map& map;
    std::vector<std::vector<Node*>> astar_map;
    std::vector<Node*> closed_nodes;
    std::vector<Node*> open_nodes;
    std::vector<Position> road;
    std::map<std::string,int> terrain_modifier;
    int unit_id, unit_speed;
    Size unit_size;
    bool finished, clear;
    Position destiny;

public:
    // The Compass receives the map of Cells for calculations and the
    // basic unit speed
    Compass(Map &map, Size &unit_size, int unit_id, int unit_speed);

    // Receives the current position of the unit and the destiny
    // Returns a vector of Cells with the fastest way
    std::vector<Position> getFastestWay(Position& from, Position& to);

    // Returns true if the position is empty
    bool canIWalkToThisPosition(Size& size);

    // Returns true if the position is empty except for the occupant
    bool canBulletWalkToThisPosition(Size& size,Occupant &shooter,
                                     Occupant& occupant);

    // Returns the Terrain factor on the (x,y) position
    double getTerrainFactorOn(int x, int y);

    void changeUnitSize(Size& unit_size);

    void changeUnitSpeed(int speed);

    void changeUnitId(int id);

    void clearCompass();

    bool checkIfItIsGrabbable(std::string& type) const;

    bool checkIfItIsABuilding(std::string& type) const;

    Occupant* checkForEnemiesOnRange(Occupant& unit, Size &range);

    // Builds a Node map with the size of the original map
    void buildNodeMap();

```

```
    // Returns the position of destiny. If destiny is not a valid position
    // it returns the closest valid position
    Position getAValidPositionForDestiny(Position& destiny);

    ~Compass();

private:

    void setTerrainModifier();
    // Writes the H value on every node of astar_map for the received position
    // It use Manhattan distance
    void setHValueForDestiny(Position& to);

    // Only valid for Manhattan distance.
    // Returns true if other is a diagonal node of reference
    // otherwise flase. Reference and other must be adjacent.
    bool isThisNodeOnDiagonal(Node* reference, Node* other);

    // Put the adyacents nodes that can be walk to on the open_nodes vector
    void getAdjacents(Node* node, int step);

    // Returns true if node and other are different nodes. Else, false
    bool isNotMe(Node* node, Node* other);

    // Puts the node in the correct order.
    // The node with lower F will be on the back
    void addToOpenInOrder(Node* node);

    // If adj node hasn’t been seen or the g value from ref node is lower
    // than previous, it chages g value and the parent.
    bool writeGandSetParent(Node *ref, Node *adj, int walk, int steps);

    // Changes the position of the node
    void changeNodePosition(Node* node);

    // Inserts the node on the correct position
    void insertNodeOnOpen(Node* node);

    void getRoad(Position& from, Node* destiny);

    // Returns the closest node on the closed_nodes vector
    Node* searchForClosestNode();

    // Returns a positive value of the result of x − y.
    int getModuleOfSubtraction(int x, int y);

    // checks the Neighbor nodes to see if destiny is among them
    void checkIfIsDestinyNeighbor(Node *new_node, int step);

    // Returns the closest valid position to pos
    Position getClosestValidPosition(Position& pos);

    void addPositions(Position& position);

    void manageSteps(int& step,Position& start,
                     Position& current_pos, Position& to);

    void setHValueOnNode(Node* node);

    void addPositionsInOrder(bool increase_x, bool increase_y, int x_max,
                             int x_min, int y_max, int y_min);

    int getModule(int x, int y);
};


```

```
#endif //Z_TPGRUPAL_COMPASS_H
```

```
//
// Created by rodian on 14/05/17.
//

#include "compass.h"

#define SIDEWALK 10
#define DIAGONALWALK 14
#define HMIN 100
#define STEP 2
#define CLOSERAREA 32
#define MIDDLEAREA 120
#define MAXLOOP 800

Compass::Compass(Map &map, Size &unit_size, int unit_id, int unit_speed)
        : map(map),
          unit_size(unit_size), unit_id(unit_id) ,unit_speed(unit_speed),
        destiny(0,0), clear(true){
    this→buildNodeMap();
    this→setTerrainModifier();
}

void Compass::setTerrainModifier() {
    terrain_modifier.insert(std::pair<std::string,int>("Carretera",1));
    terrain_modifier.insert(std::pair<std::string,int>("Camino Asfaltado",1));
    terrain_modifier.insert(std::pair<std::string,int>("Tierra",2));
    terrain_modifier.insert(std::pair<std::string,int>("Pradera",2));
    terrain_modifier.insert(std::pair<std::string,int>("Nieve",2));
    terrain_modifier.insert(std::pair<std::string, int>("Agua", 10));
}

void Compass::buildNodeMap() {
    // the nodes has the size of the unit that is using this compass
    for(int it = 0; it < map.getWidth(); ++it) {
        std::vector<Node*> row_vec;
        astar_map.push_back(row_vec);
        for(int jt = 0; jt < map.getHeigth(); ++jt) {
            astar_map.back().push_back(new Node(it, jt,
                                  unit_size.getWidth(), unit_size.getHeight()));
        }
    }
}

std::vector<Position> Compass::getFastestWay(Position& from, Position& to) {
    if (¬clear)
        clearCompass();
    // check if it’s a possible position
    destiny = getAValidPositionForDestiny(to);
    // if I’m already on the closest position return it
    if (from.getX() ≡ destiny.getX() ∧ from.getY() ≡ destiny.getY()) {
        this→road.push_back(destiny);
        return road;
    } else {
        // start algorithm
        // add "from" to visited list
        Node *start_node = astar_map[from.getX()][from.getY()];
        std::string terrain_type = map.getTerrainType(from.getX(), from.getY());
        start_node→setGValue(0, terrain_modifier[terrain_type]);
        start_node→setNewParent(start_node);
        Position start_pos = start_node→getPosition();
        this→closed_nodes.push_back(start_node);
        clear = false;

        Node *closer_node = start_node;
        // While haven’t reach destiny node or open_nodes has nodes to visit.
        finished = false;
        bool open_nodes_empty = false;

        int step = 1;
        int step_check = step;
        int i = 0;
        while (¬finished ∧ (¬open_nodes_empty)) {
            // get adjacent's and add them to looking list in order of F value.
            // On tie use H value.

            this→getAdjacents(closer_node,step);

            // if there are no adjacent's and open_node is empty, end search
            if (open_nodes.empty()) {
                open_nodes_empty = true;
            } else {
                // get the minimum F and add it to visit list
                // (remove from looking list)
                closer_node = open_nodes.back();
                open_nodes.pop_back();
                this→closed_nodes.push_back(closer_node);
                Position cls_pos = closer_node→getPosition();
                // check if destiny is between them

                if (closed_nodes.back()→getHvalue() ≡ 0)
                    finished = true;

                if (¬finished)
                    manageSteps(step,start_pos ,cls_pos,
                            destiny);
            }

            if (step_check ≠ step) {
                step_check = step;
            }
            ++i;
            if (i ≥ MAXLOOP) {
                finished = true;
                closer_node = start_node;
            }
        }
        Node *closest;
        if (finished) {
            this→getRoad(from, closer_node);
        } else {
            closest = this→searchForClosestNode();
            this→getRoad(from, closest);
        }
        finished = false;
        return road;
    }
}

void Compass::setHValueForDestiny(Position& to) {
    astar_map[to.getX()][to.getY()]→setHValue(0);

    for (auto x: astar_map) {
        for(auto y: x){
            Position tmp = y→getPosition();
            int h_value = HMIN * (this→getModuleOfSubtraction(tmp.getX(),
               to.getX()) + this→getModuleOfSubtraction(tmp.getY(),to.getY()));
            y→setHValue(h_value);
        }
    }
}

void Compass::getAdjacents(Node *node, int step) {
```

```
    // get limits
    int x_min = node→getPosition().getX() − step;
    int x_max = node→getPosition().getX() + step;
    int y_min = node→getPosition().getY() − step;
    int y_max = node→getPosition().getY() + step;

    bool adj_new_g;
    Node* adj;
    for (int x_pos = x_min; x_pos ≤ x_max;x_pos += step) {
        for (int y_pos = y_min; y_pos ≤ y_max; y_pos += step) {
            if (map.doesThisPositionExist(x_pos, y_pos)){
                adj = astar_map[x_pos][y_pos];
                Size size = adj→getSize();

                // Check if whether node fit or the position is not available.
                // Also discard the node looking for his adjacent
                if ((map.canIWalkToThisPosition(size, unit_id)) ∧
                    this→isNotMe(node, adj)) {
                    this→setHValueOnNode(adj);
                    // G value differs when the node is diagonal or next to it
                    if (this→isThisNodeOnDiagonal(node, adj)) {
                        adj_new_g = this→writeGandSetParent(node, adj,
                                                             DIAGONALWALK, 0);

                    } else {
                        adj_new_g = this→writeGandSetParent(node, adj,
                                                             SIDEWALK, 0);
                    }
                    if (adj_new_g)
                        this→addToOpenInOrder(adj);
                }
            }
            if (finished)
                break;
        }
        if (finished)
            break;
    }
}

bool Compass::isThisNodeOnDiagonal(Node* ref, Node* other) {
    Position pos_ref = ref→getPosition();
    Position pos_other = other→getPosition();
    int diff_y = getModuleOfSubtraction(pos_ref.getY(),pos_other.getY());
    int diff_x = getModuleOfSubtraction(pos_ref.getX(),pos_other.getX());
    return ((diff_x > 0) ∧ (diff_y > 0));
}

bool Compass::isNotMe(Node* node, Node* other) {
    Position ref = node→getPosition();
    Position ady = other→getPosition();
    return ¬((ref.getX() ≡ ady.getX()) ∧ (ref.getY() ≡ ady.getY()));
}

void Compass::addToOpenInOrder(Node* new_node) {
    // Only add to the vector those that haven't been seen
    if (¬new_node→beenSeen()){
        this→insertNodeOnOpen(new_node);
    } else {
        this→changeNodePosition(new_node);
    }
}

bool Compass::writeGandSetParent(Node *ref, Node *adj, int walk, int steps) {
    Size adj_size = adj→getSize();
    //calculate new g
    int new_g = walk + ref→getGValue();
    // get additional g for all steps
    road.push_back(adj→getPosition());
    Position actual = ref→getPosition();
    addPositions(actual);
    for (auto& pos: road) {
        std::string terrain_type = map.getTerrainType(pos.getX(),pos.getY());
        // when is a vehicle and it’s water, don’t add it to open list
        if (¬(unit_speed ≠ 2 ∧ terrain_type ≡ "Agua" ∧
              ¬map.thereIsABridge(adj_size))) {
            new_g += terrain_modifier[terrain_type];
        } else {
            new_g += (terrain_modifier[terrain_type] * 20);
        }
    }
    road.clear();

    bool adj_change_g = false;
    // if F value from node is lower than previous or this
    // adjacent hasn’t been seen yet,
    // add new g value and change parent.
    Position pos = adj→getPosition();
    std::string terrain_type = map.getTerrainType(pos.getX(),pos.getY());
    // when is a vehicle and it’s water, don’t add it to open list
    if (¬(unit_speed ≠ 2 ∧ terrain_type ≡ "Agua" ∧
          ¬map.thereIsABridge(adj_size))) {
        int terrain_factor = terrain_modifier[terrain_type];
        if ((adj→beenSeen() ∧
             (adj→getFValueIfGWere(new_g, terrain_factor) <
              adj→getFValue())) ∨
            (¬adj→beenSeen())) {
            adj→setGValue(new_g, terrain_factor);
            adj→setNewParent(ref);
            adj_change_g = true;
        }
    }
    return adj_change_g;
}

void Compass::changeNodePosition(Node *node) {
    // first erase node from vector
    bool erased = false;
    Position node_pos = node→getPosition();
    std::vector<Node *>::iterator it = open_nodes.begin();
    while ((¬erased) ∧ (it ≠ open_nodes.end())) {
        Position it_pos = (*it)→getPosition();
        if ((it_pos.getX() ≡ node_pos.getX()) ∧
            (it_pos.getY() ≡ node_pos.getY())){
            it = open_nodes.erase(it);
            erased = true;
        } else {
            ++it;
        }
    }
    // Add it again in correct position
    this→insertNodeOnOpen(node);
}

void Compass::insertNodeOnOpen(Node *new_node) {
    if (new_node→getHvalue() ≡ 0) {
        open_nodes.push_back(new_node);
        finished = true;
    } else {
        bool inserted = false;
        // Save nodes by F value. The lowest on the back.
        // If two nodes have same F value, the one with the lowest H value
```

```
        // will be closer to the back.
        std::vector<Node *>::iterator it = open_nodes.begin();
        while ((¬inserted) ∧ (it ≠ open_nodes.end())) {
            if (((*it)→getFValue()) < new_node→getFValue()) {
                open_nodes.insert(it, new_node);
                inserted = true;
            } else if (((*it)→getFValue()) ≡ new_node→getFValue()) {
                if (((*it)→getHvalue()) < new_node→getHvalue()) {
                    open_nodes.insert(it, new_node);
                    inserted = true;
                }
            }
            ++it;
        }
        if (¬inserted) {
            open_nodes.push_back(new_node);
        }
    }
    if (¬finished)
        this→checkIfIsDestinyNeighbor(new_node, STEP);
}

void Compass::getRoad(Position& from,Node *destiny) {
    road.push_back(destiny→getPosition());
    Node* next_node = destiny→getParent();

    Position current_pos = next_node→getPosition();
    while ((current_pos.getX() ≠ from.getX()) ∨
           (current_pos.getY() ≠ from.getY())) {
        this→addPositions(current_pos);
//        road.push_back(current_pos);
        next_node = next_node→getParent();
        current_pos = next_node→getPosition();
    }
}

Node *Compass::searchForClosestNode() {
    Node* closest = closed_nodes.front();
    for (auto x: closed_nodes){
        if ((x→getHvalue() < closest→getHvalue()) ∨
            ((x→getHvalue() ≡ closest→getGValue()) ∧
             (x→getFValue() < closest→getFValue()))) {
            closest = x;
        }
    }
    return closest;
}

int Compass::getModuleOfSubtraction(int x, int y) {
    if ((x − y) > 0)
        return x − y;
    return y − x;
}

void Compass::checkIfIsDestinyNeighbor(Node *node, int step) {
    if ((node→getHvalue() ≤ HMIN*2) ∧ (node→getHvalue() ≠ 0)){
        // get limits
        int x_min = node→getPosition().getX() − step;
        int x_max = node→getPosition().getX() + step;
        int y_min = node→getPosition().getY() − step;
        int y_max = node→getPosition().getY() + step;

        Node *adj;
        bool adj_new_g;
        for (int x_pos = x_min; x_pos ≤ x_max;x_pos += step) {
            for (int y_pos = y_min; y_pos ≤ y_max; y_pos += step) {
                if (map.doesThisPositionExist(x_pos, y_pos)) {
                    adj = astar_map[x_pos][y_pos];
                    Size size = adj→getSize();
                    this→setHValueOnNode(adj);
                    if (adj→getHvalue() ≡ 0) {
                        // G value differs when the node is diagonal
                        // or next to it
                        if ((map.canIWalkToThisPosition(size, unit_id)) ∧
                            this→isNotMe(node, adj)) {
                            if (this→isThisNodeOnDiagonal(node, adj)) {
                                adj_new_g = this→writeGandSetParent(node, adj,
                                                                     DIAGONALWAL
K,
                                                                     0);
                            } else {
                                adj_new_g = this→writeGandSetParent(node, adj,
                                                                     SIDEWALK,
                                                                     0);
                            }
                            if (adj_new_g)
                                this→addToOpenInOrder(adj);
                        }
                    }
                }
            }
        }
    }
}

bool Compass::canIWalkToThisPosition(Size &size) {
    return map.canIWalkToThisPosition(size, unit_id);
}

double Compass::getTerrainFactorOn(int x, int y) {
    return map.getTerrainFactorOn(x,y);
}

bool Compass::canBulletWalkToThisPosition(Size &size, Occupant &shooter,
                                          Occupant &target) {
    return map.canBulletWalkToThisPosition(size,shooter,target);
}

void Compass::changeUnitSize(Size &new_size) {
    this→unit_size = new_size;
}

void Compass::changeUnitSpeed(int speed) {
    this→unit_speed = speed;
}

Position Compass::getAValidPositionForDestiny(Position &destiny) {
    Node *dest = astar_map[destiny.getX()][destiny.getY()];
    Size size = dest→getSize();
    if (map.canIWalkToThisPosition(size, unit_id)) {
        return destiny;
    } else {
        return getClosestValidPosition(destiny);
    }
}

Position Compass::getClosestValidPosition(Position &pos) {
    bool found = false;
    int i = 1;
    Node* closest_node = astar_map[pos.getX()][pos.getY()];
    while (¬found) {
        int x_min = pos.getX() − i;
```

```
        int x_max = pos.getX() + i;
        int y_min = pos.getY() − i;
        int y_max = pos.getY() + i;

        for (int x_pos = x_min; x_pos ≤ x_max; ++x_pos) {
            if (map.doesThisPositionExist(x_pos, y_max)) {
                Node *tmp = astar_map[x_pos][y_max];
                Size size = tmp→getSize();
                std::string terrain_type = map.getTerrainType(x_pos,y_max);
                // if you fit on the position. When it's a vehicule check
                // if it's different to water.
                if ((map.canIWalkToThisPosition(size, unit_id)) ∧
                    (¬(unit_speed ≠ 2 ∧ terrain_type ≡ "Agua" ∧
                       ¬map.thereIsABridge(size)))) {
                    found = true;
                    closest_node = tmp;
                    break;
                }
            }
        }

        if (¬found) {
            for (int x_pos = x_min; x_pos ≤ x_max; ++x_pos) {
                if (map.doesThisPositionExist(x_pos, y_min)) {
                    Node *tmp = astar_map[x_pos][y_min];
                    Size size = tmp→getSize();
                    std::string terrain_type = map.getTerrainType(x_pos,y_min);
                    // if you fit on the position. When it's a vehicule check
                    // if it's different to water.
                    if ((map.canIWalkToThisPosition(size,unit_id)) ∧
                        (¬(unit_speed ≠ 2 ∧ terrain_type ≡ "Agua" ∧
                           ¬map.thereIsABridge(size)))) {
                        found = true;
                        closest_node = tmp;
                        break;
                    }
                }
            }
        }

        if (¬found) {
            for (int y_pos = y_min; y_pos ≤ y_max; ++y_pos) {
                if (map.doesThisPositionExist(x_max, y_pos)) {
                    Node *tmp = astar_map[x_max][y_pos];
                    Size size = tmp→getSize();
                    std::string terrain_type = map.getTerrainType(x_max,y_pos);
                    // if you fit on the position. When it's a vehicule check
                    // if it's different to water.
                    if ((map.canIWalkToThisPosition(size, unit_id)) ∧
                        (¬(unit_speed ≠ 2 ∧ terrain_type ≡ "Agua" ∧
                           ¬map.thereIsABridge(size)))) {
                        found = true;
                        closest_node = tmp;
                        break;
                    }
                }
            }
        }

        if (¬found) {
            for (int y_pos = y_min; y_pos ≤ y_max; ++y_pos) {
                if (map.doesThisPositionExist(x_min, y_pos)) {
                    Node *tmp = astar_map[x_min][y_pos];
                    Size size = tmp→getSize();
                    std::string terrain_type = map.getTerrainType(x_min,y_pos);
                    // if you fit on the position. When it's a vehicule check
                    // if it’s different to water.
                    if ((map.canIWalkToThisPosition(size, unit_id)) ∧
                        (¬(unit_speed ≠ 2 ∧ terrain_type ≡ "Agua" ∧
                           ¬map.thereIsABridge(size)))) {
                        found = true;
                        closest_node = tmp;
                        break;
                    }
                }
            }
        }

        ++i;
    }
    return closest_node→getSize().getPosition();
}

void Compass::changeUnitId(int id) {
    this→unit_id = id;
}

void Compass::addPositions(Position& next_pos) {
    Position pos = road.back();
    bool increase_x = false, increase_y = false;
    int x_max = 0, x_min = 0, y_max = 0, y_min = 0;
    if (next_pos.getX() > pos.getX()) {
        x_max = next_pos.getX();
        x_min = pos.getX();
        increase_x = true;
    } else if (next_pos.getX() < pos.getX()) {
        x_max = pos.getX();
        x_min = next_pos.getX();
    } else if (next_pos.getX() ≡ pos.getX()){
        x_max = pos.getX();
        x_min = x_max;
        increase_x = true;
    }

    if (next_pos.getY() > pos.getY()) {
        y_max = next_pos.getY();
        y_min = pos.getY();
        increase_y = true;
    } else if (next_pos.getY() < pos.getY()) {
        y_max = pos.getY();
        y_min = next_pos.getY();
    } else if (next_pos.getY() ≡ pos.getY()) {
        increase_y = true;
        y_max = pos.getY();
        y_min = y_max;
    }

    addPositionsInOrder(increase_x,increase_y,x_max,x_min,y_max,y_min);
    Position last = road.back();
    if (last.getX() ≠ next_pos.getX() ∨ last.getY() ≠ next_pos.getY())
        road.push_back(next_pos);
}

void Compass::manageSteps(int &step, Position &start, Position &current_pos,
                          Position &to) {
    int tmp_h = HMIN * (this→getModuleOfSubtraction(current_pos.getX(),
    to.getX()) + this→getModuleOfSubtraction(current_pos.getY(),to.getY()));
    int closer_h =  HMIN * CLOSERAREA * 2;
    //Get smaller H depending on where start and destiny are
    int close_x = 0, close_y = 0, mid_x = 0, mid_y = 0;
    if (start.getX() ≤  to.getX()) {
        close_x = this→getModuleOfSubtraction
```

```
                (start.getX() + CLOSERAREA, to.getX());
        mid_x = this→getModuleOfSubtraction
                (start.getX() + MIDDLEAREA, to.getX());
    } else if (start.getX() >  to.getX()) {
        close_x = this→getModuleOfSubtraction
                (start.getX() − CLOSERAREA, to.getX());
        mid_x = this→getModuleOfSubtraction
                (start.getX() − MIDDLEAREA, to.getX());
    }
    if (start.getY() ≤ to.getY()) {
        mid_y = this→getModuleOfSubtraction
                (start.getY() + MIDDLEAREA,to.getY());
        close_y = this→getModuleOfSubtraction
                (start.getY() + CLOSERAREA,to.getY());
    } else if (start.getY() > to.getY()) {
        close_y = this→getModuleOfSubtraction
                (start.getY() − CLOSERAREA,to.getY());
        mid_y = this→getModuleOfSubtraction
                (start.getY() − MIDDLEAREA,to.getY());
    }
    int start_h =  HMIN * (close_x  + close_y);
    int mid_h = HMIN * (mid_x + mid_y);
    // select step
    if (tmp_h < closer_h ∨ getModule(start_h,closer_h ) < tmp_h) {
        step = 1;
    } else if ((tmp_h > closer_h ∨ getModule(start_h, tmp_h) > closer_h)
               ∧ tmp_h < mid_h) {
        if (unit_size.getWidth() > unit_size.getHeight()) {
            step = (int) (unit_size.getHeight()*2);
        } else {
            step = (int) (unit_size.getWidth()*2);
        }
    } else {
        step = (int) (unit_size.getHeight() * 10);
    }
}

void Compass::setHValueOnNode(Node *node) {
    Position tmp = node→getPosition();
    int h_value = HMIN * (this→getModuleOfSubtraction(tmp.getX(),
    destiny.getX()) + this→getModuleOfSubtraction(tmp.getY(),destiny.getY()));
    node→setHValue(h_value);
}

void Compass::clearCompass() {
    if (¬clear) {
        this→road.clear();
        this→closed_nodes.clear();
        this→open_nodes.clear();
        for (auto x: astar_map) {
            for (auto y: x) {
                y→clean();
            }
        }
        clear = true;
    }
}

void Compass::addPositionsInOrder(bool increase_x, bool increase_y, int x_max,
                                  int x_min, int y_max, int y_min) {
    int i = x_min;
    int j = y_min;
    if (increase_x ∧ increase_y) {
        while (i < x_max ∨ j < y_max) {
            if (i < x_max)
                ++i;
```

```cpp
                if (j < y_max)
                    ++j;
                road.push_back(Position(i, j));
            }
        } else if (increase_x ∧ ¬increase_y) {
            j = y_max;
            while (i < x_max ∨ j > y_min) {
                if (i < x_max)
                    ++i;
                if (j > y_min)
                    −−j;
                road.push_back(Position(i, j));
            }
        } else if (¬increase_x ∧ increase_y) {
            i = x_max;
            while (i > x_min ∨ j < y_max) {
                if (i > x_min)
                    −−i;
                if (j < y_max)
                    ++j;
                road.push_back(Position(i, j));
            }
        } else {
            i = x_max;
            j = y_max;
            while (i > x_min ∨ j > y_min) {
                if (i > x_min)
                    −−i;
                if (j > y_min)
                    −−j;
                road.push_back(Position(i, j));
            }
        }
}

int Compass::getModule(int x, int y) {
    if (x − y > 0) {
        return x − y;
    } else {
        return y − x;
    }
}

Occupant* Compass::checkForEnemiesOnRange(Occupant& unit, Size &range) {
    return map.checkForEnemiesOn(range,unit);
}

bool Compass::checkIfItIsGrabbable(std::string& type) const {
    return map.tellIfItIsGrabbable(type);
}

Compass::~Compass() {
    if (¬astar_map.empty()) {
        int j = 0;
        for (auto x: astar_map) {
            int i = 0;
            for (auto& y: x) {
                delete (y);
                ++i;
            }
            ++j;
        }
    }
}

bool Compass::checkIfItIsABuilding(std::string &type) const {
```

```cpp
    return map.tellIfItIsBuilding(type);
}
```

```cpp
//
// Created by rodian on 15/06/17.
//

#ifndef Z_TPGRUPAL_COMMANDMONITOR_H
#define Z_TPGRUPAL_COMMANDMONITOR_H


#include "command.h"
class Command;
class ControlUnit;

class CommandMonitor {
private:
    std::vector<Command> commands;
    std::mutex& m;
public:
    CommandMonitor(std::mutex& m);

    void addCommand(std::string id, std::string& cmd, ControlUnit* control);

    void copyCommands(std::vector<Command>& commands_copy);
};


#endif //Z_TPGRUPAL_COMMANDMONITOR_H
```

```cpp
//
// Created by rodian on 15/06/17.
//

#include "commandMonitor.h"

CommandMonitor::CommandMonitor(std::mutex &m) : m(m) {}

void CommandMonitor::addCommand(std::string id, std::string& cmd,
                                ControlUnit* control) {
    Lock l(m);
    commands.push_back(Command(id, cmd, control));
}

void CommandMonitor::copyCommands(std::vector<Command> &commands_copy) {
    Lock l(m);
    for (auto cmd: commands) {
        commands_copy.push_back(cmd);
    }

    commands.clear();
}
```

```
//
// Created by rodian on 27/05/17.
//

#ifndef Z_TPGRUPAL_COMMAND_H
#define Z_TPGRUPAL_COMMAND_H

#include "controlUnit.h"
class ControlUnit;

class Command {
protected:
    std::string player_id, cmd;
    ControlUnit* control;
    int unit_id, other_id, x,y;
public:
    Command(std::string id, std::string& cmd, ControlUnit* control);

     void run();

     void operator()();

private:
    std::string getNextData(std::string& line);

    void analizeFactoryCommand(std::string& cmd, int id_factory );
};


#endif //Z_TPGRUPAL_COMMAND_H
```

```
//
// Created by rodian on 27/05/17.
//

#include "command.h"

Command::Command(std::string id,std::string& cmd, ControlUnit* control) :
        player_id(id), cmd(cmd), control(control) {}

void Command::run() {
    std::string action = getNextData(cmd);
    // if command is to move
    if (action ≡ "mv") {
        // get id, then position (x,y)
        std::string id_str = getNextData(cmd);
        int id = std::stoi(id_str);
        std::string x_str = getNextData(cmd);
        int x = std::stoi(x_str);
        std::string y_str = getNextData(cmd);
        int y = std::stoi(y_str);
        // make move
        control→cmdMoveUnit(player_id,id,x,y);
    } else if (action ≡ "atk") {
        std::string id_unit_str = getNextData(cmd);
        int id_unit = std::stoi(id_unit_str);
        std::string target_str = getNextData(cmd);
        int target = std::stoi(target_str);
        control→cmdAttack(player_id,id_unit,target);
    } else if (action ≡ "grab") {
        std::string id_unit_str = getNextData(cmd);
        int id_unit = std::stoi(id_unit_str);
        std::string target_str = getNextData(cmd);
        int target = std::stoi(target_str);
        control→cmdGrab(player_id,id_unit,target);
    }else if (action ≡ "factory") {
        std::string id_unit_str = getNextData(cmd);
        int id_factory = std::stoi(id_unit_str);
        std::string factory_cmd = getNextData(cmd);
        analizeFactoryCommand(factory_cmd, id_factory);
    }
}

void Command::operator()() {
    this→run();
}

std::string Command::getNextData(std::string& line) {
    std::size_t found = line.find('−');
    std::string data = line.substr(0,found);
    line.erase(0,found+1);
    return data;
}

void Command::analizeFactoryCommand(std::string& cmd, int id_factory) {
    std::string action = getNextData(cmd);
    if (action ≡ "create") {
        control→cmdFactoryCreate(player_id,id_factory);
    } else if (action ≡ "prev") {
        control→cmdFactoryPrev(player_id,id_factory);
    } else if (action ≡ "next") {
        control→cmdFactoryNext(player_id,id_factory);
    } else if (action ≡ "current") {
        control→cmdFactoryCurrent(player_id,id_factory);
    }
}
```

```
#ifndef Z_TPGRUPAL_CELL_H
#define Z_TPGRUPAL_CELL_H

#include "Terrain.h"
#include "size.h"
#include <iostream>
#include <string>
#include <cstdbool>
#include <mutex>

// Class Cell to represent a position on the map
class Cell {
    private:
        Terrain terrain;
        Size size;

    public:
        // Cell constructor on position (x,y). Builds the Terrain inside it and
        // always has an Occupant. The empty Cell will be the one who has an
        // Occupant with id = −1.
        Cell(int x, int y, int width, int lenght,
                std::string& kind, int factor);

        // Returns a string with the king of Terrain
        std::string getTerrainType() const;

        // Returns the movement factor of the Terrain
        double getMovementFactor() const;

        Position getPosition() const;

        int getWidthOfCell();

        bool areYouOnThisPosition(int x_pos, int y_pos);

    bool isThereACollision(Size& size);

        ~Cell();
};

#endif
```

```
//
// Created by rodian on 13/05/17.
//

#include "cell.h"

Cell::Cell(int x, int y, int width, int lenght, std::string &kind, int factor):
            size(x, y, width, lenght), terrain(kind,factor) {}

std::string Cell::getTerrainType() const{
    return this→terrain.getKind();
}

double Cell::getMovementFactor() const {
    return this→terrain.getFactor();
}

Position Cell::getPosition() const {
    return this→size.getPosition();
}

bool Cell::areYouOnThisPosition(int x_pos, int y_pos) {
    return this→size.areYouOnThisPoint(x_pos, y_pos);
}

int Cell::getWidthOfCell() {
    return size.getWidth();
}

bool Cell::isThereACollision(Size &size) {
    return size.isThereACollision(size);
}

Cell::~Cell() {}
```

```
//
// Created by rodian on 22/05/17.
//

#ifndef Z_TPGRUPAL_BULLET_H
#define Z_TPGRUPAL_BULLET_H

#include <string>
#include <vector>
#include "size.h"
#include "Occupant.h"

class Bullet {
private:
    std::string type;
    int damage, w_speed;
    Size w_size;
    bool hit, must_disapear;
    Occupant* target;
    std::vector<Position> road;
    int id;

public:
    Bullet(std::string type, int damage, int w_speed, Size& w_size,
           Occupant* target);

    Bullet(std::string type, int damage, int w_speed, Size& w_size);

    // Pursues the Target in straight line on each TIC til hits
    void shotTarget(Occupant* target);

    void calculateRoadToTarget();

    void move();

    bool didHit();

    Position calculateNextPosition(double a, double b, int x);

    Position calculateNextInvertPosition(double a,double b,int y);

    Size getSize() const;

    bool isRoadEmpty();

    std::vector<Position>& getRoad();

    void damageThis(Occupant* occupant);

    void setStartLocation(int x, int y);

    void setCorrectId(int id);

    int getId() const;

    Position getPosition() const;

    void disapear();

    bool doYouHaveToDisapear();

    std::string getType();
};


#endif //Z_TPGRUPAL_BULLET_H
```

```
//
// Created by rodian on 22/05/17.
//

#include "bullet.h"

Bullet::Bullet(std::string type, int dmg, int w_speed, Size& w_size,
               Occupant* target) : type(type), damage(dmg), w_speed(w_speed),
                           w_size(w_size), hit(false), must_disapear(false)
                             , target(target), id(0) {}

Bullet::Bullet(std::string type, int damage, int w_speed, Size& w_size) :
        type(type), damage(damage), w_speed(w_speed),
        w_size(w_size), hit(false), must_disapear(false), id(0) {}


void Bullet::shotTarget(Occupant* target) {
    this→target = target;
    calculateRoadToTarget();
}

bool Bullet::didHit() {
    return hit;
}

void Bullet::calculateRoadToTarget() {
    road.clear();
    Position target_pos = target→getPosition();
    Position bullet_pos = w_size.getPosition();

    if (bullet_pos.getX() − target_pos.getX() ≠ 0) {
        // solve the equation system getting the linear function y = ax + b
        int res_y = (bullet_pos.getY() − target_pos.getY());
        int res_x = (bullet_pos.getX() − target_pos.getX());
        double a =  (double) res_y / res_x;
        double b = (double) (bullet_pos.getY() − (a * bullet_pos.getX()));

        if (bullet_pos.getX() > target_pos.getX()) {
            for (int i = target_pos.getX(); i < bullet_pos.getX(); ++i) {
                road.push_back(calculateNextPosition(a, b, i));
            }
        } else if (bullet_pos.getX() < target_pos.getX()) {
            for (int i = target_pos.getX(); i > bullet_pos.getX(); −−i) {
                road.push_back(calculateNextPosition(a, b, i));
            }
        }
    } else {
        // solve the equation system getting the linear function x = ay + b
        double a = (bullet_pos.getX() − target_pos.getX()) /
                (bullet_pos.getY() − target_pos.getY());
        double b = bullet_pos.getX() − (a * bullet_pos.getY());

        if (bullet_pos.getY() > target_pos.getY()) {
            for (int i = target_pos.getY(); i < bullet_pos.getY(); ++i) {
                road.push_back(calculateNextInvertPosition(a, b, i));
            }
        } else if (bullet_pos.getY() < target_pos.getY()) {
            for (int i = target_pos.getY(); i > bullet_pos.getY(); −−i) {
                road.push_back(calculateNextInvertPosition(a, b, i));
            }
        }
    }
}

Position Bullet::calculateNextPosition(double a, double b, int x) {
    double temp_y = a * x + b;
```

```cpp
    return Position(x, (int)temp_y);
}

Size Bullet::getSize() const {
    return this→w_size;
}

void Bullet::move() {
    if (¬hit) {
        // cause target might be moving, recalculate road and then move
        calculateRoadToTarget();
        int distance = w_speed;
        int steps = 0;
        while (¬road.empty() ∧ steps ≤ distance) {
            Position pos = road.back();
            this→w_size.moveTo(pos.getX(), pos.getY());
            road.pop_back();
            ++steps;
        }
        // If you get the target, inflict damage
        Size targ_size = target→getSize();
        if (this→w_size.isThereACollision(targ_size)) {
            target→reduceLifeBy(damage);
            hit = true;
        }
    }
}

Position Bullet::calculateNextInvertPosition(double a, double b, int y) {
    double temp_x = a * y + b;
    return Position((int) temp_x, y);
}

bool Bullet::isRoadEmpty() {
    return road.empty();
}

std::vector<Position>& Bullet::getRoad() {
    return road;
}

void Bullet::damageThis(Occupant* other_target) {
    other_target→reduceLifeBy(damage);
    hit = true;
}

void Bullet::setStartLocation(int x, int y) {
    this→w_size.moveTo(x,y);
}

void Bullet::setCorrectId(int id) {
    this→id = id;
}

int Bullet::getId() const {
    return id;
}

Position Bullet::getPosition() const {
    return this→w_size.getPosition();
}

bool Bullet::doYouHaveToDisapear() {
    return must_disapear;
}

```

```cpp
void Bullet::disapear() {
    must_disapear = true;
}

std::string Bullet::getType() {
    return this→type;
}

```

```
#ifndef Z_TPGRUPAL_RANDOM_H
#define Z_TPGRUPAL_RANDOM_H

#include <ctime>

/* Random number generator */
class Random {
    unsigned int seed;
public:
    Random();

    /* Returns a random int from 0 to INT_MAX */
    int generate();
};


#endif //Z_TPGRUPAL_RANDOM_H
```

```
#include <stdlib.h>
#include "Random.h"

Random::Random() : seed((unsigned int) time(NULL)) {
}

int Random::generate() {
    return rand_r(&seed);
}
```

```
#ifndef Z_TPGRUPAL_MAPGENERATOR_H
#define Z_TPGRUPAL_MAPGENERATOR_H

#include <fstream>
#include <vector>
#include <string>
#include <pugixml.hpp>
#include "Random.h"

// Config variables, percentages / amounts of features the map will have
#define ROCK_PCT 2
#define BRIDGE_AMT size / 20
#define RIVER_END_PCT 5
#define FORTS_AMT 4



/* Map generator. Randomly generates a readable .xml map file basing off the
 * passed arguments on the constructor. The maps are saved to the 'maps' folder
 * in the root directory. */
class MapGenerator {
    std::vector<std::vector<bool>> liquid_cells;
    std::ofstream output;
    int size;
    float lava_pct;
    float water_pct;
    int water_cells;
    int lava_cells;
    int terr;

    std::string name;
    // Random number generator
    Random r;

public:
    MapGenerator(int size, float lava_pct,
                 float water_pct, int territories);
    ~MapGenerator();

    // Generate the map, saving it to "maps/<name>.xml".
    void generate(const std::string& name);

private:
    /* Inits a map */
    void generate_blank_map(pugi::xml_node root_node);

    /* Generates cell_amt of cells, ordered in a river like structure, in the
     * map given by the root node. The cells are written with a "terrain"
     * attribute as children of the root node, with the value given by the
     * string 'terrain'. */
    std::vector<std::vector<bool>> generate_rivers(pugi::xml_node root_node,
                                                   int cell_amt,
                                                   const std::string &terrain);

    /* Generates river-like paths in a 'size' big square map, represented by
     * a matrix of boolean values. */
    void  generate_path(int amt, std::vector<std::vector<bool>>& path);

    /* Generates rocks */
    void generate_rocks(pugi::xml_node root);

    /* Generates FORTS_AMT forts in the map, placed separate from each other */
    void generate_territories(pugi::xml_node root);

    /* Generates 1 or 2 factories inside the territory delimited by the
     * specified bounds */
```

```
    void generate_factories(pugi::xml_node &territory, int min_x, int min_y, int
 max_x,
                                int max_y, pugi::xml_node &map);
};


#endif //Z_TPGRUPAL_MAPGENERATOR_H
```

```cpp
#include <fstream>
#include <string>
#include <vector>
#include <iostream>
#include "MapGenerator.h"
#include <pugixml.hpp>
#include <random>

#define UNIT 0
#define VEHICLE 1

#define TERRAIN "terrain"

MapGenerator::MapGenerator(int size, float lava_pct,
                           float water_pct, int territories) :
    size(size),
    lava_pct(lava_pct),
    water_pct(water_pct),
    terr(territories)
{
    for (int i = 0; i < size; ++i) {
        std::vector<bool> row;
        for (int j = 0; j < size; ++j) {
            row.push_back(false);
        }
        liquid_cells.push_back(row);
    }
    water_cells = (int) (size * size * water_pct / 100);
    lava_cells = (int) (size * size * lava_pct / 100);

    /* Adjustment to size to split territories evenly */
    int territories_per_row =  (int) floor(sqrt(terr));
    if (size % territories_per_row) {
        this→size = size − territories_per_row;
    }
}


void MapGenerator::generate_blank_map(pugi::xml_node root_node) {
    for (int i = 0; i < size; ++i) {
        pugi::xml_node row = root_node.append_child("Row");
        for (int j = 0; j < size; ++j) {
            pugi::xml_node cell = row.append_child("Cell");
            pugi::xml_attribute attr = cell.append_attribute(TERRAIN);
            attr.set_value("Tierra");
        }
    }
}


std::vector<std::vector<bool>>
MapGenerator::generate_rivers(pugi::xml_node root_node, int cell_amt,
                                      const std::string &terrain) {
    std::vector<std::vector<bool>> map;
    generate_path(cell_amt, map);
    int count_y = 0;
    for (pugi::xml_node& row : root_node.children()) {
        int count_x = 0;
        for (pugi::xml_node& node : row.children()) {
            if (map[count_x][count_y]) {
                node.attribute(TERRAIN).set_value(terrain.c_str());
            }
            count_x++;
        }
        count_y++;
    }
```

```cpp

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; ++j) {
            if (map[i][j]) {
                liquid_cells[i][j] = true;
            }
        }
    }
    return map;
}


void MapGenerator::generate_path(int amt,
                                 std::vector<std::vector<bool>>& path) {
    for (int i = 0; i < size; ++i) {
        std::vector<bool> row;
        for (int j = 0; j < size; ++j) {
            row.push_back(false);
        }
        path.push_back(row);
    }

    int river_x = r.generate() % size;
    int river_y = r.generate() % size;

    while (amt) {
        path[river_x][river_y] = true;

        bool found = false;

        while (¬found) {
            int end = r.generate() % 100;
            if (end < RIVER_END_PCT) { // Start another river somewhere else
                river_x = r.generate() % size;
                river_y = r.generate() % size;
            }
            // Grab an adjacent tile randomly to be the next water tile
            int next = r.generate() % 4;
            int next_x, next_y;
            if (next ≡ 0) {
                next_x = 1;
                next_y = 0;
            } else if (next ≡ 1) {
                next_x = 0;
                next_y = −1;
            } else if (next ≡ 2) {
                next_x = −1;
                next_y = 0;
            } else {
                next_x = 0;
                next_y = 1;
            }
            next_x += river_x;
            next_y += river_y;


            // Check for out of bounds
            if (¬(next_x > 0 ∧ next_y > 0 ∧ next_x < size ∧ next_y < size)) {
                continue;
            }

            if (¬path[next_x][next_y]) {
                found = true;
                amt−−;
                river_x = next_x;
                river_y = next_y;
            }
        }
    }
}


void MapGenerator::generate_rocks(pugi::xml_node root) {
    root.set_name("Structures");
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            if (¬liquid_cells[i][j]) {
                int chance = r.generate() % 100;
                if (chance < ROCK_PCT) {
                    pugi::xml_node rock = root.append_child("Struct");
                    rock.append_attribute("Type").set_value("Rock");
                    rock.append_attribute("x").set_value(i);
                    rock.append_attribute("y").set_value(j);
                }
            }
        }
    }
}


void MapGenerator::generate(const std::string& name) {
    std::string path = "maps/" + name + ".xml";
    pugi::xml_document document;
    pugi::xml_node root = document.append_child("Map");
    pugi::xml_node terrain = root.append_child("Terrain");
    generate_blank_map(terrain);

    generate_territories(root);
    generate_rivers(terrain, water_cells, "Agua");
    generate_rivers(terrain, lava_cells, "Lava");


    pugi::xml_node structs = root.append_child("Structs");
    generate_rocks(structs);
    bool saved = document.save_file(path.c_str());
    if (¬saved) {
        std::cout << "Error saving map to " << path << std::endl;
    }
}


void MapGenerator::generate_territories(pugi::xml_node root) {
    pugi::xml_node forts = root.append_child("Territories");

    /* Choose exactly FORTS_AMT of territories to be designed as central.
     * There’s one fort for each expected player in the map */
    int fort_territories[FORTS_AMT];
    for (int k = 0; k < FORTS_AMT; ++k) {
        bool found = false;
        while (¬found) {
            int position = r.generate() % terr;
            bool repeat = false;
            for (int i = 0; i < k; ++i) {
                if (fort_territories[i] ≡ position) {
                    repeat = true;
                }
            }
            if (repeat) {
                continue;
            }
            fort_territories[k] = position;
            found = true;
```

```
        }
    }

    double size_sqrt =  sqrt(terr);
    int territories_x = (int) floor(size_sqrt);
    int territories_y = (int) ceil(size_sqrt);

    int div_x = size / territories_x;
    int div_y = size / territories_y;
    int count = 0;
    for (int i = 0; i < territories_y; ++i) {
        for (int j = 0; j < territories_x; ++j) {
            /* Randomize positions in the territories */
            int terr_min_x = div_x * j,
                terr_min_y = div_y * i,
                terr_max_x = div_x * (j + 1) − 1,
                terr_max_y = div_y * (i + 1) − 1;

            std::string name = "Flag";
            for (int k = 0; k < FORTS_AMT; ++k) {
                if (fort_territories[k] ≡ count) {
                    name = "Fort";
                }
            }

            bool found = false;
            int flag_x = 0;
            int flag_y = 0;

            while (¬found) {
                flag_x = terr_min_x + r.generate() % (size / terr);
                flag_y = terr_min_y + r.generate() % (size / terr);
                if (¬liquid_cells[flag_x][flag_y]) {
                    found = true;
                }
            }
            pugi::xml_node flag = forts.append_child(name.c_str());
            flag.append_attribute("center_x").set_value(flag_x);
            flag.append_attribute("center_y").set_value(flag_y);
            flag.append_attribute("min_x").set_value(terr_min_x);
            flag.append_attribute("min_y").set_value(terr_min_y);
            flag.append_attribute("max_x").set_value(terr_max_x);
            flag.append_attribute("max_y").set_value(terr_max_y);

            pugi::xml_node map = root.child("Terrain");
            generate_factories(flag, terr_min_x, terr_min_y, terr_max_x,
                               terr_max_y, map);
            count ++;
        }
    }
}

void MapGenerator::generate_factories(pugi::xml_node &territory, int min_x,
                                      int min_y, int max_x,
                                      int max_y, pugi::xml_node &map) {
    int territories = 2;
    for (int i = 0; i < territories; ++i) {
        bool found = false;
        while(¬found) {
            /* Randomize the position, inside the territory */
            int fact_x = r.generate() % (max_x − min_x) + min_x;
            int fact_y = r.generate() % (max_y − min_y) + min_y;

            /* Select type: unit or vehicle */
            int unit_or_vehicle_factory = r.generate() % 2;
            std::string type;
            if (unit_or_vehicle_factory ≡ UNIT) {
                type = "UnitFactory";
            } else if (unit_or_vehicle_factory ≡ VEHICLE) {
                type = "VehicleFactory";
            }

            int count_x = 0;
            for (pugi::xml_node row : map.children()) {
                int count_y = 0;
                for (pugi::xml_node cell : row.children()) {
                    if (fact_x ≡ count_x ∧ fact_y ≡ count_y) {
                        const char* terrain = cell.attribute(TERRAIN).value();
                        if (¬liquid_cells[fact_x][fact_y]) {
                            pugi::xml_node factory =
                                    territory.append_child(type.c_str());
                            factory.append_attribute("x").set_value(fact_x);
                            factory.append_attribute("y").set_value(fact_y);
                            found = true;
                            break;
                        }
                    }
                    count_y++;
                }
                if (found) {
                    break;
                }
                count_x++;
            }

        }
    }
}

MapGenerator::~MapGenerator() {
    if (output.is_open()) {
        output.close();
    }
}
```

```
#include <iostream>
#include "MapGenerator.h"

#define SIZE 1
#define WATER_PCT 2
#define LAVA_PCT 3
#define TERRITORIES 4
#define NAME 5

int main(int argc, char **argv) {
    if (argc < 6) {
        std::cout << "Usage: " << argv[0] << " <size> <water_pct> <lava_pct> "
                "<territories_amt> <name>" << std::endl;
        return 1;
    }

    try {
        int size = std::stoi(argv[SIZE]),
        water = std::stoi(argv[WATER_PCT]),
        lava = std::stoi(argv[LAVA_PCT]),
        territories = std::stoi(argv[TERRITORIES]);
        std::string name = argv[NAME];
        MapGenerator generator(size, lava, water, territories);
        generator.generate(name);
    } catch (const std::invalid_argument& e) {
        std::cout << "Usage: " << argv[0] << " <size> <water_pct> <lava_pct> "
                "<territories_amt> <name>" << std::endl;
        return 1;
    }
}
```

```
#ifndef Z_TPGRUPAL_THREAD_H
#define Z_TPGRUPAL_THREAD_H


#include <thread>

class Thread {
protected:
    std::thread thread;

public:
    void start();

    virtual void run() = 0;

    void join();
};


#endif //Z_TPGRUPAL_THREAD_H
```

```
#include "Thread.h"

void Thread::start() {
    thread = std::thread(&Thread::run, this);
}

void Thread::join() {
    this→thread.join();
}
```

```
#ifndef TP3TALLER_COMMON_SPLIT_H
#define TP3TALLER_COMMON_SPLIT_H

#include <vector>
#include <string>

/* Splits the input string in as many strings as possible, using the char
 * 'delim' as the delimiter between the result strings. Returns the smaller
 * strings in a vector. The function is guaranteed to return with at least
 * one string in the vector.
 */
namespace utils {
    std::vector<std::string> split(const std::string& input, char delim);
}

#endif //TP3TALLER_COMMON_SPLIT_H

```

```
#include <vector>
#include <string>
#include <sstream>

namespace utils {
    std::vector<std::string> split(const std::string& input, char delim) {
        std::vector<std::string> result;
        std::istringstream stream(input);
        for (std::string field; std::getline(stream, field, delim); ) {
            result.push_back(field);
        }
        return result;
    }
};

```

jun 27, 17 14:46 **socket.h** Page 1/1

```cpp
#ifndef TP3TALLER_COMMON_SOCKET_H
#define TP3TALLER_COMMON_SOCKET_H

#include <string>
#include "socketError.h"

#define LISTEN_BACKLOG 10 // Amt. of connections to have in the accept backlog

// Socket class. Wraps functionality of glibc’s socket functions.
class Socket {
    int fd;

public:
    /* Server constructor. Creates a socket, binds and listens to the specified
     * port. */
    explicit Socket(int port);

    /* Client constructor. Creates a socket and attempts to connect to the
     * specified address/port. Raises exception if the connection fails. */
    Socket(const char *addr, int port);

    ~Socket();

    // Returns a new client.
    Socket accept_client();

    // Sends/recieves len bytes of data
    ssize_t send(const char *msg, unsigned int len);
    ssize_t receive(char *dest, size_t len);

    // Wrapper for socket shutdown/close
    void shutdown();
    void close(); // Effectively makes the socket object useless

    bool is_valid();
    // Move constructor
    Socket(Socket∧ other);

private:
    Socket();

    Socket(Socket&) = delete;
    void operator=(Socket&) = delete;
};

#endif //TP3TALLER_COMMON_SOCKET_H
```

jun 27, 17 14:46 **socketError.h** Page 1/1

```cpp
//
// Created by rodian on 22/05/17.
//

#ifndef Z_TPGRUPAL_SOCKETERROR_H
#define Z_TPGRUPAL_SOCKETERROR_H

#include <iostream>
#include <cstring>

////////////////////////
// SocketError Class to warn
// of an error on the socket
////////////////////////

class SocketError : public std::exception {
private:
    char buffer[124];

public:
    explicit SocketError(const char* message, ...) noexcept;

    // Returns the error message
    virtual const char* what() const noexcept;
};

#endif //Z_TPGRUPAL_SOCKETERROR_H
```

```
//
// Created by rodian on 22/05/17.
//

#include "socketError.h"

SocketError::SocketError(const char *message, ...) noexcept {
    strncpy(buffer, message, strlen(message));
}

const char* SocketError::what() const noexcept {
    return buffer;
}
```

```
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <cstring>
#include <utility>
#include <iostream>
#include <string>
#include "socket.h"

Socket::Socket(int port) {
    fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) {
        throw SocketError("Couldn’t create a socket!\n");
    }

    struct sockaddr_in srv;
    memset(&srv, 0, sizeof(srv));
    srv.sin_family = AF_INET;
    srv.sin_addr.s_addr = htonl(INADDR_ANY);
    srv.sin_port = htons((uint16_t) port);

    int yes = 1;
    if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) < 0)
        throw SocketError("setsockopt(SO_REUSEADDR) failed");

    int error = bind(fd, (struct sockaddr *) &srv, sizeof(srv));
    if (error) {
        throw SocketError("Error binding socket on creation! "
                                              "Most likely port already in use");
    }

    listen(fd, LISTEN_BACKLOG);
}

Socket::Socket(const char *addr, int port) {
    fd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in srv;
    srv.sin_family = AF_INET;
    srv.sin_port = htons((uint16_t)port);
    srv.sin_addr.s_addr = inet_addr(addr);

    socklen_t len = (socklen_t)sizeof(struct sockaddr);
    int error = connect(fd, (struct sockaddr *) &srv, len);

    if (error) {
        throw SocketError("Error connecting to server!");
    }
}


Socket Socket::accept_client() {
    struct sockaddr_in client;
    socklen_t clilen = (socklen_t) sizeof(struct sockaddr_in);

    int client_fd = accept(fd, (struct sockaddr *) &client, &clilen);
    if (client_fd < 0 ∨ fd < 0) {
        throw SocketError("Socket disconnected");
    }

    Socket client_socket;
    client_socket.fd = client_fd;
    return client_socket;
}

Socket::Socket() {
```

```cpp
}

Socket::Socket(Socket&& other) {
    fd = other.fd;
    other.fd = -1; // "Deactivates" other
}

ssize_t Socket::send(const char *msg, unsigned int len) {
    size_t total_bytes = 0;
    ssize_t sent = 1;

    // Sends msg until it's complete OR socket_send returns 0 (connection
    // closed)
    while (total_bytes < len && sent) {
        sent = ::send(fd, msg + total_bytes, len - total_bytes,
                      MSG_NOSIGNAL);
        if (sent < 0) {
            return -1;
        }
        total_bytes += sent;
    }

    return total_bytes;
}

ssize_t Socket::receive(char *dest, size_t len) {
    ssize_t received = 1;
    size_t total_bytes = 0;

    // Writes to dest until it's complete OR socket_recv returns 0 (connection
    // closed)
    while (total_bytes < len && received) {
        received = recv(fd, dest + total_bytes, len - total_bytes,
                        MSG_NOSIGNAL);
        if (received < 0) {
            return -1;
        }
        total_bytes += received;
    }
    return total_bytes;
}


Socket::~Socket() {
    if (fd > 0) {
        close();
    }
}

void Socket::shutdown() {
    ::shutdown(fd, SHUT_RDWR);
    fd = -1;
}

void Socket::close() {
    ::close(fd);
    fd = -1;
}

bool Socket::is_valid() {
    return fd > 0;
}
```

```cpp
//
// Created by rodian on 22/05/17.
//

#ifndef Z_TPGRUPAL_MESSENGER_H
#define Z_TPGRUPAL_MESSENGER_H

#include "socket.h"
#include <iostream>
#include <string>
////////////////////////
// Messenger Class meant to use sockets
// to send messages between Client and
// Server using a specific protocol.
// Send lenght of message first, then the message.
////////////////////////
class Messenger{
private:
    Socket socket;

public:
    // Recieves a unique socket to send
    // and recieves messages from
    explicit Messenger(Socket& socket);

    // Recieves a Message from the remote
    // connected socket.
    // Returns the message on a string
    std::string recieveMessage();

    // Sends a message to the remote socket
    // Recieves the message on a string
    void sendMessage(const std::string &message);

    // Shuts down the socket for read and write
    void shutdown();

    // If the sockets are still connected returns true
    // otherwise false.
    bool isConnected();

    ~Messenger();

    Messenger(Messenger& other);
};


#endif //Z_TPGRUPAL_MESSENGER_H
```

```cpp
#include <netinet/in.h>
#include <string>
#include <cstdint>
#include <stdint-gcc.h>
#include "socket.h"
#include "messenger.h"

Messenger::Messenger(Socket& socket) : socket(std::move(socket)) {}


std::string Messenger::recieveMessage() {
    // Receive length first, then the message
    uint32_t len = 0;
    socket.receive((char*) &len, sizeof(len));
    len = ntohl(len);
    char* buf = new char[len];
    ssize_t sent = socket.receive(buf, len);
    if (sent ≤ 0) {
        socket.close();
        throw(SocketError("Socket closed"));
    }
    std::string result(buf);

    delete[] buf;
    return result;
}

void Messenger::sendMessage(const std::string &message) {
    if (isConnected()) {
        uint32_t len = (uint32_t) message.size() + 1;
        // Send length first, then the message
        uint32_t network_len = htonl(len);
        socket.send((char *) &network_len, sizeof(network_len));
        socket.send(message.c_str(), len);
    }
}

void Messenger::shutdown() {
    socket.shutdown();
}

bool Messenger::isConnected() {
    return socket.is_valid();
}

Messenger::~Messenger() {
}

Messenger::Messenger(Messenger &other) : socket(std::move(other.socket)){
}
```

```cpp
#ifndef Z_TPGRUPAL_LOCK_H
#define Z_TPGRUPAL_LOCK_H


#include <mutex>

class Lock {
private:
    std::mutex &m;

public:
    explicit Lock(std::mutex  &m);

    ~Lock();
};


#endif //Z_TPGRUPAL_LOCK_H
```

```cpp
#include "Lock.h"

Lock::Lock(std::mutex &m) : m(m){
    m.lock();
}

Lock::~Lock() {
    m.unlock();
}
```

```cpp
#ifndef Z_TPGRUPAL_RESULTWINDOW_H
#define Z_TPGRUPAL_RESULTWINDOW_H


#include <gtkmm/window.h>
#include <gtkmm/builder.h>
#include <gtkmm/button.h>
#include <gtkmm/label.h>

class ResultWindow : public Gtk::Window {
    Gtk::Button* menu;
    Gtk::Button* close;
    Gtk::Label* winner;
    Gtk::Label* loser;

    bool back_to_menu;
public:
    ResultWindow(BaseObjectType *cobject,
                 const Glib::RefPtr<Gtk::Builder> &builder);

    void display_lose_screen();
    void display_win_screen();

    void menu_click();

    void close_click();

    void on_show();

    bool go_back_to_menu();
};


#endif //Z_TPGRUPAL_RESULTWINDOW_H
```

```
#include "ResultWindow.h"

ResultWindow::ResultWindow(BaseObjectType *cobject,
                           const Glib::RefPtr<Gtk::Builder> &builder) :
        Gtk::Window(cobject)
{

    builder->get_widget("BackToMenuButton", menu);
    builder->get_widget("CloseGameButton", close);
    builder->get_widget("WinnerLabel", winner);
    builder->get_widget("LoserLabel", loser);

    menu->signal_clicked().connect(sigc::mem_fun(*this,
                                             &ResultWindow::menu_click));
    close->signal_clicked().connect(sigc::mem_fun(*this,
                                              &ResultWindow::close_click));
}

void ResultWindow::display_lose_screen() {
    winner->hide();
}

void ResultWindow::display_win_screen() {
    loser->hide();
}

void ResultWindow::menu_click() {
    back_to_menu = true;
    this->hide();
}

void ResultWindow::close_click() {
    back_to_menu = false;
    this->hide();
}

bool ResultWindow::go_back_to_menu() {
    return back_to_menu;
}

void ResultWindow::on_show() {
    back_to_menu = false;
    Gtk::Widget::on_show();
}
```

```
#ifndef Z_TPGRUPAL_MENUWINDOW_H
#define Z_TPGRUPAL_MENUWINDOW_H


#include <gtkmm/window.h>
#include <gtkmm/builder.h>
#include <gtkmm/listbox.h>
#include <gtkmm/button.h>
#include <gtkmm/listviewtext.h>
#include <gtkmm/textview.h>
#include "../ServerMessenger.h"

class MenuWindow : public Gtk::Window {
    Gtk::Label* available_lobbies;
    Gtk::Button* join_button;
    Gtk::Button* create_button;
    Gtk::Entry* lobby_entry;
    ServerMessenger* messenger;

    bool joined_successfully;
public:
    MenuWindow(BaseObjectType *cobject,
               const Glib::RefPtr<Gtk::Builder> &builder);

    void join_click();

    void load_messenger(ServerMessenger* messenger);

    void create_click();
    void update_lobbies(const std::vector<std::string>& lobbies);

    void on_show();

    void join_lobby();
    bool joined_lobby();
};


#endif //Z_TPGRUPAL_MENUWINDOW_H
```

```cpp
#include <gtkmm/label.h>
#include "MenuWindow.h"

MenuWindow::MenuWindow(BaseObjectType *cobject,
                       const Glib::RefPtr<Gtk::Builder> &builder) :
    Gtk::Window(cobject)
{

    builder->get_widget("AvailableLobbies", available_lobbies);
    builder->get_widget("CreateLobbyButton", create_button);
    builder->get_widget("JoinLobbyButton", join_button);
    builder->get_widget("LobbyEntry", lobby_entry);
    join_button->signal_clicked().connect(
            sigc::mem_fun(*this,
                          &MenuWindow::join_click));

    create_button->signal_clicked().connect(
            sigc::mem_fun(*this,
                          &MenuWindow::create_click));
}

void MenuWindow::join_click() {
    std::string lobby = lobby_entry->get_text();
    try {
        std::stoi(lobby);
    } catch(std::invalid_argument& e) {
        std::cerr << "Invalid lobby ID. Insert only numbers!" << std::endl;
        return;
    }
    messenger->send("joinlobby-" + lobby);
}

void MenuWindow::load_messenger(ServerMessenger *messenger) {
    this->messenger = messenger;
}

void MenuWindow::create_click() {
    messenger->send("createlobby");
}

void MenuWindow::update_lobbies(const std::vector<std::string> &lobbies) {
    std::stringstream text;
    for (const std::string& lobby : lobbies) {
        text << lobby << std::endl;
    }
    available_lobbies->set_text(text.str());
}

bool MenuWindow::joined_lobby() {
    return joined_successfully;
}

void MenuWindow::on_show() {
    joined_successfully = false;
    Gtk::Widget::on_show();
}

void MenuWindow::join_lobby() {
    joined_successfully = true;
    hide();
}
```

```cpp
#ifndef Z_TPGRUPAL_LOBBYWINDOW_H
#define Z_TPGRUPAL_LOBBYWINDOW_H


#include <gtkmm/window.h>
#include <gtkmm/button.h>
#include <gtkmm/label.h>
#include <gtkmm/builder.h>
#include <gtkmm/togglebutton.h>
#include <gtkmm/entry.h>
#include <string>
#include <vector>

#define PLAYERS_AMT 4
#include "../ServerMessenger.h"

class LobbyWindow : public Gtk::Window {
    Gtk::Button *start;
    Gtk::Button *ready;
    Gtk::Label *players[PLAYERS_AMT];
    ServerMessenger *m;
    Gtk::Label* maps_label;
    Gtk::Entry* maps_entry;
    std::string default_label;

    bool started = false;

public:
    LobbyWindow(BaseObjectType *cobject,
                const Glib::RefPtr<Gtk::Builder> &builder);

    void set_messenger(ServerMessenger& m);
    void update_player_name(int at, const std::string& name);

    std::vector<std::string> get_player_names();
    void start_game();
    bool game_started();
    void on_show();
    void update_maps(const std::string& maps);

private:
    void click_start();
    void click_ready();
};


#endif //Z_TPGRUPAL_LOBBYWINDOW_H
```

```cpp
#include <iostream>
#include "LobbyWindow.h"
#include "../ServerMessenger.h"

LobbyWindow::LobbyWindow(BaseObjectType *cobject,
                         const Glib::RefPtr<Gtk::Builder> &builder) :
    Gtk::Window(cobject)
{

    builder→get_widget("PlayerStatus1", players[0]);
    builder→get_widget("PlayerStatus2", players[1]);
    builder→get_widget("PlayerStatus3", players[2]);
    builder→get_widget("PlayerStatus4", players[3]);
    builder→get_widget("StartGame", start);
    builder→get_widget("ReadyButton", ready);
    builder→get_widget("MapsLabel", maps_label);
    builder→get_widget("MapsEntry", maps_entry);
    default_label = players[0]→get_text();
    start→signal_clicked().connect(sigc::mem_fun(*this,
                                              &LobbyWindow::click_start));

    ready→signal_clicked().connect(sigc::mem_fun(*this,
                                              &LobbyWindow::click_ready));
}

void LobbyWindow::click_start() {
    std::string map = maps_entry→get_text();
    m→send("startgame−" + map);
}

void LobbyWindow::set_messenger(ServerMessenger &m) {
    this→m = &m;
}

void LobbyWindow::update_player_name(int at, const std::string &name) {
    if (at < 4) {
        players[at]→set_text(name);
    }
}

std::vector<std::string> LobbyWindow::get_player_names() {
    std::vector<std::string> names;
    for (Gtk::Label* player : players) {
        std::string name = player→get_text();
        if (name ≡ default_label) {
            name = "";
        }
        names.push_back(name);
    }
    return names;
}


void LobbyWindow::click_ready() {
    if (ready→get_label() ≡ "Ready") { // already pressed
        m→send("ready");
        ready→set_label("Unready");
    } else if(ready→get_label() ≡ "Unready") {
        m→send("unready");
        ready→set_label("Ready");
    }
}

void LobbyWindow::start_game() {
    started = true;
    this→hide();
```

```cpp
}

bool LobbyWindow::game_started() {
    return started;
}

void LobbyWindow::update_maps(const std::string &maps) {
    maps_label→set_text(maps);
}

void LobbyWindow::on_show() {
    started = false;
    Gtk::Widget::on_show();
}

```

```
#ifndef Z_TPGRUPAL_INITIALWINDOW_H
#define Z_TPGRUPAL_INITIALWINDOW_H


#include <gtkmm/window.h>
#include <gtkmm/builder.h>
#include <gtkmm/entry.h>
#include <gtkmm/button.h>
#include <socket.h>
#include "../ServerMessenger.h"

class InitialWindow : public Gtk::Window {
    Gtk::Entry *address_entry;
    Gtk::Entry *port_entry;
    Gtk::Entry *name_entry;
    Gtk::Button *connect;
    std::shared_ptr<ServerMessenger> messenger;
    std::string name;

public:
    InitialWindow(BaseObjectType *cobject,
                  const Glib::RefPtr<Gtk::Builder> &builder);

    std::shared_ptr<ServerMessenger> get_socket();

    const std::string &get_username();

private:
    void on_click();

    void send_name();
};


#endif //Z_TPGRUPAL_INITIALWINDOW_H
```

```
#include <iostream>
#include "InitialWindow.h"

#define ERROR_MSG "error"
#define OK_MSG "ok"

InitialWindow::InitialWindow(BaseObjectType *cobject,
                             const Glib::RefPtr<Gtk::Builder> &builder) :
        Gtk::Window(cobject) {
    builder→get_widget("AddressEntry", address_entry);
    builder→get_widget("PortEntry", port_entry);
    builder→get_widget("NameEntry", name_entry);
    builder→get_widget("ConnectButton", connect);
    connect→signal_clicked().connect(sigc::mem_fun(*this,
                                                   &InitialWindow::on_click));
}

void InitialWindow::on_click() {
    std::string addr_str = address_entry→get_text();
    std::string port_str = port_entry→get_text();
    name = name_entry→get_text();
    if (messenger.get()) { // Connection already established
        send_name();
        return;
    }

    try {
        int port = 0;
        port = std::stoi(port_str);
        Socket s(addr_str.c_str(), port);
        messenger = std::shared_ptr<ServerMessenger>(new ServerMessenger(s));
        send_name();
    } catch (SocketError &e) {
        std::cerr << "Could not connect to specified addr/port" << std::endl;
        return;
    }
}

void InitialWindow::send_name() {
    messenger.get()→send("changename−" + name);
    std::string response = messenger.get()→receive();
    if (response ≡ ERROR_MSG) {
        std::cerr << "A player with this name already exists" << std::endl;
        return;
    }
    hide();
}

std::shared_ptr<ServerMessenger> InitialWindow::get_socket() {
    return messenger;
}

const std::string &InitialWindow::get_username() {
    return name;
}


```

```
#ifndef Z_TPGRUPAL_GAMEWINDOW_H
#define Z_TPGRUPAL_GAMEWINDOW_H

#include <gtkmm/window.h>
#include <gtkmm/grid.h>
#include <gtkmm/button.h>
#include <gtkmm/builder.h>
#include <gtkmm/box.h>
#include <gtkmm/label.h>
#include <gtkmm/applicationwindow.h>
#include "../GameArea.h"
#include "../BuildingsMonitor.h"
#include "../MapMonitor.h"
#include "../ServerMessenger.h"
#include "../panels/UnitPanel.h"
#include "../panels/BuildingPanel.h"

class GameWindow : public Gtk::ApplicationWindow {
    GameArea *gameArea;
    Gtk::Box *panel;
    UnitPanel *unit_panel;
    BuildingPanel *building_panel;
    Gtk::Box *group_panel;
    Gtk::Label *panelLabel;

    UnitsMonitor *unitsMonitor;
    BuildingsMonitor *buildingsMonitor;
    MapMonitor *mapMonitor;
    ServerMessenger *messenger;

    Unit selected_unit;
    Building selected_building;

    bool unit_selection;
    bool building_selection;

    std::string me;
public:
    GameWindow(BaseObjectType *cobject,
               const Glib::RefPtr<Gtk::Builder> &builder);

    virtual ~GameWindow();

    /**
     * Saves resources’ monitors and passes them to the gameArea.
     */
    void
    setResources(UnitsMonitor *unitsMonitor, BuildingsMonitor *buildingsMonitor,
                 MapMonitor *mapMonitor, ServerMessenger *messenger,
                 const std::string &owner);

    void setMapData();
    void factory_change_unit(std::string &path);

    void update_factory_panel(const std::string& type, int fire_rate, int hp);
    void update_factory_timer(int minutes, int seconds);

    void update_name(const std::string& name);

protected:
    bool onTimeout();

private:
    void update_side_panels();
    // Functions to change the window’s side panel
    bool change_view_to_unit();
```

```

    bool change_view_to_building();

    bool change_view_to_unit_group();

    bool on_button_release_event(GdkEventButton *event);

    void factory_next();

    void factory_create_unit();

    void process_attack();

    bool on_key_press_event(GdkEventKey *event) override;

    void remove_side_panel();

    void factory_prev();

    void process_movement() const;
};

#endif //Z_TPGRUPAL_GAMEWINDOW_H
```

```
#include <iostream>
#include <giomm.h>
#include <gdkmm.h>
#include "GameWindow.h"

#define SCREENWIDTH 1200
#define SCREENHEIGHT 800
#define FRAMERATE 10        //fps

GameWindow::GameWindow(BaseObjectType *cobject,
                       const Glib::RefPtr<Gtk::Builder> &builder) :
        Gtk::ApplicationWindow(cobject)
{
    builder→get_widget_derived("GameArea", gameArea);

    gameArea→set_size_request(SCREENWIDTH * 6 / 7, SCREENHEIGHT);
    builder→get_widget("SidePanel", panel);
    builder→get_widget_derived("BuildingView", building_panel);
    builder→get_widget_derived("UnitView", unit_panel);
    builder→get_widget("GroupView", group_panel);
    builder→get_widget("PanelDisplayLabel", panelLabel);


    building_panel→next_button()→
            signal_clicked().connect(
            sigc::mem_fun(*this, &GameWindow::factory_next));

    building_panel→create_button()→
            signal_clicked().connect(
            sigc::mem_fun(*this, &GameWindow::factory_create_unit));


    building_panel→prev_button()→
            signal_clicked().connect(
            sigc::mem_fun(*this, &GameWindow::factory_prev));
    // Logic for redrawing the map every frame
    sigc::slot<bool> mySlot = sigc::mem_fun(*this, &GameWindow::onTimeout);
    Glib::signal_timeout().connect(mySlot, 1000 / FRAMERATE);

    show_all_children();
    add_events(Gdk::EventMask::KEY_PRESS_MASK);
}

GameWindow::~GameWindow() {
}

bool GameWindow::onTimeout() {
    // force our program to redraw the entire thing
    auto win = get_window();
    if (win) {
        Gdk::Rectangle r(0, 0, get_allocation().get_width(),
                         get_allocation().get_height());
        win→invalidate_rect(r, false);
    }
    update_side_panels();
    return true;
}

void GameWindow::setResources(UnitsMonitor *unitsMonitor,
                              BuildingsMonitor *buildingsMonitor,
                              MapMonitor *mapMonitor,
                              ServerMessenger *messenger,
                              const std::string &owner) {
    this→unitsMonitor = unitsMonitor;
    this→buildingsMonitor = buildingsMonitor;
    this→mapMonitor = mapMonitor;
```

```
    this→messenger = messenger;
    gameArea→setResources(unitsMonitor, buildingsMonitor, mapMonitor, owner);
}


bool GameWindow::change_view_to_unit() {
    for (auto child : panel→get_children()) {
        child→hide();
    }

    unit_panel→show();
    panelLabel→set_text(unit_panel→get_label());
    unit_panel→update_portrait(selected_unit.getType(),
                               selected_unit.getTeam());
    return true;
}

bool GameWindow::change_view_to_building() {
    for (auto child : panel→get_children()) {
        child→hide();
    }
    building_panel→show();
    panelLabel→set_text(building_panel→get_label());

    return true;
}

bool GameWindow::change_view_to_unit_group() {
    for (auto child : panel→get_children()) {
        child→hide();
    }

    group_panel→show();

    return true;
}


bool GameWindow::on_button_release_event(GdkEventButton *event) {
    if (event→button ≡ GDK_BUTTON_SECONDARY) {
        if (¬(selected_unit.get_owner() ≡ me)) {
            return true;
        }

        if (gameArea→unit_selected() ∨ gameArea→buildings_selected()) {
            // We already are selecting an unit, process attack
            process_attack();
        } else {  // Click on empty place, movement
            process_movement();
        }
    } else if (event→button ≡ GDK_BUTTON_PRIMARY) {
        if (gameArea→buildings_selected()) { // New building selected
            selected_building = buildingsMonitor→get_selected().at(0);
            messenger→send(
                    "factory−" + std::to_string(selected_building.get_ID())
                    + "−current");

            change_view_to_building();

            // Change selection status
            unit_selection = false;
            building_selection = true;
        } else if (gameArea→unit_selected()) { // New unit selected
            selected_unit = unitsMonitor→getSelectedUnits().at(0);
            change_view_to_unit();

```

```cpp
            // Change selection status
            building_selection = false;
            unit_selection = true;
        }
    }
    return true;
}


void GameWindow::process_movement() const {
    int id = selected_unit.get_ID();
    std::pair<int, int> coords = gameArea→get_coords();
    int x = coords.first;
    int y = coords.second;

    std::stringstream s;
    int flag = mapMonitor→get_flag_at(x, y);
    if (flag > 0) { // Issue a flag grabbing cmd to move towards the position
        s << "grab−" << id << "−" << flag;
    } else {
        s << "mv−" << id << "−" << x << "−" << y;
    };
    messenger→send(s.str());
}

void GameWindow::factory_next() {
    int id = selected_building.get_ID();
    messenger→send("factory−"+std::to_string(id)+"−next");
}


void GameWindow::factory_change_unit(std::string &path) {
    building_panel→change_unit(path);
}

void GameWindow::factory_create_unit() {
    int id = selected_building.get_ID();
    messenger→send("factory−"+std::to_string(id)+"−create");
}

void GameWindow::setMapData() {
    gameArea→setMapData();
}

void GameWindow::process_attack() {
    std::vector<Unit> units = unitsMonitor→getSelectedUnits();
    std::string target;
    if (units.size()) { // other unit selected
        Unit other = units.at(0);
        if (selected_unit.getTeam() ≡ other.getTeam()) {
            return;
        }
        target = std::to_string(other.get_ID());
    }
    std::vector<Building> buildings = buildingsMonitor→get_selected();
    if (buildings.size()) {
        Building other = buildings.at(0);
        if (selected_unit.getTeam() ≡ other.getTeam()) {
            return;
        }
        target = std::to_string(other.get_ID());
    }

    std::string attack = "atk−" + std::to_string(selected_unit.get_ID()) +
                         "−" + target;
    messenger→send(attack);
```

```cpp
}

void GameWindow::update_name(const std::string &name) {
    me = name;
}

bool GameWindow::on_key_press_event(GdkEventKey *event) {
    // Clear selection
    if (event→keyval ≡ GDK_KEY_Escape) {
        remove_side_panel();
    }
    return Gtk::Window::on_key_press_event(event);
}

void GameWindow::remove_side_panel() {
    selected_building = Building();
    selected_unit = Unit();
    for (auto child : panel→get_children()) {
        child→hide();
    }
    panelLabel→set_text("Z");
}

void GameWindow::update_side_panels() {
    if (¬unit_selection ∧ ¬building_selection) {
        remove_side_panel();
    } else if (unit_selection) {
        int unit_id = selected_unit.get_ID();
        // UpdateUnit the unit reference
        selected_unit = unitsMonitor→get_unit(unit_id);
        unit_panel→set_name(selected_unit.get_unit_name());
        unit_panel→set_owner(selected_unit.get_owner());
        unit_panel→set_max_hp(selected_unit.get_max_hp());
        unit_panel→set_hp(selected_unit.get_hp());
    } else if (building_selection){
        int building_id = selected_building.get_ID();

        selected_building = buildingsMonitor→get_building(building_id);
        building_panel→set_max_hp(selected_building.get_max_hp());
        building_panel→set_hp(selected_building.get_hp());
        building_panel→set_owner(selected_building.get_owner());
        building_panel→set_time_left(selected_building.get_time_left());
    }
}

void GameWindow::factory_prev() {
    int id = selected_building.get_ID();
    messenger→send("factory−"+std::to_string(id)+"−prev");
}

void GameWindow::update_factory_panel(const std::string &type, int fire_rate,
                                      int hp) {
    building_panel→set_unit_hp(hp);
    building_panel→set_unit_fire_rate(fire_rate);
    building_panel→set_unit_type(type, selected_building.getTeam());
}

void GameWindow::update_factory_timer(int minutes, int seconds) {
    building_panel→set_time_left(std::pair<int, int>(minutes, seconds));
}
```

```
#ifndef Z_TPGRUPAL_UNITSMONITOR_H
#define Z_TPGRUPAL_UNITSMONITOR_H


#include <mutex>
#include "Unit.h"

class UnitsMonitor {
    std::vector<Unit> units;
    std::mutex m;

public:

    void addUnit(Unit &unit);

    void removeUnit(int id);

    void update_position(int unit_id, int x, int y);

    std::vector<Unit>
    getUnitsToDraw(unsigned int minX, unsigned int maxX, unsigned int minY,
                   unsigned int maxY);

    void
    markAsSelectedInRange(bool &unitsSelected, gdouble xStartCoordinate,
                          gdouble yStartCoordinate, gdouble xFinishCoordinate,
                          gdouble yFinishCoordinate);

    void wipeSelected();

    std::vector<Unit> getSelectedUnits();

    void update_position(int id, ActionsEnum state, int x, int y);
    void update_health(int id, unsigned int hp);

    Unit get_unit(int id);

    void clear();
};


#endif //Z_TPGRUPAL_UNITSMONITOR_H
```

```
#include <Lock.h>
#include <iostream>
#include "UnitsMonitor.h"
#include <vector>


void UnitsMonitor::addUnit(Unit &unit) {
    Lock l(m);
    units.push_back(unit);
}

void UnitsMonitor::removeUnit(int id) {
    Lock l(m);
    for (auto unit = units.begin(); unit ≠ units.end(); ++unit) {
        if (unit→get_ID() ≡ id) {
            units.erase(unit);
        }
    }
}

void UnitsMonitor::update_position(int unit_id, int x, int y) {
    Lock l(m);
    for (auto unit = units.begin(); unit ≠ units.end(); ++unit) {
        if (unit→get_ID() ≡ unit_id) {
            unit→update_position(x, y);
        }
    }
}

void UnitsMonitor::update_position(int id, ActionsEnum state, int x, int y) {
    Lock l(m);
    for (auto unit = units.begin(); unit ≠ units.end(); ++unit) {
        if (unit→get_ID() ≡ id) {
            unit→update_position(x, y);
            unit→update_state(state);
        }
    }
}

std::vector<Unit>
UnitsMonitor::getUnitsToDraw(unsigned int minX, unsigned int maxX,
                             unsigned int minY, unsigned int maxY) {
    Lock l(m);
    std::vector<Unit> returnVector;

    for (Unit &unit : units) {
        if (unit.getXCoordinate() ≥ minX and
            unit.getXCoordinate() ≤ maxX and
            unit.getYCoordinate() ≥ minY and
            unit.getYCoordinate() ≤ maxY) {
            returnVector.emplace_back(unit);
        }
    }
    return returnVector;
}

void
UnitsMonitor::markAsSelectedInRange(bool &unitsSelected,
                                    gdouble xStartCoordinate,
                                    gdouble yStartCoordinate,
                                    gdouble xFinishCoordinate,
                                    gdouble yFinishCoordinate) {
    Lock l(m);
    for (Unit &unit : units) {
        unit.markAsSelectedInRange(unitsSelected, xStartCoordinate,
                                   yStartCoordinate, xFinishCoordinate,
```

```cpp
                                   yFinishCoordinate);
        /* if unit selected, break because we only want to select one unit */
        if (unitsSelected)
            break;
    }
}

std::vector<Unit> UnitsMonitor::getSelectedUnits() {
    std::vector<Unit> units;
    for (Unit &unit : this→units) {
        if (unit.is_selected()) {
            units.push_back(unit);
        }
    }
    return units;
}

void UnitsMonitor::wipeSelected() {
    for (Unit &unit : units) {
        unit.unselect();
    }
}

void UnitsMonitor::update_health(int id, unsigned int hp) {
    Lock l(m);
    for (auto unit = units.begin(); unit ≠ units.end(); ++unit) {
        if (unit→get_ID() ≡ id) {
            if (¬hp) {
                units.erase(unit);
                break;
            } else {
                unit→update_hp(hp);
            }
        }
    }
}

Unit UnitsMonitor::get_unit(int id) {
    for (Unit& unit: units) {
        if (unit.get_ID() ≡ id){
            return unit;
        }
    }
    return Unit();
}

void UnitsMonitor::clear() {
    Lock l(m);
    units.clear();
}
```

```cpp
#ifndef Z_TPGRUPAL_UNIT_H
#define Z_TPGRUPAL_UNIT_H


#include <vector>
#include <string>
#include <map>
#include "gtkmm/drawingarea.h"
#include "Armament.h"
#include "enums/TeamEnum.h"
#include "enums/ActionsEnum.h"
#include "enums/UnitsEnum.h"
#include "enums/RotationsEnum.h"
#include <utility>
#include <mutex>

class Unit {
private:
    int id;

    /* unitType can be: robot, vehicle, tank */
    UnitsEnum unitType;

    std::string unit_name;

    ActionsEnum actionType;

    RotationsEnum rotation;

    Armament armament;

    /* freq. with which the unit shoots */
    unsigned short frequency;

    unsigned int lifeLeft;

    unsigned int totalLife;

    unsigned short velocity;

    std::pair<unsigned int, unsigned int> position;
    std::pair<unsigned int, unsigned int> prev_position;

    /* bool selected: indicates weather the unit has been selected
     * with the mouse or not */
    bool selected;

    /* bool that indicates whether the unit is shooting or not */
    bool shooting;

    /* indicates to which getTeam the unit belongs */
    TeamEnum team;

    /* counters to know which img of the ones that conform an animation
     * should be drawn*/
    unsigned short shootingDrawingCounter;
    unsigned short standingDrawingCounter;
    unsigned short walkingDrawingCounter;

    std::string owner;
public:
    Unit(int id, std::pair<unsigned int, unsigned int> position,
             UnitsEnum unitType, TeamEnum team, unsigned int hp);

    Unit();
    int get_ID() const;

    void update_position(int x, int y);

    void update_state(ActionsEnum state);

    void markAsSelectedInRange(bool &unitsSelected, gdouble xStartCoordinate,
                               gdouble yStartCoordinate,
                               gdouble xFinishCoordinate,
                               gdouble yFinishCoordinate);

    bool isShooting();

    TeamEnum getTeam();

    RotationsEnum getRotation();

    UnitsEnum getType();

    ActionsEnum getAction();

    unsigned int getXCoordinate();

    unsigned int getYCoordinate();

    bool is_selected();

    void unselect();

    void update_rotation();

    void update_owner(const std::string& owner);

    void update_unit_name(const std::string& name);

    void update_hp(unsigned int hp);
    std::string get_owner();
    int get_max_hp();
    int get_hp();
    std::string get_unit_name();
};


#endif //Z_TPGRUPAL_UNIT_H
```

```
#include "Unit.h"
#include <iostream>

#define IMG_SIZE_IN_PX 4
#define TEAM_NEUTRAL "None"
Unit::Unit(int id, std::pair<unsigned int, unsigned int> position,
           UnitsEnum unitType, TeamEnum team, unsigned int hp)
        : id(id), position(position), unitType(unitType),
          rotation(RotationsEnum::r090), team(team),
          selected(false), actionType(ActionsEnum::STAND),
          owner(TEAM_NEUTRAL), totalLife(hp), lifeLeft(hp)
{
    /* units initial rotation is facing 'to the player'; action: standing. */
    //todo check what else should be initialized
}

int Unit::get_ID() const {
    return id;
}

void Unit::update_position(int x, int y) {
    std::pair<int, int> new_pos(x, y);

    prev_position = position;
    position = new_pos;
    update_rotation();
}

void Unit::update_rotation() {
    auto prev_x = prev_position.first;
    auto prev_y = prev_position.second;
    auto x = position.first;
    auto y = position.second;

    if (x ≡ prev_x and y < prev_y) {
        rotation = RotationsEnum::r090;
    } else if (x > prev_x and y < prev_y) {
        rotation = RotationsEnum::r045;
    } else if (x > prev_x and y ≡ prev_y) {
        rotation = RotationsEnum::r000;
    } else if (x > prev_x and y > prev_y) {
        rotation = RotationsEnum::r315;
    } else if (x ≡ prev_x and y > prev_y) {
        rotation = RotationsEnum::r270;
    } else if (x < prev_x and y > prev_y) {
        rotation = RotationsEnum::r225;
    } else if (x < prev_x, y ≡ prev_y) {
        rotation = RotationsEnum::r180;
    } else if (x < prev_x and y < prev_y) {
        rotation = RotationsEnum::r135;
    }
}

void Unit::update_state(ActionsEnum state) {
    this→actionType = state;
}

void
Unit::markAsSelectedInRange(bool &unitsSelected, gdouble xStartCoordinate,
                            gdouble yStartCoordinate,
                            gdouble xFinishCoordinate,
                            gdouble yFinishCoordinate) {
    if (position.first ≥ xStartCoordinate − IMG_SIZE_IN_PX ∧
        position.first ≤ xFinishCoordinate + IMG_SIZE_IN_PX ∧
        position.second ≥ yStartCoordinate − IMG_SIZE_IN_PX ∧
        position.second ≤ yFinishCoordinate + IMG_SIZE_IN_PX) {
```

```
        selected = true;
        unitsSelected = true;
    }
}

bool Unit::isShooting() {
    return shooting;
}

TeamEnum Unit::getTeam() {
    return team;
}

RotationsEnum Unit::getRotation() {
    return rotation;
}

unsigned int Unit::getXCoordinate() {
    return position.first;
}

unsigned int Unit::getYCoordinate() {
    return position.second;
}

UnitsEnum Unit::getType() {
    return unitType;
}

ActionsEnum Unit::getAction() {
    return actionType;
}

bool Unit::is_selected() {
    return selected;
}

void Unit::unselect() {
    selected = false;
}

Unit::Unit() {
    id = 0;
}

void Unit::update_owner(const std::string &owner) {
    this->owner = owner;
}

std::string Unit::get_owner() {
    return owner;
}

int Unit::get_max_hp() {
    return totalLife;
}

int Unit::get_hp() {
    return lifeLeft;
}

void Unit::update_unit_name(const std::string &name) {
    unit_name = name;
}

std::string Unit::get_unit_name() {
```

```
    return unit_name;
}

void Unit::update_hp(unsigned int hp) {
    lifeLeft = hp;
}
```

```cpp
#ifndef Z_TPGRUPAL_SERVERMESSENGER_H
#define Z_TPGRUPAL_SERVERMESSENGER_H

#include <mutex>
#include <string>
#include <messenger.h>
#include "../common/socket.h"

/* Handles communication with the server */
class ServerMessenger {
    Messenger messenger;
    std::mutex send_m, recv_m;
public:
    /* Constructor: connects to the given addr/port combination */
    explicit ServerMessenger(Socket &s);

    void send(const std::string &message);

    std::string receive();

    void kill();

    ServerMessenger(ServerMessenger& other);
};


#endif //Z_TPGRUPAL_SERVERMESSENGER_H
```

```cpp

#include <string>
#include "ServerMessenger.h"
#include "../common/Lock.h"

ServerMessenger::ServerMessenger(Socket &s) :
        messenger(s) {
}

void ServerMessenger::send(const std::string &message) {
    Lock l(send_m);
    messenger.sendMessage(message);
}

std::string ServerMessenger::receive() {
    Lock l(recv_m);
    return messenger.recieveMessage();
}

void ServerMessenger::kill() {
    messenger.shutdown();
}

ServerMessenger::ServerMessenger(ServerMessenger &other) :
    messenger(other.messenger)
{
}
```

```cpp
#ifndef Z_TPGRUPAL_UNITPANEL_H
#define Z_TPGRUPAL_UNITPANEL_H


#include <gtkmm/box.h>
#include <gtkmm/builder.h>
#include <gtkmm/button.h>
#include <gtkmm/label.h>
#include <gtkmm/image.h>
#include "../enums/UnitsEnum.h"
#include "../enums/TeamEnum.h"

class UnitPanel : public Gtk::Box {
    Gtk::Label *owner;
    Gtk::Label *max_hp_label;
    Gtk::Label *hp_label;
    Gtk::Label *name_label;
    Gtk::Image* portrait;
public:
    UnitPanel(BaseObjectType *cobject,
              const Glib::RefPtr<Gtk::Builder> &builder);

    std::string get_label();

    void set_name(std::string name);

    void set_hp(int hp);

    void set_max_hp(int hp);

    void set_owner(const std::string& owner);

    void update_portrait(UnitsEnum unit, TeamEnum team);
};


#endif //Z_TPGRUPAL_UNITPANEL_H
```

```cpp
#include <iostream>
#include "UnitPanel.h"
#include "../enums/UnitsEnum.h"
#include "../enums/TeamEnum.h"

#define PORTRAITS "res/portraits/"
const std::map<UnitsEnum, std::string> units = {
        {UnitsEnum::GRUNT, std::string("grunt")},
        {UnitsEnum::PSYCHO, std::string("psycho"),},
        {UnitsEnum::TOUGH, std::string("tough")},
        {UnitsEnum::PYRO, std::string("pyro")},
        {UnitsEnum::SNIPER, std::string("sniper")},
        {UnitsEnum::LASER, std::string("laser")},
        {UnitsEnum::GENERIC_ROBOT, std::string("generic_robot")},
        {UnitsEnum::JEEP, std::string("jeep")},
        {UnitsEnum::MEDIUM_TANK, std::string("medium_tank")},
        {UnitsEnum::LIGHT_TANK, std::string("light_tank")},
        {UnitsEnum::HEAVY_TANK, std::string("heavy_tank")},
        {UnitsEnum::MML, std::string("mml")}
};

const std::map<TeamEnum, std::string> teams = {
        {TeamEnum::BLUE, "blue"},
        {TeamEnum::GREEN, "green"},
        {TeamEnum::RED, "red"},
        {TeamEnum ::YELLOW, "yellow"}
};

UnitPanel::UnitPanel(BaseObjectType *cobject,
                     const Glib::RefPtr<Gtk::Builder> &builder) :
    Gtk::Box(cobject)
{
    builder->get_widget("OwnerLabel", owner);
    builder->get_widget("MaxHPLabel", max_hp_label);
    builder->get_widget("HPLabel", hp_label);
    builder->get_widget("NameLabel", name_label);
    builder->get_widget("Portrait", portrait);

}

std::string UnitPanel::get_label() {
    return "Unit";
}

void UnitPanel::set_name(std::string name) {
    name_label->set_text(name);
}

void UnitPanel::set_hp(int hp) {
    hp_label->set_text(std::to_string(hp));
}

void UnitPanel::set_max_hp(int hp) {
    max_hp_label->set_text(std::to_string(hp));
}

void UnitPanel::set_owner(const std::string &owner) {
    this->owner->set_text(owner);
}

void UnitPanel::update_portrait(UnitsEnum unit, TeamEnum team) {
    std::string unit_name = units.find(unit)->second;
    std::string color = teams.find(team)->second;
    std::string path = PORTRAITS + unit_name + "_" + color + ".png";
    portrait->set(path);
    set_name(unit_name);
}

```

```
#ifndef Z_TPGRUPAL_BUILDINGPANEL_H
#define Z_TPGRUPAL_BUILDINGPANEL_H


#include <gtkmm/box.h>
#include <gtkmm/image.h>
#include <gtkmm/builder.h>
#include <string>
#include <gtkmm/button.h>
#include "../enums/TeamEnum.h"

class BuildingPanel : public Gtk::Box {
    Gtk::Image* building;
    Gtk::Image* unit;
    const std::string label = "Factory";
    Gtk::Button* prev;
    Gtk::Button* next;
    Gtk::Button* create;

    Gtk::Label* max_hp_label;
    Gtk::Label* hp_label;
    Gtk::Label* owner_label;
    Gtk::Label* unit_type;
    Gtk::Label* unit_hp;
    Gtk::Label* unit_fire_rate;

    Gtk::Label* minutes;
    Gtk::Label* seconds;

    int max_hp;
    int hp;
    std::string owner;
    std::string type;
    int fire_rate;
    std::pair<int, int> time_left;
public:
    BuildingPanel(BaseObjectType* cobject,
                  const Glib::RefPtr<Gtk::Builder>& builder);

    const std::string& get_label();
    Gtk::Button* next_button();
    Gtk::Button* create_button();

    void change_unit(std::string& path);


    void set_hp(int hp);

    void set_max_hp(int hp);

    void set_owner(const std::string& owner);

    void set_unit_hp(int hp);

    void set_unit_type(const std::string& type, TeamEnum team);

    void set_unit_fire_rate(int fire_rate);

    void set_time_left(std::pair<int, int> time);
    Gtk::Button * prev_button();

protected:
    bool update_labels();
};


```

```
#endif //Z_TPGRUPAL_BUILDINGPANEL_H
```

```
#include <giomm.h>
#include "BuildingPanel.h"
#include "../windows/GameWindow.h"
#define PORTRAITS_PATH "res/portraits/"

#define TIMEOUT 100

BuildingPanel::BuildingPanel(BaseObjectType *cobject,
                             const Glib::RefPtr<Gtk::Builder> &builder) :
    Gtk::Box(cobject) {
    builder->get_widget("FactoryImage", building);
    builder->get_widget("FactoryUnitImage", unit);
    builder->get_widget("PrevUnitButton", prev);
    builder->get_widget("NextUnitButton", next);
    builder->get_widget("FactoryCreateButton", create);

    builder->get_widget("FactoryMaxHPLabel", max_hp_label);
    builder->get_widget("FactoryHPLabel", hp_label);
    builder->get_widget("FactoryOwnerLabel", owner_label);

    builder->get_widget("FactoryUnitHPLabel", unit_hp);
    builder->get_widget("FactoryFireRateLabel", unit_fire_rate);
    builder->get_widget("FactoryTypeLabel", unit_type);

    builder->get_widget("FactoryMinutesLabel", minutes);
    builder->get_widget("FactorySecondsLabel", seconds);
    building->set("res/buildings/base_city.png");
    Glib::signal_timeout().connect(sigc::mem_fun(*this,
                                                 &BuildingPanel::update_labels),
                                   TIMEOUT);
}


const std::map<TeamEnum, std::string> teams = {
        {TeamEnum::NEUTRAL, "blue"},
        {TeamEnum::BLUE, "blue"},
        {TeamEnum::GREEN, "green"},
        {TeamEnum::RED, "red"},
        {TeamEnum::YELLOW, "yellow"}
};

const std::string &BuildingPanel::get_label() {
    return label;
}

Gtk::Button *BuildingPanel::next_button() {
    return next;
}

void BuildingPanel::change_unit(std::string &path) {
    unit->set(path);
}

Gtk::Button *BuildingPanel::create_button() {
    return create;
}

void BuildingPanel::set_hp(int hp) {
    this->hp = hp;
}

void BuildingPanel::set_max_hp(int hp) {
    this->max_hp = hp;
}

void BuildingPanel::set_owner(const std::string &owner) {
```

```cpp
    this->owner = owner;
}

void BuildingPanel::set_unit_hp(int hp) {
    unit_hp->set_text(std::to_string(hp));
}

void BuildingPanel::set_unit_type(const std::string &type, TeamEnum team) {
    std::string color = teams.find(team)->second;
    this->type = type;
    unit->set(PORTRAITS_PATH + type + "_" + color + ".png");
}

void BuildingPanel::set_unit_fire_rate(int fire_rate) {
    this->fire_rate = fire_rate;
}

Gtk::Button* BuildingPanel::prev_button() {
    return prev;
}

void BuildingPanel::set_time_left(std::pair<int, int> time) {
    this->time_left = time;
}

bool BuildingPanel::update_labels() {
    hp_label->set_text(std::to_string(hp));
    max_hp_label->set_text(std::to_string(max_hp));
    minutes->set_text(std::to_string(time_left.first));
    seconds->set_text(std::to_string(time_left.second));
    unit_fire_rate->set_text(std::to_string(fire_rate));
    unit_type->set_text(type);
    return true;
}
```

```cpp
#ifndef Z_TPGRUPAL_NATURE_H
#define Z_TPGRUPAL_NATURE_H


#include <utility>
#include "enums/NatureEnum.h"

class Nature {
    NatureEnum type;
    std::pair<unsigned int, unsigned int> position;
    int id;

public:
    Nature(NatureEnum type, std::pair<unsigned int, unsigned int> position,
               int id);

    std::pair<unsigned int, unsigned int> getPosition();

    NatureEnum getType();
};


#endif //Z_TPGRUPAL_NATURE_H
```

```cpp
#include "Nature.h"

Nature::Nature(NatureEnum type, std::pair<unsigned int, unsigned int> position,
               int id) :
        type(type), position(position), id(id){
}

std::pair<unsigned int, unsigned int> Nature::getPosition() {
    return position;
}

NatureEnum Nature::getType() {
    return type;
}
```

```cpp
#ifndef Z_TPGRUPAL_MAPMONITOR_H
#define Z_TPGRUPAL_MAPMONITOR_H


#include <mutex>
#include "Map.h"
#include "enums/TeamEnum.h"
#include <string>

class MapMonitor {
private:
    Map map;
    std::vector<std::string> players;

    std::mutex m;

    bool winner = false;
    bool loser = false;
public:

    void setCell(unsigned int xCoordinate,
                 unsigned int yCoordinate,
                 std::string terrainType);

    void initializeMap(unsigned int xSize, unsigned int ySize);

    unsigned int getXSize();

    unsigned int getYSize();

    std::string getTerrainTypeAt(unsigned int xCoordinate,
                                 unsigned int yCoordinate);

    void addNature(Nature nature);

    void markAsSelectedInRange(bool& terrainSelected, double xStartCoordinate,
                               double yStartCoordinate,
                               double xFinishCoordinate,
                               double yFinishCoordinate);

    std::vector<Nature>
    getNatureToDraw(unsigned int minX, unsigned int maxX, unsigned int minY,
                    unsigned int maxY);

    void update_players(const std::vector<std::string>& names);
    int get_player_id(const std::string& player);

    void finish_winner();
    void finish_loser();

    bool is_winner();
    bool is_loser();

    void clear();

    void update_territory(const int &id, const TeamEnum &team, const int &x,
                          const int &y);

    std::map<int, std::pair<TeamEnum, std::pair<unsigned int, unsigned int>>>
    getFlags();

    int get_flag_at(int x, int y);

};


```

```cpp
#endif //Z_TPGRUPAL_MAPMONITOR_H
```

```cpp
#include "MapMonitor.h"
#include <Lock.h>


void MapMonitor::setCell(unsigned int xCoordinate, unsigned int yCoordinate,
                         std::string terrainType) {
    Lock l(m);
    map.setCell(xCoordinate, yCoordinate, terrainType);
}

void MapMonitor::initializeMap(unsigned int xSize, unsigned int ySize) {
    Lock l(m);
    map.initializeMap(xSize, ySize);
}

unsigned int MapMonitor::getXSize() {
    Lock l(m);
    return map.getXSize();
}

unsigned int MapMonitor::getYSize() {
    Lock l(m);
    return map.getYSize();
}

std::string MapMonitor::getTerrainTypeAt(unsigned int xCoordinate,
                                         unsigned int yCoordinate) {
    Lock l(m);
    return map.getTerrainTypeAt(xCoordinate, yCoordinate);
}

void MapMonitor::markAsSelectedInRange(bool& terrainSelected, double
                                       xStartCoordinate,
                                       double yStartCoordinate,
                                       double xFinishCoordinate,
                                       double yFinishCoordinate) {
    //todo implementar esto que falta ver el tema de accidentales sobre el mapa.
}

void MapMonitor::addNature(Nature nature) {
    Lock l(m);
    map.addNature(nature);
}

std::vector<Nature>
MapMonitor::getNatureToDraw(unsigned int minX, unsigned int maxX,
                            unsigned int minY, unsigned int maxY) {
    Lock l(m);
    std::vector<Nature> returnVector;

    for (Nature &nature : map.getNature()) {
        if (nature.getPosition().first ≥ minX and
            nature.getPosition().first ≤ maxX and
            nature.getPosition().second ≥ minY and
            nature.getPosition().second ≤ maxY) {
            returnVector.emplace_back(nature);
        }
    }
    return returnVector;
}

void MapMonitor::update_players(const std::vector<std::string> &names) {
    Lock l(m);
    this→players = names;
}

```

```
int MapMonitor::get_player_id(const std::string &player) {
    int id = 1; // id 0 is neutral
    Lock l(m);
    auto it = players.begin();
    for (; it ≠ players.end(); ++it) {
        if (*it ≡ player) {
            break;
        }
        id++;
    }
    if (it ≡ players.end()) {
        return 0;
    }
    return id; // Not found, return NEUTRAL id
}

void MapMonitor::finish_winner() {
    Lock l(m);
    winner = true;
}

void MapMonitor::finish_loser() {
    Lock l(m);
    loser = true;
}

bool MapMonitor::is_winner() {
    Lock l(m);
    return winner;
}

bool MapMonitor::is_loser() {
    Lock l(m);
    return loser;
}


void MapMonitor::clear() {
    Lock l(m);
    loser = false;
    winner = false;
    map.clear();
    players.clear();

}

void MapMonitor::update_territory(const int &id, const TeamEnum &team,
                                  const int &x, const int &y) {
    Lock l(m);
    map.update_territory(id, team, x, y);
}

std::map<int, std::pair<TeamEnum, std::pair<unsigned int, unsigned int>>>
MapMonitor::getFlags() {
    Lock l(m);
    return map.getFlags();
}

int MapMonitor::get_flag_at(int x, int y) {
    Lock l(m);
    return map.get_flag_at(x, y);
}
```

```
#ifndef Z_TPGRUPAL_MAP_H
#define Z_TPGRUPAL_MAP_H


#include <vector>
#include "Cell.h"
#include "Nature.h"
#include "enums/TeamEnum.h"
#include <string>
#include <map>

class Map {
private:
    std::vector<std::vector<Cell>> baseMap;
    std::vector<Nature> nature;
    /**vector storing flags' positions.
    * Key = ID,
    * Value = pair team, coordinates
    */
    std::map<int, std::pair<TeamEnum, std::pair<unsigned int, unsigned int>>>
            flags;

public:
    void initializeMap(unsigned int xSize, unsigned int ySize);

    void addNature(Nature nature);

    /* vector storing all nature, i.e. rocks, and all which is not plain
     * terrain or buildings and independent units, in the map. */
    std::vector<Nature> getNature();

    void setCell(unsigned int xCoordinate,
                 unsigned int yCoordinate,
                 std::string terrainType);

    unsigned int getXSize();

    unsigned int getYSize();

    std::string getTerrainTypeAt(unsigned int xCoordinate,
                                 unsigned int yCoordinate);


    void clear();

    void update_territory(const int &id, const TeamEnum &team, const int &x,
                          const int &y);

    std::map<int, std::pair<TeamEnum, std::pair<unsigned int, unsigned int>>>
    getFlags();

    int get_flag_at(int x, int y);
};


#endif //Z_TPGRUPAL_MAP_H
```

```
#include "Map.h"

#define LENIENCY 3

void Map::initializeMap(unsigned int xSize, unsigned int ySize) {
    baseMap.resize(xSize);
    for (int i = 0; i < xSize; ++i) {
        baseMap[i].resize(ySize);
    }
}

void Map::setCell(unsigned int xCoordinate, unsigned int yCoordinate,
                  std::string terrainType) {
    baseMap.at(xCoordinate).at(yCoordinate).assignTerrainType(terrainType);
}

unsigned int Map::getXSize() {
    return (unsigned int) baseMap.size();
}

unsigned int Map::getYSize() {
    if (¬baseMap.size()) {
        return 0;
    }
    return (unsigned int) baseMap[0].size();
}

std::string
Map::getTerrainTypeAt(unsigned int xCoordinate, unsigned int yCoordinate) {
    return baseMap.at(xCoordinate).at(yCoordinate).getTerrainType();
}

void Map::addNature(Nature nature) {
    this→nature.emplace_back(nature);
}

std::vector<Nature> Map::getNature() {
    return nature;
}

void Map::clear() {
    baseMap.clear();
    nature.clear();
    flags.clear();
}

void Map::update_territory(const int &id, const TeamEnum &team, const int &x,
                           const int &y) {
    flags[id] = {team,{x,y}};
}

std::map<int, std::pair<TeamEnum, std::pair<unsigned int, unsigned int>>>
Map::getFlags() {
    return flags;
}

int Map::get_flag_at(int x, int y) {
    for (auto flag : flags) {
        std::pair<int, int> position = flag.second.second;
        int x_abs = abs(x − position.first);
        int y_abs = abs(y − position.second);
        if (x_abs ≤ LENIENCY ∧ y_abs ≤ LENIENCY) {
            return flag.first;
        }
    }
    return −1;
```

```
}
```

```cpp
#include <gtkmm.h>
#include <iostream>
#include "ClientThread.h"
#include "GameBuilder.h"
#include "Game.h"
#include <split.h>

#define SUCCESSRETURNCODE 0
#define ERRORCODE 1
int main(int argc, char **argv) {
    try {
        auto app = Gtk::Application::create();

        GameBuilder builder;
        InitialWindow *window = builder.get_initial_window();
        app->run(*window);

        // Once initial window closes, we fetch the socket
        std::shared_ptr<ServerMessenger> m = window->get_socket();
        std::string player_name = window->get_username();
        if (m) {
            ServerMessenger messenger = *m.get();
            MapMonitor mapMonitor;
            UnitsMonitor units_monitor;
            BuildingsMonitor buildingsMonitor;
            ClientThread clientThread(units_monitor, buildingsMonitor,
                                      mapMonitor, messenger, builder);
            clientThread.start();
            bool keep_playing = true;
            while(keep_playing) {
                // Starts the game
                Game g(builder, messenger, player_name, mapMonitor,
                       units_monitor, buildingsMonitor);

                // Game finishes
                keep_playing = g.get_play_again_status();
            }
            clientThread.finish();
            clientThread.join();
        }

        return SUCCESSRETURNCODE;
    } catch (std::exception const &ex) {
        std::cerr << ex.what() << std::endl;
        return ERRORCODE;
    }
}
```

```cpp
#ifndef Z_TPGRUPAL_GAME_H
#define Z_TPGRUPAL_GAME_H


#include "GameBuilder.h"
#include "ServerMessenger.h"
#include "windows/ResultWindow.h"

class Game {
    std::string me;

    ServerMessenger &messenger;
    MapMonitor& mapMonitor;
    UnitsMonitor& units_monitor;
    BuildingsMonitor& buildingsMonitor;

    // Windows of different game stages
    MenuWindow* menu;
    LobbyWindow* lobby;
    GameWindow* game;
    ResultWindow* result;

    bool play_again = false;
public:
    Game(GameBuilder& builder, ServerMessenger& server_messenger,
         const std::string& player_name, MapMonitor& map,
         UnitsMonitor& units, BuildingsMonitor& buildings);

    bool get_play_again_status();

    ~Game();
private:
    void start_menu();

    void start_lobby() const;

    void start_game(const std::vector<std::string> &names);

    void results_screen(bool winner, bool loser);
};


#endif //Z_TPGRUPAL_GAME_H
```

```cpp
#include "Game.h"
#include "ClientThread.h"

void Game::start_game(const std::vector<std::string> &names) {
    game→update_name(me);
    // Start up the game
    game→setResources(&units_monitor, &buildingsMonitor,
                          &mapMonitor, &messenger, me);

    auto app = Gtk::Application::create();
    app→run(*game);
}

Game::Game(GameBuilder &builder, ServerMessenger &server_messenger,
           const std::string& player_name, MapMonitor& map,
           UnitsMonitor& units, BuildingsMonitor& buildings) :
    messenger(server_messenger),
    me(player_name),
    menu(builder.get_menu_window()),
    lobby(builder.get_lobby_window()),
    game(builder.get_window()),
    result(builder.get_result_window()),
    mapMonitor(map),
    units_monitor(units),
    buildingsMonitor(buildings)
{

    start_menu();

    if (menu→joined_lobby()) {
        start_lobby();
        if (lobby→game_started()) {
            std::vector<std::string> names = lobby→get_player_names();
            mapMonitor.update_players(names);
            start_game(names);
        }
    }

    bool winner = mapMonitor.is_winner();
    bool loser = mapMonitor.is_loser();
    results_screen(winner, loser);
}

void Game::start_lobby() const {
    lobby→set_messenger(messenger);
    auto app = Gtk::Application::create();
    app→run(*lobby);
}

void Game::start_menu() {
    menu→load_messenger(&messenger);
    messenger.send("lobbyinfo");
    auto app = Gtk::Application::create();
    app→run(*menu);
}

void Game::results_screen(bool winner, bool loser) {
    if (¬winner ∧ ¬loser) { // Played closed the window before game was over
        play_again = false;
        return;
    } else if (winner) {
        result→display_win_screen();
    } else {
        result→display_lose_screen();
    }

```

```cpp
    auto app = Gtk::Application::create();
    app→run(*result);
    bool play_again = result→go_back_to_menu();
    if (play_again) {
        messenger.send("returntomenu");
    }
    this→play_again = play_again;
}

bool Game::get_play_again_status() {
    return play_again;
}

Game::~Game() { // Game finishes, clear assets
    units_monitor.clear();
    mapMonitor.clear();
    buildingsMonitor.clear();
}
```

```cpp
#ifndef Z_TPGRUPAL_GAMEBUILDER_H
#define Z_TPGRUPAL_GAMEBUILDER_H

#include <gtkmm.h>
#include <string>

#include "windows/GameWindow.h"
#include "BuildingsMonitor.h"
#include "MapMonitor.h"
#include "ServerMessenger.h"
#include "windows/InitialWindow.h"
#include "windows/LobbyWindow.h"
#include "windows/MenuWindow.h"
#include "windows/ResultWindow.h"

class GameBuilder {
    InitialWindow *init_window;
    MenuWindow* menu_window;
    LobbyWindow* lobby_window;
    GameWindow *window;
    ResultWindow* result_window;
    Glib::RefPtr<Gtk::Builder> refBuilder;

public:
    GameBuilder();
    ~GameBuilder();

    // returns the generated window
    GameWindow *get_window();

    InitialWindow *get_initial_window();
    LobbyWindow* get_lobby_window();
    MenuWindow* get_menu_window();
    ResultWindow* get_result_window();

private:
    void start();
    void clean();
};


#endif //Z_TPGRUPAL_GAMEWINDOW_H
```

```cpp
#include <iostream>
#include "GameBuilder.h"
#include "windows/ResultWindow.h"


GameBuilder::GameBuilder() :
    init_window(nullptr),
    menu_window(nullptr),
    lobby_window(nullptr),
    window(nullptr),
    result_window(nullptr)
{
    //Load the GtkBuilder file and instantiate its widgets:
    start();

}

void GameBuilder::start() {
    this→refBuilder = Gtk::Builder::create();
    try {
        this→refBuilder→add_from_file("Z.glade");
    }
    catch (const Glib::FileError &ex) {
        std::cerr << "FileError: " << ex.what() << std::endl;
        return;
    }
    catch (const Glib::MarkupError &ex) {
        std::cerr << "MarkupError: " << ex.what() << std::endl;
        return;
    }
    catch (const Gtk::BuilderError &ex) {
        std::cerr << "BuilderError: " << ex.what() << std::endl;
        return;
    }

    // Save the widget refs in the class attributes
    this→refBuilder→get_widget_derived("GameWindow", this→window);
    this→refBuilder→get_widget_derived("InitialWindow", this→init_window);
    this→refBuilder→get_widget_derived("LobbyWindow", this→lobby_window);
    this→refBuilder→get_widget_derived("MenuWindow", this→menu_window);
    this→refBuilder→get_widget_derived("ResultWindow", this→result_window);
}


GameBuilder::~GameBuilder() {
    clean();
}

GameWindow *GameBuilder::get_window() {
    return window;
}

InitialWindow *GameBuilder::get_initial_window() {
    return init_window;
}

LobbyWindow *GameBuilder::get_lobby_window() {
    return lobby_window;
}

MenuWindow *GameBuilder::get_menu_window() {
    return menu_window;
}

ResultWindow *GameBuilder::get_result_window() {
    return result_window;
}

void GameBuilder::clean() {
    if (window) {
        delete window;
    }
    if (init_window) {
        delete init_window;
    }
    if (lobby_window) {
        delete lobby_window;
    }
    if (menu_window) {
        delete menu_window;
    }
    if (result_window) {
        delete result_window;
    }
}



```

```cpp
#ifndef Z_TPGRUPAL_GAMEAREA_H
#define Z_TPGRUPAL_GAMEAREA_H


#include <gtkmm/drawingarea.h>
#include <utility>
#include "BuildingsMonitor.h"
#include "MapMonitor.h"
#include "Camera.h"
#include "enums/TeamEnum.h"
#include "enums/ActionsEnum.h"
#include "enums/UnitsEnum.h"
#include "enums/BuildingsEnum.h"
#include "enums/RotationsEnum.h"
#include "Counter.h"
#include "UnitsMonitor.h"
#include <map>
#include <string>
#include <vector>

class GameArea : public Gtk::DrawingArea {
private:
    /* shared resources */
    UnitsMonitor *unitsMonitor;
    BuildingsMonitor *buildingsMonitor;
    MapMonitor *mapMonitor;

    std::string owner;

    Camera camera;

    /* general resources */
    std::map<std::string, Glib::RefPtr<Gdk::Pixbuf>> tiles;

    std::map<NatureEnum, Glib::RefPtr<Gdk::Pixbuf>> nature;
    Glib::RefPtr<Gdk::Pixbuf> someImg;

    /* map holding all units imgs */
    std::map<TeamEnum,
            std::map<UnitsEnum,
                    std::map<ActionsEnum,
                            std::map<RotationsEnum,
                                    std::vector<Glib::RefPtr<Gdk::Pixbuf>>>>>>
            unitsAnimations;

    std::map<RotationsEnum, std::vector<Glib::RefPtr<Gdk::Pixbuf>>> jeepTires;

    /* BUILDINGS RESOURCES */
    std::map<BuildingsEnum, std::vector<Glib::RefPtr<Gdk::Pixbuf>>> buildings;

    /* map holding all flags */
    std::map<TeamEnum, std::vector<Glib::RefPtr<Gdk::Pixbuf>>> flags;

    /* declare counter used to know which of the flag imgs
     * which compose the flag’s animation should be showed. This counters are
     * updated every time on_draw() is called.
     * */
    Counter flagCounter;
    Counter standingRobotCounter;
    Counter walkingRobotCounter;
    Counter shootingRobotCounter;
    Counter jeepCounter;
    Counter tireCounter;
    Counter tankCounter;
    Counter mmlCounter;
    Counter buildingsCounter;

    bool move_cmd = false;
    std::pair<int, int> coords;

    /* DRAWING METHODS */
    void drawBaseMap(const Cairo::RefPtr<Cairo::Context> &cr);

    void drawTileAt(const Cairo::RefPtr<Cairo::Context> &cr,
                    unsigned int xTileCoordinate, unsigned int yTileCoordinate,
                    std::string terrainType);

    void drawBuildingsInView(const Cairo::RefPtr<Cairo::Context> &cr);


    /* Event handling methods */
    bool on_key_press_event(GdkEventKey *event) override;

    bool on_button_press_event(GdkEventButton *event) override;

    bool on_button_release_event(GdkEventButton *event) override;

    /* vars. for event handling */
    /* this coordinates are sytem coordinatess */
    gdouble xStartCoordinate;
    gdouble xFinishCoordinate;
    gdouble yStartCoordinate;
    gdouble yFinishCoordinate;
    bool selectionMade;

    /* unitsSelected is true if the players' units are selected. This is used
     * to manage user clicks.
     * Turns true in Unit::markAsSelectedInRange, when some unit has been
     * selected.
     * Turns to false at the end of GameArea::processSelection(), when the
     * selection has already been processed. */
    bool unitsSelected;
    bool buildingSelected;
    bool terrainSelected;

    void makeSelection();

    /* FLAG res loading */
    void loadFlagAnimations();

    /* UNIT FIRE res loading */
    void loadUnitsResources();

    void loadGruntFireAnimations();

    void loadLaserFireAnimations();

    void loadPsychoFireAnimations();

    void loadPyroFireAnimations();

    void loadSniperFireAnimations();

    void loadToughFireAnimations();

    void loadBlueGruntFireAnimations();

    void loadGreenGruntFireAnimations();

    void loadRedGruntFireAnimations();

    void loadYellowGruntFireAnimations();
```

```

    void loadBlueLaserFireAnimations();

    void loadGreenLaserFireAnimations();

    void loadRedLaserFireAnimations();

    void loadYellowLaserFireAnimations();

    void loadBluePsychoFireAnimations();

    void loadGreenPsychoFireAnimations();

    void loadRedPsychoFireAnimations();

    void loadYellowPsychoFireAnimations();

    void loadBluePyroFireAnimation();

    void loadGreenPyroFireAnimation();

    void loadRedPyroFireAnimation();

    void loadYellowPyroFireAnimation();

    void loadBlueSniperFireAnimations();

    void loadGreenSniperFireAnimations();

    void loadRedSniperFireAnimations();

    void loadYellowSniperFireAnimations();

    void loadBlueToughFireAnimations();

    void loadGreenToughFireAnimations();

    void loadRedToughFireAnimations();

    void loadYellowToughFireAnimations();

    /* UNIT WALK res loading */
    void loadBlueWalkingAnimations();

    void loadGreenWalkingAnimations();

    void loadRedWalkingAnimations();

    void loadYellowWalkingAnimations();

    /* UNIT STAND res loading */
    void loadBlueStandingAnimations();

    void loadGreenStandingAnimations();

    void loadRedStandingAnimations();

    void loadYellowStandingAnimations();

    /* BUILDING res loading */
    void loadBuildingsResources();

    /* VEHICLES res loading */
    void loadTiresAnimations();

    void loadJeepTires();

    void loadNeuterVehiclesAnimations();

    void loadNeuterJeepAnimations();

    void loadBlueVehiclesAnimations();

    void loadGreenVehiclesAnimations();

    void loadRedVehiclesAnimations();

    void loadYellowVehiclesAnimations();

    void loadBlueJeepAnimations();

    void loadBlueLightTankAnimations();

    void loadBlueMediumTankAnimations();

    void loadBlueMMLAnimations();

    void loadBlueHeavyTankAnimations();

    void loadGreenJeepAnimations();

    void loadGreenLightTankAnimations();

    void loadGreenMediumTankAnimations();

    void loadGreenMMLAnimations();

    void loadGreenHeavyTankAnimations();

    void loadRedJeepAnimations();

    void loadRedLightTankAnimations();

    void loadRedMediumTankAnimations();

    void loadRedMMLAnimations();

    void loadRedHeavyTankAnimations();

    void loadYellowJeepAnimations();

    void loadYellowLightTankAnimations();

    void loadYellowMediumTankAnimations();

    void loadYellowMMLAnimations();

    void loadYellowHeavyTankAnimations();

    void drawJeepTires(const Cairo::RefPtr<Cairo::Context> &cr,
                       unsigned int xGraphicCoordinate,
                       unsigned int yGraphicCoordinate,
                       RotationsEnum rotation);

    void drawUnit(TeamEnum team, UnitsEnum unitType, ActionsEnum actionType,
                  RotationsEnum rotation, unsigned short unitCounter,
                  const Cairo::RefPtr<Cairo::Context> &cr,
                  unsigned int xGraphicCoordinate,
                  unsigned int yGraphicCoordinate);

    void loadResources();

```

```
    /**
     * draws all units that are in camera’s scope
     * @param cr receive smart pointer to cairo context
     */
    void drawUnitsInMap(const Cairo::RefPtr<Cairo::Context> &cr);

    /**
     * counters used to know whic img of each animation should be drawn are
     * updated. This method should get called once per frame, otherwise it is
     * not guaranteed that animations will be correctly drawn.
     */
    void updateCounters();

protected:
    bool on_draw(const Cairo::RefPtr<Cairo::Context> &cr) override;

public:
    virtual ~GameArea();

    GameArea(BaseObjectType *cobject,
             const Glib::RefPtr<Gtk::Builder> &builder);

    /**
     * initialize shared resources.
     */
    void
    setResources(UnitsMonitor *playersMonitor,
                 BuildingsMonitor *buildingsMonitor,
                 MapMonitor *mapMonitor, std::string owner);

    void processClick();

    void initializeCounters();

    unsigned short getCounter(Unit &unit) const;

    void processUnitToDrawEnums(UnitsEnum &unitType, ActionsEnum &actionType,
                                RotationsEnum &rotation) const;

    unsigned int cameraToRealMapX(unsigned int coordinate);

    unsigned int cameraToRealMapY(unsigned int coordinate);

    std::pair<int, int> get_coords();

    unsigned int screenMapToCameraX(gdouble coordinate);

    unsigned int screenMapToCameraY(gdouble coordinate);

    void setMapData();

    void drawBuilding(BuildingsEnum buildingType, unsigned short counter,
                      TeamEnum team,
                      const Cairo::RefPtr<Cairo::Context> &cr,
                      unsigned int xGraphicCoordinate,
                      unsigned int yGraphicCoordinate);

    bool unitIsRobot(UnitsEnum unitType);

    void drawFlag(const TeamEnum &team, const Cairo::RefPtr<Cairo::Context> &cr,
                  unsigned int xGraphicCoordinate,
                  unsigned int yGraphicCoordinate) const;

    void drawTerritoriesFlagsInView(const Cairo::RefPtr<Cairo::Context> &cr);

    bool unit_selected();
```

```cpp

    bool buildings_selected();

    void drawNatureInView(const Cairo::RefPtr<Cairo::Context> &cr);

    void
    drawNature(NatureEnum natureType, const Cairo::RefPtr<Cairo::Context> &cr,
               unsigned int x, unsigned int y);

    void loadMapResources();
};


#endif //Z_TPGRUPAL_GAMEAREA_H
```

```cpp
#include <gtkmm/builder.h>
#include <gdkmm.h>
#include <iostream>
#include "GameArea.h"
#include <giomm.h>

#define TILESIZE 16    //tile width in pixels. This define is also present
                        //in AddBuilding Command
#define NUMBER_OF_TILES_TO_SHOW 10

GameArea::GameArea(BaseObjectType *cobject,
                   const Glib::RefPtr<Gtk::Builder> &builder) :
        Gtk::DrawingArea(cobject),
        unitsMonitor(nullptr),
        buildingsMonitor(nullptr),
        mapMonitor(nullptr),
        unitsSelected(false),
        buildingSelected(false),
        coords({−1, −1}),
        /* camera is initialized with size 0,0 because we don’t
         * have this data yet */
        camera(TILESIZE, 0, 0, NUMBER_OF_TILES_TO_SHOW) {
    loadResources();
    initializeCounters();

    add_events(Gdk::EventMask::BUTTON_PRESS_MASK);
    add_events(Gdk::EventMask::BUTTON_RELEASE_MASK);
    add_events(Gdk::EventMask::KEY_PRESS_MASK);
    set_can_focus(true);
}

void GameArea::loadResources() {
    try {
        /* load flags animations */
        loadFlagAnimations();
        /* load units resources */
        loadUnitsResources();
        /* load buildings resources */
        loadBuildingsResources();

        loadMapResources();
    } catch (Glib::FileError e) {
        std::cerr << e.what();
    }
}

void GameArea::loadMapResources() {
    /* Load tiles */
    tiles["Tierra"] = Gdk::Pixbuf::create_from_file(
            "res/assets/tiles/tierra.png");
    tiles["Agua"] = Gdk::Pixbuf::create_from_file
            ("res/assets/tiles/agua.png");
    tiles["Lava"] = Gdk::Pixbuf::create_from_file(
            "res/assets/tiles/lava.png");
    /* Load nature items */
    nature[NatureEnum::ROCK] = Gdk::Pixbuf::create_from_file
            ("res/assets/nature/rock.png");
}

GameArea::~GameArea() {}

bool GameArea::on_draw(const Cairo::RefPtr<Cairo::Context> &cr) {
    drawBaseMap(cr);
    drawBuildingsInView(cr);
    drawTerritoriesFlagsInView(cr);
    drawUnitsInMap(cr);
    updateCounters();
    return true;
}

void GameArea::drawBaseMap(const Cairo::RefPtr<Cairo::Context> &cr) {
    /* check if map is emtpy*/
    if (mapMonitor→getXSize() ≡ 0 and mapMonitor→getYSize() ≡ 0) {
        return;
    }

    /* cameraPosition is given in pixels.
     * i,j indicate TILES. */
    for (unsigned int i = 0; i < NUMBER_OF_TILES_TO_SHOW; ++i) {
        for (unsigned int j = 0; j < NUMBER_OF_TILES_TO_SHOW; ++j) {
            drawTileAt(cr, i, j, mapMonitor→getTerrainTypeAt(
                    camera.getPosition().first / TILESIZE −
                    NUMBER_OF_TILES_TO_SHOW / 2 + i,
                    camera.getPosition().second / TILESIZE −
                    NUMBER_OF_TILES_TO_SHOW / 2 + j));
        }
    }
    drawNatureInView(cr);
}

void GameArea::drawNatureInView(const Cairo::RefPtr<Cairo::Context> &cr) {
    /* pointers (Nature*) are not used here because we are working with a shared
     * resource. This way, we copy the units we want to draw in a protected way,
     * and then we can draw without blocking other code. */
    std::vector<Nature> natureToDraw = mapMonitor→getNatureToDraw(
            camera.getPosition().first −
            (NUMBER_OF_TILES_TO_SHOW * TILESIZE) / 2,
            camera.getPosition().first +
            (NUMBER_OF_TILES_TO_SHOW * TILESIZE) / 2,
            camera.getPosition().second −
            (NUMBER_OF_TILES_TO_SHOW * TILESIZE) / 2,
            camera.getPosition().second +
            (NUMBER_OF_TILES_TO_SHOW * TILESIZE) / 2);

    for (auto &nature : natureToDraw) {
        /* call actual drawing method */
        drawNature(nature.getType(), cr,
                   cameraToRealMapX(camera.idealMapToCameraXCoordinate(
                           nature.getPosition().first)),
                   cameraToRealMapY(camera.idealMapToCameraYCoordinate(
                           nature.getPosition().second)));
    }
}

void GameArea::drawNature(NatureEnum natureType,
                          const Cairo::RefPtr<Cairo::Context> &cr,
                          unsigned int x, unsigned int y) {
    cr→save();
    Gdk::Cairo::set_source_pixbuf(cr, nature[natureType], x, y);
    cr→rectangle(x, y, nature[natureType]→get_width(),
                  nature[natureType]→get_height());
    cr→fill();
    cr→restore();
}

void GameArea::drawTileAt(const Cairo::RefPtr<Cairo::Context> &cr,
                          unsigned int xTileCoordinate,
                          unsigned int yTileCoordinate,
                          std::string terrainType) {
    cr→save();
    auto pixbuf = tiles.find(terrainType);
    if (pixbuf ≡ tiles.end()) {
```

```
        return;
    }
    const unsigned int xGraphicCoordinate = xTileCoordinate * get_width() /
                                            NUMBER_OF_TILES_TO_SHOW;
    const unsigned int yGraphicCoordinate = yTileCoordinate * get_height() /
                                            NUMBER_OF_TILES_TO_SHOW;
    Gdk::Cairo::set_source_pixbuf(cr, pixbuf→second,
                                  xGraphicCoordinate,
                                  yGraphicCoordinate);

    cr→rectangle(xGraphicCoordinate, yGraphicCoordinate,
                  get_width() / NUMBER_OF_TILES_TO_SHOW,
                  get_height() / NUMBER_OF_TILES_TO_SHOW);
    cr→fill();
    cr→restore();
}


void GameArea::drawJeepTires(const Cairo::RefPtr<Cairo::Context> &cr,
                             unsigned int xGraphicCoordinate,
                             unsigned int yGraphicCoordinate,
                             RotationsEnum rotation) {
    cr→save();
    /* first draw jeepTires */
    Gdk::Cairo::set_source_pixbuf(cr, jeepTires.at(rotation).
                                          at(tireCounter.getCounter()),
                                  xGraphicCoordinate, yGraphicCoordinate);
    cr→rectangle(xGraphicCoordinate, yGraphicCoordinate,
                  jeepTires.at(rotation).
                          at(tireCounter.getCounter())→get_width(),
                  jeepTires.at(rotation).
                          at(tireCounter.getCounter())→get_height());
    cr→fill();
    cr→restore();
}

void GameArea::drawUnitsInMap(const Cairo::RefPtr<Cairo::Context> &cr) {
    /* pointers (Unit*) are not used here because we are working with a shared
     * resource. This way, we copy the units we want to draw in a protected way,
     * and then we can draw without blocking other code. */
    std::vector<Unit> unitsToDraw = unitsMonitor→getUnitsToDraw(
            camera.getPosition().first −
            (NUMBER_OF_TILES_TO_SHOW * TILESIZE) / 2,
            camera.getPosition().first +
            (NUMBER_OF_TILES_TO_SHOW * TILESIZE) / 2,
            camera.getPosition().second −
            (NUMBER_OF_TILES_TO_SHOW * TILESIZE) / 2,
            camera.getPosition().second +
            (NUMBER_OF_TILES_TO_SHOW * TILESIZE) / 2);

    for (auto &unit : unitsToDraw) {
        /* check what is being drawn, and choose the counter appropriately. */
        unsigned short counter;
        counter = getCounter(unit);

        /* call actual drawing method */
        drawUnit(unit.getTeam(), unit.getType(), unit.getAction(),
                 unit.getRotation(),
                 counter, cr,
                 cameraToRealMapX(
                         camera.idealMapToCameraXCoordinate(
                                 unit.getXCoordinate())),
                 cameraToRealMapY(
                         camera.idealMapToCameraYCoordinate(
                                 unit.getYCoordinate())));
    }
```

```
}

void GameArea::drawUnit(TeamEnum team, UnitsEnum unitType,
                        ActionsEnum actionType,
                        RotationsEnum rotation, unsigned short unitCounter,
                        const Cairo::RefPtr<Cairo::Context> &cr,
                        unsigned int xGraphicCoordinate,
                        unsigned int yGraphicCoordinate) {
    cr→save();
    /* adapt given data to saved imgs. Applies to vehicles */
    if (unitType ≡ UnitsEnum::JEEP ∧
        rotation ≠ RotationsEnum::r090 ∧
        rotation ≠ RotationsEnum::r270) {
        /* rotations 090 and 270 dont have tires */
        drawJeepTires(cr, xGraphicCoordinate, yGraphicCoordinate, rotation);
    }
    processUnitToDrawEnums(unitType, actionType, rotation);

    auto team_map = unitsAnimations.find(team);
    if (team_map ≡ unitsAnimations.end()) {
        std::cerr << "Drawing failed at finding valid team" << std::endl;
    }

    auto unit_map = team_map→second.find(unitType);
    if (unit_map ≡ team_map→second.end()) {
        std::cerr << "Drawing failed at finding valid unitType" << std::endl;
    }

    auto actions_map = unit_map→second.find(actionType);
    if (actions_map ≡ unit_map→second.end()) {
        std::cerr << "Drawing failed at finding valid actionType" << std::endl;
    }

    auto rotations_map = actions_map→second.find(rotation);
    if (rotations_map ≡ actions_map→second.end()) {
        std::cerr << "Drawing failed at finding valid rotation" << std::endl;
    }

    if (unitIsRobot(unitType)){
        cr→scale(1.5, 1.5);
        xGraphicCoordinate = xGraphicCoordinate/1.5;
        yGraphicCoordinate = yGraphicCoordinate/1.5;
    }

    auto next = rotations_map→second.at(unitCounter);
    /* perform actual drawing */
    Gdk::Cairo::set_source_pixbuf(cr, next,
                                  xGraphicCoordinate, yGraphicCoordinate);

    cr→rectangle(xGraphicCoordinate, yGraphicCoordinate, next→get_width(),
                 next→get_height());
    cr→fill();
    cr→restore();
}

void
GameArea::processUnitToDrawEnums(UnitsEnum &unitType, ActionsEnum &actionType,
                                 RotationsEnum &rotation) const {
    if (unitType ≡ UnitsEnum::HEAVY_TANK
        or unitType ≡ UnitsEnum::LIGHT_TANK
        or unitType ≡ UnitsEnum::MEDIUM_TANK
        or unitType ≡ UnitsEnum::MML) {
        actionType = ActionsEnum::STAND;
        /* same assets are used for given rotations;
         * e.g.: 135 and 315 are drawn with same img */
        if (rotation ≡ RotationsEnum::r135) {
```

```
            rotation = RotationsEnum::r315;
        } else if (rotation ≡ RotationsEnum::r180) {
            rotation = RotationsEnum::r000;
        } else if (rotation ≡ RotationsEnum::r225) {
            rotation = RotationsEnum::r045;
        } else if (rotation ≡ RotationsEnum::r270) {
            rotation = RotationsEnum::r090;
        }
    } else if ((unitType ≡ UnitsEnum::GRUNT
                or unitType ≡ UnitsEnum::LASER
                or unitType ≡ UnitsEnum::PSYCHO
                or unitType ≡ UnitsEnum::PYRO
                or unitType ≡ UnitsEnum::SNIPER
                or unitType ≡ UnitsEnum::TOUGH)
               and(actionType ≡ ActionsEnum::MOVE
                    or actionType ≡ ActionsEnum::STAND)) {
        /* because same imgs are used to draw all different types of robots
         * when these are moving or standing still, if this is the case, we
         * set the unit type to generic robot */
        unitType = UnitsEnum::GENERIC_ROBOT;
    }
    if (unitType ≡ UnitsEnum::JEEP){
        actionType = ActionsEnum ::STAND;
    }
}

void GameArea::drawBuildingsInView(const Cairo::RefPtr<Cairo::Context> &cr) {
    /* pointers (Unit*) are not used here because we are working with a shared
     * resource. This way, we copy the units we want to draw in a protected way,
     * and then we can draw without blocking other code. */
    std::vector<Building> buildingsToDraw =
            buildingsMonitor→getBuildingsToDraw(
                    camera.getPosition().first − (NUMBER_OF_TILES_TO_SHOW *
                                                  TILESIZE) / 2,
                    camera.getPosition().first + (NUMBER_OF_TILES_TO_SHOW *
                                                  TILESIZE) / 2,
                    camera.getPosition().second − (NUMBER_OF_TILES_TO_SHOW *
                                                   TILESIZE) / 2,
                    camera.getPosition().second + (NUMBER_OF_TILES_TO_SHOW *
                                                   TILESIZE) / 2);

    for (auto &building : buildingsToDraw) {
        /* call actual drawing method */
        drawBuilding(building.getBuildingType(), buildingsCounter.getCounter(),
                     building.getTeam(), cr,
                     cameraToRealMapX(camera.idealMapToCameraXCoordinate
                             (building.getXCoordinate())),
                     cameraToRealMapY(camera.idealMapToCameraYCoordinate
                             (building.getYCoordinate())));
    }
}

void GameArea::drawBuilding(BuildingsEnum buildingType, unsigned short counter,
                            TeamEnum team,
                            const Cairo::RefPtr<Cairo::Context> &cr,
                            unsigned int xGraphicCoordinate,
                            unsigned int yGraphicCoordinate) {
    cr→save();
    Glib::RefPtr<Gdk::Pixbuf> next = buildings.at(buildingType).at(counter);

    /* perform actual drawing */
    Gdk::Cairo::set_source_pixbuf(cr, next,
                                  xGraphicCoordinate, yGraphicCoordinate);

    cr→rectangle(xGraphicCoordinate, yGraphicCoordinate, next→get_width(),
                 next→get_height());

    cr→fill();
    cr→restore();
    /* draw flag */
    if (buildingType ≠ BuildingsEnum::FORT_DESTROYED or
            buildingType ≠ BuildingsEnum::FORT) {
        /* forts' flags are territories' flags, so we dont draw them with the
         * building, but with the other terrritories' flags instead. */
        //todo check flag's optimal position
        drawFlag(team, cr, xGraphicCoordinate, yGraphicCoordinate);
    }
}

void GameArea::drawFlag(const TeamEnum &team,
                        const Cairo::RefPtr<Cairo::Context> &cr,
                        unsigned int xGraphicCoordinate,
                        unsigned int yGraphicCoordinate) const {
    cr→save();
    Gdk::Cairo::set_source_pixbuf(cr,
                                  flags.at(team).at(flagCounter.getCounter()),
                                  xGraphicCoordinate, yGraphicCoordinate);

    cr→rectangle(xGraphicCoordinate,
                 yGraphicCoordinate,
                 flags.at(team).at(flagCounter.getCounter())→get_width(),
                 flags.at(team).at(flagCounter.getCounter())→get_height());
    cr→fill();
    cr→restore();
}

void GameArea::drawTerritoriesFlagsInView(const Cairo::RefPtr<Cairo::Context> &c
r) {
    std::map<int, std::pair<TeamEnum, std::pair<unsigned int, unsigned int>>>
    flagsToDraw = mapMonitor→getFlags();

    for (auto &flag : flagsToDraw) {
        /* call actual drawing method */
        drawFlag(flag.second.first, cr,
                 cameraToRealMapX(camera.idealMapToCameraXCoordinate(
                         flag.second.second.first)),
                 cameraToRealMapY(camera.idealMapToCameraYCoordinate(
                         flag.second.second.second)));
    }
}

void GameArea::setResources(UnitsMonitor *unitsMonitor,
                            BuildingsMonitor *buildingsMonitor,
                            MapMonitor *mapMonitor, std::string owner) {
    this→unitsMonitor = unitsMonitor;
    this→buildingsMonitor = buildingsMonitor;
    this→mapMonitor = mapMonitor;
    this→owner = owner;
}

bool GameArea::on_key_press_event(GdkEventKey *event) {
    if (event→keyval ≡ GDK_KEY_Up and event→keyval ≡ GDK_KEY_Left) {
        camera.moveUp();
        camera.moveLeft();
        //returning true, cancels the propagation of the event
        return true;
    } else if (event→keyval ≡ GDK_KEY_Down) {
        camera.moveDown();
        //returning true, cancels the propagation of the event
        return true;
    } else if (event→keyval ≡ GDK_KEY_Left) {
        camera.moveLeft();
```

```
        //returning true, cancels the propagation of the event
        return true;
    } else if (event→keyval ≡ GDK_KEY_Right) {
        camera.moveRight();
        //returning true, cancels the propagation of the event
        return true;
    } else if (event→keyval ≡ GDK_KEY_Up) {
        camera.moveUp();
        //returning true, cancels the propagation of the event
        return true;
    }
//  todo ver si el event handling se pasa arriba o no
    //if the event has not been handled, call the base class
    return Gtk::DrawingArea::on_key_press_event(event);
}

bool GameArea::on_button_press_event(GdkEventButton *event) {
    /** From https://developer.gnome.org/gdk3/stable/gdk3-Event
     *                              -Structures.html#GdkEventButton
     *
     * GdkEventType type;   the type of the event (GDK_BUTTON_PRESS,
     *                      GDK_2BUTTON_PRESS, GDK_3BUTTON_PRESS or
     *                      GDK_BUTTON_RELEASE).
     *
     * GdkWindow *window;   the window which received the event.
     *
     * gint8 send_event;    TRUE if the event was sent explicitly.
     *
     * guint32 time;        the time of the event in milliseconds.
     *
     * gdouble x;           the x coordinate of the pointer relative to the
     *                      window.
     *
     * gdouble y;           the y coordinate of the pointer relative to the
     *                      window.
     *
     * gdouble *axes;       x , y translated to the axes of device , or NULL
     *                      if device is the mouse.
     *
     * guint state;         a bit-mask representing the state of the modifier
     *                      keys (e.g. Control, Shift and Alt) and the pointer
     *                      buttons. See GdkModifierType.
     *
     * guint button;        the button which was pressed or released, numbered
     *                      from 1 to 5. Normally button 1 is the left mouse
     *                      button, 2 is the middle button, and 3 is the right
     *                      button. On 2-button mice, the middle button can
     *                      often be simulated by pressing both
     *                      mouse buttons together.
     *
     * GdkDevice *device;   the master device that the event originated from.
     *                      Use gdk_event_get_source_device()
     *                      to get the slave device.
     *
     * gdouble x_root;      the x coordinate of the pointer relative to
     *                      the root of the screen.
     *
     * gdouble y_root;      the y coordinate of the pointer relative to
     *                      the root of the screen.
     *
     */
    if (event→button ≡ 1 ∨ event→button ≡ 3) {
        unitsMonitor→wipeSelected();
        buildingsMonitor→wipe_selected();
        unitsSelected = false;
        buildingSelected = false;
        xStartCoordinate = event→x;
        yStartCoordinate = event→y;
        /* returning true, cancels the propagation of the event */
    }
    return true;
}

bool GameArea::on_button_release_event(GdkEventButton *event) {
    if (event→button ≡ 1 ∨ event→button ≡ 3) {
        xFinishCoordinate = event→x;
        yFinishCoordinate = event→y;
        makeSelection();
        coords = {camera.cameraToMapXCoordinate(screenMapToCameraX(event→x)),
                  camera.cameraToMapYCoordinate(screenMapToCameraY(event→y))};
        /* returning true, cancels the propagation of the event. We return
         * false, so the event can be handled by the game window
         * */
    }
    return false;
}

void GameArea::makeSelection() {
    /* tell each of the structures storing objects in the map to mark as
     * selected the items which are within the mouse selection */
    //todo filter out other players' units.
    unitsMonitor→markAsSelectedInRange(unitsSelected,
                                        camera.cameraToMapXCoordinate(
                                                screenMapToCameraX(
                                                        xStartCoordinate)),
                                        camera.cameraToMapYCoordinate(
                                                screenMapToCameraY(
                                                        yStartCoordinate)),
                                        camera.cameraToMapXCoordinate(
                                                screenMapToCameraX(
                                                        xFinishCoordinate)),
                                        camera.cameraToMapYCoordinate(
                                                screenMapToCameraY
                                                        (yFinishCoordinate)));
    if (¬unitsSelected) {
        buildingsMonitor→markAsSelectedInRange(
                buildingSelected,
                camera.cameraToMapXCoordinate(
                        screenMapToCameraX(xStartCoordinate)),
                camera.cameraToMapYCoordinate(
                        screenMapToCameraY(yStartCoordinate)),
                camera.cameraToMapXCoordinate(
                        screenMapToCameraX(xFinishCoordinate)),
                camera.cameraToMapYCoordinate(
                        screenMapToCameraY(yFinishCoordinate)));
    } else {
        mapMonitor→markAsSelectedInRange(
                terrainSelected,
                camera.cameraToMapXCoordinate(
                        screenMapToCameraX(xStartCoordinate)),
                camera.cameraToMapYCoordinate(
                        screenMapToCameraY(yStartCoordinate)),
                camera.cameraToMapXCoordinate(
                        screenMapToCameraX(xFinishCoordinate)),
                camera.cameraToMapYCoordinate(
                        screenMapToCameraY(yFinishCoordinate)));
    }
    selectionMade = false;
}


void GameArea::processClick() {
```

```
    //todo complete method processClick
    if (unitsSelected) {
    }
}

void GameArea::loadUnitsResources() {
    /* load fire animations */
    loadGruntFireAnimations();
    loadLaserFireAnimations();
    loadPsychoFireAnimations();
    loadPyroFireAnimations();
    loadSniperFireAnimations();
    loadToughFireAnimations();

    /* load walking animations */
    loadBlueWalkingAnimations();
    loadGreenWalkingAnimations();
    loadRedWalkingAnimations();
    loadYellowWalkingAnimations();

    /* load standing animations */
    loadBlueStandingAnimations();
    loadGreenStandingAnimations();
    loadRedStandingAnimations();
    loadYellowStandingAnimations();

    /* load vehicles' animations */
    loadTiresAnimations();
    loadNeuterVehiclesAnimations();
    loadBlueVehiclesAnimations();
    loadGreenVehiclesAnimations();
    loadRedVehiclesAnimations();
    loadYellowVehiclesAnimations();
}

unsigned short GameArea::getCounter(Unit &unit) const {
    if (unit.getType() ≡ UnitsEnum::JEEP) {
        if (unit.getTeam() ≡ TeamEnum::NEUTRAL) {
            return 0;
        }
        return jeepCounter.getCounter();
    } else if (unit.getType() ≡ UnitsEnum::LIGHT_TANK or
               unit.getType() ≡ UnitsEnum::MEDIUM_TANK or
               unit.getType() ≡ UnitsEnum::HEAVY_TANK) {
        return tankCounter.getCounter();
    } else if (unit.getType() ≡ UnitsEnum::MML){
        return mmlCounter.getCounter();
    } else if (unit.getAction() ≡ ActionsEnum::FIRE) {
        return shootingRobotCounter.getCounter();
    } else if (unit.getAction() ≡ ActionsEnum::MOVE) {
        return walkingRobotCounter.getCounter();
    } else {
        return standingRobotCounter.getCounter();
    }
}

void GameArea::updateCounters() {
    /* update units counters */
    flagCounter.updateCounter();

    shootingRobotCounter.updateCounter();

    walkingRobotCounter.updateCounter();

    standingRobotCounter.updateCounter();

    jeepCounter.updateCounter();

    tireCounter.updateCounter();

    tankCounter.updateCounter();

    mmlCounter.updateCounter();
    /* end update counter section */
}

void GameArea::initializeCounters() {
    /* one of the vectors of each category is accessed to get the size of the
     * vectors of all the category. This is possible because all vectors of
     * the same category share the same size */
    flagCounter.initialize(flags.at(TeamEnum::BLUE).size());

    jeepCounter.initialize(unitsAnimations.operator[](TeamEnum::BLUE)
                           [UnitsEnum::JEEP][ActionsEnum::STAND]
                           [RotationsEnum::r000].size());

    tireCounter.initialize(jeepTires.at(RotationsEnum::r000).size());

    standingRobotCounter.initialize(unitsAnimations.operator[](TeamEnum::BLUE)
                                    [UnitsEnum::GENERIC_ROBOT]
                                    [ActionsEnum::STAND]
                                    [RotationsEnum::r000].size());

    walkingRobotCounter.initialize(unitsAnimations.operator[](TeamEnum::BLUE)
                                   [UnitsEnum::GENERIC_ROBOT][ActionsEnum::MOVE]
                                   [RotationsEnum::r000].size());

    shootingRobotCounter.initialize(unitsAnimations.operator[](TeamEnum::BLUE)
                                    [UnitsEnum::PSYCHO][ActionsEnum::FIRE]
                                    [RotationsEnum::r000].size());

    tankCounter.initialize(unitsAnimations.operator[](TeamEnum::BLUE)
                           [UnitsEnum::LIGHT_TANK][ActionsEnum::STAND]
                           [RotationsEnum::r000].size());

    mmlCounter.initialize(unitsAnimations.operator[](TeamEnum::BLUE)
                          [UnitsEnum::MML][ActionsEnum::STAND
                          ][RotationsEnum::r000].size());

    buildingsCounter.initialize(buildings.at(BuildingsEnum::FORT).size());
}

unsigned int GameArea::cameraToRealMapX(unsigned int coordinate) {
    return get_width() * coordinate / (NUMBER_OF_TILES_TO_SHOW * TILESIZE);
}

unsigned int GameArea::cameraToRealMapY(unsigned int coordinate) {
    return get_height() * coordinate / (NUMBER_OF_TILES_TO_SHOW * TILESIZE);
}

std::pair<int, int> GameArea::get_coords() {
    return coords;
}

unsigned int GameArea::screenMapToCameraX(gdouble coordinate) {
    return (NUMBER_OF_TILES_TO_SHOW * TILESIZE * coordinate) / (get_width());
}

unsigned int GameArea::screenMapToCameraY(gdouble coordinate) {
    return (NUMBER_OF_TILES_TO_SHOW * TILESIZE * coordinate) / (get_height());
}

```

```
void GameArea::setMapData() {
    this→camera.setMapWidth(mapMonitor→getXSize());
    this→camera.setMapHeight(mapMonitor→getYSize());
}

bool GameArea::unitIsRobot(UnitsEnum unitType) {
    return (unitType ≡ UnitsEnum::GENERIC_ROBOT or
            unitType ≡ UnitsEnum::GRUNT or unitType ≡ UnitsEnum::LASER or
            unitType ≡ UnitsEnum::PSYCHO or unitType ≡ UnitsEnum::PYRO or
            unitType ≡ UnitsEnum::SNIPER or unitType ≡ UnitsEnum::TOUGH);
}

bool GameArea::unit_selected() {
    return unitsSelected;
}

bool GameArea::buildings_selected() {
    return buildingSelected;
}
```

```
#ifndef Z_TPGRUPAL_UNITSENUM_H
#define Z_TPGRUPAL_UNITSENUM_H

enum class UnitsEnum {
    /* robots */
    /* GENERIC_ROBOT is used to draw walking and standing robots, whose
     * animations don’t differ */
            GRUNT, PSYCHO, TOUGH, PYRO, SNIPER, LASER, GENERIC_ROBOT,
    /* vehicles */
            JEEP, MEDIUM_TANK, LIGHT_TANK, HEAVY_TANK, MML
};

#endif //Z_TPGRUPAL_UNITSENUM_H
```

```
#ifndef Z_TPGRUPAL_TEAMENUM_H
#define Z_TPGRUPAL_TEAMENUM_H

enum class TeamEnum {
    NEUTRAL, BLUE, GREEN, RED, YELLOW,
};

#endif //Z_TPGRUPAL_TEAMENUM_H
```

```
#ifndef Z_TPGRUPAL_ROTATIONSENUM_H
#define Z_TPGRUPAL_ROTATIONSENUM_H

enum class RotationsEnum {
    /* the numbers indicate the angle in which the unit is facing.
     * 0:   looking to the right
     * 45:  looking to upper right corner
     * 90:  looking up
     * 135: looking to upper left corner
     * 180: looking to the left
     * 225: looking to bottom left corner
     * 270: looking down
     * 315: looking to bottom right corner
     * */

            r000, r045, r090, r135, r180, r225, r270, r315
};

#endif //Z_TPGRUPAL_ROTATIONSENUM_H
```

```
#ifndef Z_TPGRUPAL_NATUREENUM_H
#define Z_TPGRUPAL_NATUREENUM_H

enum class NatureEnum {
    ROCK
};

#endif //Z_TPGRUPAL_NATUREENUM_H
```

```
#ifndef Z_TPGRUPAL_BUILDINGSENUM_H
#define Z_TPGRUPAL_BUILDINGSENUM_H

enum class BuildingsEnum {
    FORT,
    FORT_DESTROYED,
    VEHICLE_FABRIC,
    VEHICLE_FABRIC_DESTROYED,
    ROBOT_FABRIC,
    ROBOT_FABRI_DESTROYED
};

#endif //Z_TPGRUPAL_BUILDINGSENUM_H
```

```
#ifndef Z_TPGRUPAL_ACTIONSENUM_H
#define Z_TPGRUPAL_ACTIONSENUM_H

enum class ActionsEnum {
    /* REFACTOR NOTE: WALK -> MOVE*/
            FIRE, MOVE, STAND
};

#endif //Z_TPGRUPAL_ACTIONSENUM_H
```

```cpp
#ifndef Z_TPGRUPAL_COUNTER_H
#define Z_TPGRUPAL_COUNTER_H


class Counter {
    unsigned short counter;
    unsigned short maxSize;
public:
    void initialize(unsigned long maxSize);

    void updateCounter();

    unsigned short getCounter() const;
};


#endif //Z_TPGRUPAL_COUNTER_H
```

```cpp
#include "Counter.h"

void Counter::initialize(unsigned long maxSize) {
    this→maxSize = maxSize;
}

void Counter::updateCounter() {
    counter ≡ maxSize − 1 ? (counter = 0) : (counter++);
}

unsigned short Counter::getCounter() const {
    return counter;
}
```

```
#ifndef Z_TPGRUPAL_WINNER_H
#define Z_TPGRUPAL_WINNER_H


#include "Command.h"
#include "../windows/GameWindow.h"
#include "../MapMonitor.h"

class Winner : public Command {
    MapMonitor& map;
    GameWindow& window;

public:
    Winner(MapMonitor& map, GameWindow& window);
    void execute(const std::vector<std::string> &args);

};


#endif //Z_TPGRUPAL_WINNER_H
```

```
#include "Winner.h"

Winner::Winner(MapMonitor& map, GameWindow &window) :
        map(map), window(window) {
}

void Winner::execute(const std::vector<std::string> &args) {
    map.finish_winner();
    window.hide();
}
```

jun 27, 17 14:46 **UpdateUnit.h** Page 1/1

```cpp
#ifndef Z_TPGRUPAL_UPDATE_H
#define Z_TPGRUPAL_UPDATE_H


#include "Command.h"
#include "../UnitsMonitor.h"
#include "../MapMonitor.h"
#include "../ServerMessenger.h"

class UpdateUnit : public Command {
    UnitsMonitor& units;
public:
    explicit UpdateUnit(UnitsMonitor &units);

    void execute(const std::vector<std::string> &args);
};


#endif //Z_TPGRUPAL_UPDATE_H
```

jun 27, 17 14:46 **UpdateUnit.cpp** Page 1/1

```cpp
#include <iostream>
#include "UpdateUnit.h"

#define ID 0
#define STATE 1
#define X 2
#define Y 3
#define HP 4

const std::map<std::string, ActionsEnum> states = {
        {std::string("atk"), ActionsEnum::FIRE},
        {std::string("mv"), ActionsEnum::MOVE},
        {std::string("std"), ActionsEnum::STAND}
};

UpdateUnit::UpdateUnit(UnitsMonitor &units) :
        units(units)
{
}

void UpdateUnit::execute(const std::vector<std::string> &args) {
    int id = std::stoi(args[ID]);
    int x = std::stoi(args[X]);
    int y = std::stoi(args[Y]);
    ActionsEnum state = states.find(args[STATE])→second;
    unsigned int hp = std::stoul(args[HP]);
    units.update_position(id, state, x, y);
    units.update_health(id, hp);
}
```

```
#ifndef Z_TPGRUPAL_UPDATETERRITORY_H
#define Z_TPGRUPAL_UPDATETERRITORY_H


#include "Command.h"
#include "../MapMonitor.h"

class UpdateTerritory : public Command{
    MapMonitor& mapMonitor;
public:
    UpdateTerritory(MapMonitor& mapMonitor1);
    void execute(const std::vector<std::string> &args);

};


#endif //Z_TPGRUPAL_UPDATETERRITORY_H
```

```
#include <map>
#include "UpdateTerritory.h"
#include "../enums/TeamEnum.h"

#define ID 0
#define TEAM 1
#define X 2
#define Y 3

const std::map<std::string, TeamEnum> teams = {
        {std::string("blue"), TeamEnum::BLUE},
        {std::string("green"), TeamEnum::GREEN},
        {std::string("neutral"), TeamEnum::NEUTRAL},
        {std::string("red"), TeamEnum::RED},
        {std::string("yellow"), TeamEnum::YELLOW},
};

UpdateTerritory::UpdateTerritory(MapMonitor &mapMonitor1)
        : mapMonitor(mapMonitor1){
}

void UpdateTerritory::execute(const std::vector<std::string> &args) {
    int id = std::stoi(args[ID]);
    TeamEnum team = (TeamEnum) mapMonitor.get_player_id(args[TEAM]);
    teams.find(args[TEAM])->second;
    int x = std::stoi(args[X]);
    int y = std::stoi(args[Y]);

    mapMonitor.update_territory(id, team, x, y);

//    updateterritory-[id]-[team]-[x]-[y]
}
```

```
#ifndef Z_TPGRUPAL_UPDATEPOSITION_H
#define Z_TPGRUPAL_UPDATEPOSITION_H

#include <string>
#include <vector>
#include "Command.h"
#include "../UnitsMonitor.h"
#include "../MapMonitor.h"
#include "../ServerMessenger.h"

class UpdatePosition : public Command {
    UnitsMonitor &units;
public:
    explicit UpdatePosition(UnitsMonitor &units);

    void execute(const std::vector<std::string> &args);
};

#endif //Z_TPGRUPAL_UPDATEPOSITION_H
```

```
#include "UpdatePosition.h"
#include "../MapMonitor.h"
#include "../ServerMessenger.h"

#define UNIT_ID 0
#define POS_X 1
#define POS_Y 2

UpdatePosition::UpdatePosition(UnitsMonitor &monitor) :
        units(monitor)
{
}

void UpdatePosition::execute(const std::vector<std::string> &args) {
    int id = std::stoi(args[UNIT_ID]);
    int x = std::stoi(args[POS_X]);
    int y = std::stoi(args[POS_Y]);
    units.update_position(id, x, y);
}
```

```cpp
#ifndef Z_TPGRUPAL_UPDATEFACTORY_H
#define Z_TPGRUPAL_UPDATEFACTORY_H


#include "Command.h"
#include "../BuildingsMonitor.h"
#include "../MapMonitor.h"

class UpdateFactory : public Command {
    BuildingsMonitor& buildings;
    MapMonitor& map;
public:
    UpdateFactory(BuildingsMonitor& buildings, MapMonitor& map);
    void execute(const std::vector<std::string> &args);

};


#endif //Z_TPGRUPAL_UPDATEFACTORY_H
```

```cpp
#include "UpdateFactory.h"

#define ID 0
#define MINUTES 1
#define SECONDS 2
#define HP 3
#define TEAM 4

UpdateFactory::UpdateFactory(BuildingsMonitor &buildings, MapMonitor& map) :
        buildings(buildings),
        map(map)
{
}

void UpdateFactory::execute(const std::vector<std::string> &args) {
    int id = std::stoi(args[ID]);
    int min = std::stoi(args[MINUTES]);
    int sec = std::stoi(args[SECONDS]);
    int hp = std::stoi(args[HP]);
    std::string team = args[TEAM];

    TeamEnum team_enum =  (TeamEnum) map.get_player_id(team);
    buildings.update_building(id, min, sec, hp, team_enum);
}

```

```
#ifndef Z_TPGRUPAL_STARTGAME_H
#define Z_TPGRUPAL_STARTGAME_H


#include "Command.h"
#include "../windows/GameWindow.h"
#include "../ServerMessenger.h"
#include "../windows/LobbyWindow.h"

class StartGame : public Command {
    ServerMessenger &m;
    LobbyWindow& lobby;
    GameWindow& window;

public:

    StartGame(ServerMessenger& m, LobbyWindow& lobby, GameWindow& window);
    void execute(const std::vector<std::string> &args);

};


#endif //Z_TPGRUPAL_STARTGAME_H
```

```
#include "StartGame.h"

StartGame::StartGame(ServerMessenger &m,
                     LobbyWindow& lobby,
                     GameWindow &window) :
        m(m),
        window(window),
        lobby(lobby) {
}

void StartGame::execute(const std::vector<std::string> &args) {
    lobby.start_game();
}
```

```cpp
#ifndef Z_TPGRUPAL_REMOVEUNIT_H
#define Z_TPGRUPAL_REMOVEUNIT_H

#include <string>
#include <vector>
#include "Command.h"
#include "../UnitsMonitor.h"

class RemoveUnit : public Command {
    UnitsMonitor &monitor;
public:
    explicit RemoveUnit(UnitsMonitor &monitor);

    void execute(const std::vector<std::string> &args);
};


#endif //Z_TPGRUPAL_REMOVEUNIT_H
```

```cpp
#include "RemoveUnit.h"

#define UNIT_ID 0

RemoveUnit::RemoveUnit(UnitsMonitor &monitor) : monitor(monitor) {
}

void RemoveUnit::execute(const std::vector<std::string> &args) {
    monitor.removeUnit(std::stoi(args[UNIT_ID]));
}
```

```
#ifndef Z_TPGRUPAL_PLAYERNAMES_H
#define Z_TPGRUPAL_PLAYERNAMES_H


#include "Command.h"
#include "../windows/LobbyWindow.h"

class PlayerNames : public Command {
    LobbyWindow& lobby;
public:
    PlayerNames(LobbyWindow& lobby);
    void execute(const std::vector<std::string> &args);

};


#endif //Z_TPGRUPAL_PLAYERNAMES_H
```

```
#include "PlayerNames.h"


PlayerNames::PlayerNames(LobbyWindow &lobby) : lobby(lobby) {
}

void PlayerNames::execute(const std::vector<std::string> &args) {
    for (int i = 0; i < args.size(); ++i) {
        lobby.update_player_name(i, args[i]);
    }
}
```

```
#ifndef Z_TPGRUPAL_MAPSINFO_H
#define Z_TPGRUPAL_MAPSINFO_H


#include "Command.h"
#include "../windows/LobbyWindow.h"

class MapsInfo : public Command {
    LobbyWindow& lobby;
public:
    MapsInfo(LobbyWindow& lobby);
    void execute(const std::vector<std::string> &args);

};


#endif //Z_TPGRUPAL_MAPSINFO_H
```

```
#include "MapsInfo.h"

MapsInfo::MapsInfo(LobbyWindow &lobby) : lobby(lobby) {
}

void MapsInfo::execute(const std::vector<std::string> &args) {
    std::stringstream s;
    for (const std::string& map : args) {
        s << map << std::endl;
    }
    lobby.update_maps(s.str());
}
```

jun 27, 17 14:46 **Loser.h** Page 1/1

```
#ifndef Z_TPGRUPAL_LOSER_H
#define Z_TPGRUPAL_LOSER_H


#include "Command.h"
#include "../windows/GameWindow.h"

class Loser : public Command {
    MapMonitor& map;
    GameWindow& window;
public:
    Loser(MapMonitor& map, GameWindow &window);
    void execute(const std::vector<std::string> &args);
};


#endif //Z_TPGRUPAL_LOSER_H
```

jun 27, 17 14:46 **Loser.cpp** Page 1/1

```
#include "Loser.h"


Loser::Loser(MapMonitor& map, GameWindow &window) : map(map), window(window) {
}

void Loser::execute(const std::vector<std::string> &args) {
    map.finish_loser();
    window.hide();
}
```

```
#ifndef Z_TPGRUPAL_LOBBYINFO_H
#define Z_TPGRUPAL_LOBBYINFO_H


#include "Command.h"
#include "../windows/MenuWindow.h"

class LobbyInfo : public Command {
    MenuWindow& menu;
public:
    LobbyInfo(MenuWindow& menu);
    void execute(const std::vector<std::string> &args);

};


#endif //Z_TPGRUPAL_LOBBYINFO_H
```

```
#include "LobbyInfo.h"

LobbyInfo::LobbyInfo(MenuWindow &menu) : menu(menu) {
}

void LobbyInfo::execute(const std::vector<std::string> &args) {
    menu.update_lobbies(args);
}
```

```cpp
#ifndef Z_TPGRUPAL_LOADMAP_H
#define Z_TPGRUPAL_LOADMAP_H


#include "../MapMonitor.h"
#include "Command.h"
#include "../BuildingsMonitor.h"
#include "../windows/GameWindow.h"
#include <string>
#include <vector>

class LoadMap : public Command {
    MapMonitor &mapMonitor;
    BuildingsMonitor &buildingsMonitor;
    GameWindow &window;

public:
    LoadMap(MapMonitor &monitor, BuildingsMonitor &buildingsMonitor,
            GameWindow &window);

    void execute(const std::vector<std::string> &args);
};


#endif //Z_TPGRUPAL_LOADMAP_H
```

```cpp
#include <pugixml.hpp>
#include <iostream>
#include "LoadMap.h"


#define ROBOT_FACTORY "res/assets/buildings/robot/base_jungle.png"
#define VEHICLE_FACTORY "res/assets/buildings/vehicle/base_jungle.png"

const std::map<std::string, BuildingsEnum> buildings{
        {std::string("VehicleFactory"), BuildingsEnum::VEHICLE_FABRIC},
        {std::string("UnitFactory"),    BuildingsEnum::ROBOT_FABRIC},
        {std::string("Fort"),           BuildingsEnum::FORT}
};

LoadMap::LoadMap(MapMonitor &mapMonitor, BuildingsMonitor &buildings,
                 GameWindow &window) :
        mapMonitor(mapMonitor), buildingsMonitor(buildings), window(window) {
}

void LoadMap::execute(const std::vector<std::string> &args) {
    /* initialize map so then can be completed with read data */
    mapMonitor.initializeMap(100, 100);

    pugi::xml_document doc;
    /* the only arg we receive is the map,
     * which is the whole xml saved in a string */
    pugi::xml_parse_result result = doc.load_string(args[0].c_str());
    if (¬result) {
        /* FATAL ERROR LOADING MAP */
        return;
    }

    std::vector<std::vector<std::string>> map;
    pugi::xml_node root = doc.child("Map");
    pugi::xml_node terrain_node = root.child("Terrain");
    for (auto node_row : terrain_node.children()) {
        unsigned int coord_x = 0;
        for (auto cell : node_row.children()) {
            if (map.size() ≤ coord_x) {
                map.push_back(std::vector<std::string>());
            }
            std::string terrain = cell.attribute("terrain").value();

            map.at(coord_x++).push_back(terrain);
        }
    }

    unsigned long size = map.size();
    mapMonitor.initializeMap(size, size);
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            mapMonitor.setCell(i, j, map[i][j]);
        }
    }
    window.setMapData();
}
```

```
#ifndef Z_TPGRUPAL_JOINLOBBY_H
#define Z_TPGRUPAL_JOINLOBBY_H


#include "Command.h"
#include "../windows/MenuWindow.h"
#include "../windows/LobbyWindow.h"

class JoinLobby : public Command {
    MenuWindow& menu;
    LobbyWindow& lobby;
    ServerMessenger& messenger;
public:
    JoinLobby(MenuWindow& menu, LobbyWindow& lobby, ServerMessenger& messenger);
    void execute(const std::vector<std::string> &args);
};


#endif //Z_TPGRUPAL_JOINLOBBY_H
```

```
#include "JoinLobby.h"

#define STATUS 0

#define OK "ok"
#define ERROR "error"
JoinLobby::JoinLobby(MenuWindow &menu, LobbyWindow &lobby,
                     ServerMessenger& messenger) :
        menu(menu),
        lobby(lobby),
        messenger(messenger)
{
}

void JoinLobby::execute(const std::vector<std::string> &args) {
    if (args[STATUS] ≡ OK) {
        menu.join_lobby();

        // Fetch available maps
        messenger.send("mapsinfo");
    }
}
```

```
#ifndef Z_TPGRUPAL_FACTORYSTATS_H
#define Z_TPGRUPAL_FACTORYSTATS_H


#include "Command.h"
#include "../BuildingsMonitor.h"
#include "../windows/GameWindow.h"

class FactoryStats : public Command {
    GameWindow& window;
public:
    FactoryStats(GameWindow& window);
    void execute(const std::vector<std::string> &args);

};


#endif //Z_TPGRUPAL_FACTORYSTATS_H
```

```
#include "FactoryStats.h"
#include "../windows/GameWindow.h"

FactoryStats::FactoryStats(GameWindow& window) : window(window) {
}

#define TYPE 0
#define FIRE_RATE 1
#define MINUTES 2
#define SECONDS 3
#define HP 4

void FactoryStats::execute(const std::vector<std::string> &args) {
    std::string type = args[TYPE];
    int fire_rate = std::stoi(args[FIRE_RATE]);
    int hp = std::stoi(args[HP]);
    int minutes = std::stoi(args[MINUTES]);
    int seconds = std::stoi(args[SECONDS]);
    window.update_factory_panel(type, fire_rate, hp);
    window.update_factory_timer(minutes, seconds);
}
```

```cpp
#ifndef Z_TPGRUPAL_FACTORYNEXTUNIT_H
#define Z_TPGRUPAL_FACTORYNEXTUNIT_H


#include "Command.h"
#include "../windows/GameWindow.h"

class FactoryNextUnit : public Command {
    GameWindow &window;
public:
    explicit FactoryNextUnit(GameWindow &window);

    void execute(const std::vector<std::string> &args);
};


#endif //Z_TPGRUPAL_FACTORYNEXTUNIT_H
```

```cpp
#include "FactoryNextUnit.h"

#define UNIT_NAME 0

FactoryNextUnit::FactoryNextUnit(GameWindow &window) : window(window) {
}

void FactoryNextUnit::execute(const std::vector<std::string> &args) {
    std::string path = "res/portraits/" + args[UNIT_NAME] + ".png";
    window.factory_change_unit(path);
}

```

```
#ifndef Z_TPGRUPAL_COMMAND_H
#define Z_TPGRUPAL_COMMAND_H

#include <vector>
#include <string>

class Command {
public:
    virtual void execute(const std::vector<std::string> &args) = 0;
};

#endif //Z_TPGRUPAL_COMMAND_H
```

```
#ifndef Z_TPGRUPAL_ADDUNIT_H
#define Z_TPGRUPAL_ADDUNIT_H

#include <vector>
#include <string>
#include "Command.h"
#include "../UnitsMonitor.h"
#include "../MapMonitor.h"

class AddUnit : public Command {
    UnitsMonitor &unitsMonitor;
    MapMonitor& map;
public:
    explicit AddUnit(UnitsMonitor &unitsMonitor,
                     MapMonitor& map);

    void execute(const std::vector<std::string> &args);
};


#endif //Z_TPGRUPAL_ADDUNIT_H
```

```cpp
#include <iostream>
#include <vector>
#include "AddUnit.h"

#define UNIT_ID 0
#define X 1
#define Y 2
#define UNIT_NAME 3
#define TEAM 4
#define HP 5

const std::map<std::string, UnitsEnum> units = {
        {std::string("grunt"), UnitsEnum::GRUNT},
        {std::string("psycho"), UnitsEnum::PSYCHO},
        {std::string("tough"), UnitsEnum::TOUGH},
        {std::string("pyro"), UnitsEnum::PYRO},
        {std::string("sniper"), UnitsEnum::SNIPER},
        {std::string("laser"), UnitsEnum::LASER},
        {std::string("generic_robot"), UnitsEnum::GENERIC_ROBOT},
        {std::string("jeep"), UnitsEnum::JEEP},
        {std::string("medium_tank"), UnitsEnum::MEDIUM_TANK},
        {std::string("light_tank"), UnitsEnum::LIGHT_TANK},
        {std::string("heavy_tank"), UnitsEnum::HEAVY_TANK},
        {std::string("mml"), UnitsEnum::MML}
};

void AddUnit::execute(const std::vector<std::string> &args) {
    int x = std::stoi(args[X]);
    int y = std::stoi(args[Y]);
    int id = std::stoi(args[UNIT_ID]);

    std::string name = args[UNIT_NAME];
    auto type = units.find(name);
    if (type ≡ units.end()) {
        /* Error adding unit: received type */
        return;
    }

    std::string owner = args[TEAM];
    int team_id = map.get_player_id(owner);
    unsigned int hp = std::stoul(args[HP]);
    Unit unit(id, {x, y}, type→second, (TeamEnum) team_id, hp);
    unit.update_owner(owner);
    unit.update_unit_name(name);

    unitsMonitor.addUnit(unit);
}

AddUnit::AddUnit(UnitsMonitor &unitsMonitor,
                 MapMonitor& map)
        : unitsMonitor(unitsMonitor),
          map(map)
{
}
```

```cpp
#ifndef Z_TPGRUPAL_ADDNATURE_H
#define Z_TPGRUPAL_ADDNATURE_H


#include "Command.h"
#include "../MapMonitor.h"

class AddNature : public Command {
    MapMonitor &mapMonitor;
public:
    explicit AddNature(MapMonitor &mapMonitor);

    void execute(const std::vector<std::string> &args);
};


#endif //Z_TPGRUPAL_ADDNATURE_H
```

```cpp
#include <map>
#include "AddNature.h"

#define ID 0
#define X 1
#define Y 2
#define TYPE 3
#define TEAM 4

const std::map<std::string, NatureEnum > natureMap = {
        {std::string("Rock"), NatureEnum ::ROCK}
};


AddNature::AddNature(MapMonitor &mapMonitor) : mapMonitor(mapMonitor){
}

void AddNature::execute(const std::vector<std::string> &args) {
    int id = std::stoi(args[ID]);
    int x = std::stoi(args[X]);
    int y = std::stoi(args[Y]);
    NatureEnum natureType = natureMap.find(args[TYPE])->second;

    Nature nature(natureType, {x, y}, id);

    mapMonitor.addNature(nature);
//    Building b(BuildingsEnum::ROBOT_FABRIC, x, y, id, team);
//    buildings.addBuilding(b);
}
```

```cpp
#ifndef Z_TPGRUPAL_ADDBUILDING_H
#define Z_TPGRUPAL_ADDBUILDING_H


#include "../BuildingsMonitor.h"
#include "Command.h"
#include "../MapMonitor.h"

class AddBuilding : public Command {
    BuildingsMonitor &buildings;
    MapMonitor& map;
public:
    explicit AddBuilding(BuildingsMonitor &buildings,
                         MapMonitor& map);

    void execute(const std::vector<std::string> &args);
};


#endif //Z_TPGRUPAL_ADDBUILDING_H
```

```cpp
#include "AddBuilding.h"

#define TILESIZE 16    //tile width in pixels.
#define ID 0
#define X 1
#define Y 2
#define TYPE 3
#define TEAM 4
#define HP 5

const std::map<std::string, BuildingsEnum> buildingsMap = {
        {std::string("Fort"), BuildingsEnum::FORT},
        /* Since we dont yet distinguish between robot and vehicle
         * factories, all factories will be drawn as robot factories. */
        {std::string("Factory"), BuildingsEnum::ROBOT_FABRIC}
};

AddBuilding::AddBuilding(BuildingsMonitor &buildings,
                         MapMonitor& map)
        : buildings(buildings),
          map(map)
{
}

void AddBuilding::execute(const std::vector<std::string> &args) {
    int id = std::stoi(args[ID]);
    int x = std::stoi(args[X]);
    int y = std::stoi(args[Y]);

    BuildingsEnum building_type = buildingsMap.find(args[TYPE])→second;

    std::string owner = args[TEAM];
    int team_id = map.get_player_id(owner);

    unsigned int hp = std::stoul(args[HP]);
    Building b(building_type, x, y, id, (TeamEnum) team_id, owner, hp);
    buildings.addBuilding(b);
}
```

```cpp
#ifndef Z_TPGRUPAL_CLIENTTREAD_H
#define Z_TPGRUPAL_CLIENTTREAD_H

#include <map>
#include <string>
#include "BuildingsMonitor.h"
#include "MapMonitor.h"
#include "ServerMessenger.h"
#include "commands/Command.h"
#include "UnitsMonitor.h"
#include "windows/GameWindow.h"
#include "windows/LobbyWindow.h"
#include "GameBuilder.h"
#include <Thread.h>

class ClientThread : public Thread {
    UnitsMonitor &unitsMonitor;
    BuildingsMonitor &buildingsMonitor;
    MapMonitor &mapMonitor;
    ServerMessenger &messenger;
    GameWindow &window;
    MenuWindow& menu;
    LobbyWindow& lobby;
    bool finished = false; // Flag for finishing execution
    std::map<std::string, Command *> commands;
public:
    ClientThread(UnitsMonitor &unitsMonitor,
                 BuildingsMonitor &buildingsMonitor,
                 MapMonitor &mapMonitor, ServerMessenger &messenger,
                 GameBuilder& builder);

    virtual void run();

    ~ClientThread();

    /* Finish the thread's execution */
    void finish();
private:
    /* Loads commands */
    void initCommands();

    /** Main thread loop, receives commands from the server, parses them and
     * executes them */
    void loop();

    /* Parses a command string and executes it */
    void parse(std::string &s);


};


#endif //Z_TPGRUPAL_CLIENTTREAD_H
```

```cpp
#include <iostream>
#include <vector>
#include <pugixml.hpp>
#include <split.h>
#include <utility>
#include "ClientThread.h"
#include "commands/AddUnit.h"
#include "commands/RemoveUnit.h"
#include "commands/UpdatePosition.h"
#include "commands/LoadMap.h"
#include "commands/UpdateUnit.h"
#include "commands/FactoryNextUnit.h"
#include "commands/AddBuilding.h"
#include "commands/AddNature.h"
#include "commands/StartGame.h"
#include "commands/PlayerNames.h"
#include "commands/UpdateFactory.h"
#include "commands/FactoryStats.h"
#include "commands/LobbyInfo.h"
#include "commands/JoinLobby.h"
#include "commands/Winner.h"
#include "commands/Loser.h"
#include "commands/MapsInfo.h"
#include "commands/UpdateTerritory.h"

void ClientThread::run() {
    initCommands();
    loop();
}

ClientThread::ClientThread(UnitsMonitor &unitsMonitor,
                           BuildingsMonitor &buildingsMonitor,
                           MapMonitor &mapMonitor,
                           ServerMessenger &messenger,
                           GameBuilder& builder) :
        unitsMonitor(unitsMonitor),
        buildingsMonitor(buildingsMonitor),
        mapMonitor(mapMonitor),
        messenger(messenger),
        lobby(*builder.get_lobby_window()),
        window(*builder.get_window()),
        menu(*builder.get_menu_window())
{
}

void ClientThread::loop() {
    try {
        while (¬finished) {
            std::string msg = messenger.receive();
            std::vector<std::string> commands = utils::split(msg, '|');
            for (std::string &cmd : commands) {
                parse(cmd);
            }
        }
    } catch (SocketError &e) {
        return;
    }
}

void ClientThread::parse(std::string &s) {
    std::vector<std::string> params = utils::split(s, '-');
    int cmd = 0;
    auto result = commands.find(params[cmd]);
    if (result ≡ commands.end()) {
        /* Invalid command */
        return;
```

```cpp
    }
    std::vector<std::string> args(++params.begin(), params.end());
    result→second→execute(args);
}

void ClientThread::finish() {
    finished = true;
    messenger.kill();
}

void ClientThread::initCommands() {
    commands["loadmap"] = new LoadMap(mapMonitor, buildingsMonitor, window);
    commands["addunit"] = new AddUnit(unitsMonitor, mapMonitor);
    commands["removeunit"] = new RemoveUnit(unitsMonitor);
    commands["move"] = new UpdatePosition(unitsMonitor);
    commands["updateunit"] = new UpdateUnit(unitsMonitor);
    commands["nextunit"] = new FactoryNextUnit(window);
    commands["addbuilding"] = new AddBuilding(buildingsMonitor, mapMonitor);
    commands["addnature"] = new AddNature(mapMonitor);
    commands["startgame"] = new StartGame(messenger, lobby, window);
    commands["names"] = new PlayerNames(lobby);
    commands["updatefactory"] = new UpdateFactory(buildingsMonitor, mapMonitor);
    commands["factorystats"] = new FactoryStats(window);
    commands["lobbyinfo"] = new LobbyInfo(menu);
    commands["joinlobby"] = new JoinLobby(menu, lobby, messenger);
    commands["winner"] = new Winner(mapMonitor, window);
    commands["loseryousuck"] = new Loser(mapMonitor, window);
    commands["mapsinfo"] = new MapsInfo(lobby);
    commands["updateterritory"] = new UpdateTerritory(mapMonitor);
}

ClientThread::~ClientThread() {
    for (std::pair<std::string, Command *> c : commands) {
        delete c.second;
    }
}
```

```cpp
#ifndef Z_TPGRUPAL_CELL_H
#define Z_TPGRUPAL_CELL_H

#include <string>

class Cell {
private:
    std::string terrainType;

public:
    Cell();

    explicit Cell(std::string &terrainType);

    void assignTerrainType(std::string terrainType);

    std::string getTerrainType();
};


#endif //Z_TPGRUPAL_CELL_H
```

```cpp
#include "Cell.h"

Cell::Cell() : terrainType("") {
}

Cell::Cell(std::string &terrainType) : terrainType(terrainType) {
}

void Cell::assignTerrainType(std::string terrainType) {
    this→terrainType = terrainType;
}

std::string Cell::getTerrainType() {
    return terrainType;
}
```

```
#ifndef Z_TPGRUPAL_CAMERA_H
#define Z_TPGRUPAL_CAMERA_H


#include <utility>

class Camera {
private:
    /* save tile size in pixels so calculations on max and min coord.
     * can be done */
    unsigned int tileSize;

    unsigned int numberOfTilesToShow;

    /* position in pixels */
    std::pair<unsigned int, unsigned int> position;

    unsigned int minXCoordinate, minYCoordinate, maxXCoordinate, maxYCoordinate;

    unsigned int mapWidth, mapHeight;

public:
    /* camera will be initialized in (minx, miny) position */
    Camera(unsigned int tileSize, unsigned int mapWidth, unsigned int mapHeight,
           unsigned int numberOfTilesToShow);

    std::pair<unsigned int, unsigned int> getPosition();

    void setMapWidth(unsigned int width);

    void setMapHeight(unsigned int height);

    void moveUp();

    void moveDown();

    void moveLeft();

    void moveRight();

    std::pair<unsigned int, unsigned int> cameraOffset();

    unsigned int idealMapToCameraXCoordinate(unsigned int globalXPosition);

    unsigned int idealMapToCameraYCoordinate(unsigned int globalYPosition);

    unsigned int cameraToMapXCoordinate(unsigned int coordinate);

    unsigned int cameraToMapYCoordinate(unsigned int coordinate);
};


#endif //Z_TPGRUPAL_CAMERA_H
```

```
#include "Camera.h"
#include <iostream>

Camera::Camera(unsigned int tileSize, unsigned int mapWidth,
               unsigned int mapHeight,
               unsigned int numberOfTilesToShow) :
        tileSize(tileSize),
        numberOfTilesToShow(numberOfTilesToShow),
        minXCoordinate(numberOfTilesToShow * tileSize / 2),
        minYCoordinate(numberOfTilesToShow * tileSize / 2),
        maxXCoordinate(
                mapWidth * tileSize − numberOfTilesToShow * tileSize / 2),
        maxYCoordinate(
                mapHeight * tileSize − numberOfTilesToShow * tileSize / 2),
        position(numberOfTilesToShow * tileSize / 2,
                 numberOfTilesToShow * tileSize / 2) {
}

/**
 *
 * @return camera position is given in pixels.
 */
std::pair<unsigned int, unsigned int> Camera::getPosition() {
    return position;
}

void Camera::setMapWidth(unsigned int width) {
    mapWidth = width;
    maxXCoordinate = mapWidth * tileSize − numberOfTilesToShow * tileSize / 2;
}

void Camera::setMapHeight(unsigned int height) {
    mapHeight = height;
    maxYCoordinate = mapHeight * tileSize − numberOfTilesToShow * tileSize / 2;
}

void Camera::moveUp() {
    if (position.second ≠ minYCoordinate) {
        position.second −= tileSize;
    }
}

void Camera::moveDown() {
    if (position.second ≠ maxYCoordinate) {
        position.second += tileSize;
    }
}

void Camera::moveRight() {
    if (position.first ≠ maxXCoordinate) {
        position.first += tileSize;
    }
}

void Camera::moveLeft() {
    if (position.first ≠ minXCoordinate) {
        position.first −= tileSize;
    }
}

std::pair<unsigned int, unsigned int> Camera::cameraOffset() {
    return std::pair<unsigned int, unsigned int>
            (position.first − numberOfTilesToShow * tileSize / 2,
             position.second − numberOfTilesToShow * tileSize / 2);
}

```

```
unsigned int Camera::idealMapToCameraXCoordinate(unsigned int globalXPosition) {
    return globalXPosition − cameraOffset().first;
}

unsigned int Camera::idealMapToCameraYCoordinate(unsigned int globalYPosition) {
    return globalYPosition − cameraOffset().second;
}

unsigned int Camera::cameraToMapXCoordinate(unsigned int coordinate) {
    return coordinate + cameraOffset().first;
}

unsigned int Camera::cameraToMapYCoordinate(unsigned int coordinate) {
    return coordinate + cameraOffset().second;
}
```

```
#ifndef Z_TPGRUPAL_BUILDINGSMONITOR_H
#define Z_TPGRUPAL_BUILDINGSMONITOR_H


#include <mutex>
#include "Building.h"
#include <vector>

class BuildingsMonitor {
private:
    std::vector<Building> buildings;

    std::mutex m;
public:

    void markAsSelectedInRange(bool &buildingSelected,
                               gdouble xStartCoordinate,
                               gdouble yStartCoordinate,
                               gdouble xFinishCoordinate,
                               gdouble yFinishCoordinate);

    void addBuilding(Building &b);

    std::vector<Building>
    getBuildingsToDraw(unsigned int minX, unsigned int maxX, unsigned int minY,
                       unsigned int maxY);

    std::vector<Building> get_selected();
    void wipe_selected();
    Building get_building(int id);

    void update_building(int id, int minutes, int seconds, int hp,
                         TeamEnum team);
    void clear();
};


#endif //Z_TPGRUPAL_BUILDINGSMONITOR_H
```

```cpp
#include "BuildingsMonitor.h"
#include <Lock.h>


void BuildingsMonitor::markAsSelectedInRange(bool &buildingSelected,
                                             gdouble xStartCoordinate,
                                             gdouble yStartCoordinate,
                                             gdouble xFinishCoordinate,
                                             gdouble yFinishCoordinate) {
    Lock l(m);
    for (Building &building : buildings) {
        /* each player has to selects its units in range */
        building.markAsSelectedInRange(buildingSelected, xStartCoordinate,
                                       yStartCoordinate, xFinishCoordinate,
                                       yFinishCoordinate);
        if (buildingSelected) {
            break;
        }
    }
}

void BuildingsMonitor::addBuilding(Building &b) {
    Lock l(m);
    buildings.push_back(b);
}

std::vector<Building>
BuildingsMonitor::getBuildingsToDraw(unsigned int minX, unsigned int maxX,
                                     unsigned int minY, unsigned int maxY) {
    Lock l(m);
    std::vector<Building> returnVector;

    for (Building &building : buildings) {
        if (building.getXCoordinate() ≥ minX and
            building.getXCoordinate() ≤ maxX and
            building.getYCoordinate() ≥ minY and
            building.getYCoordinate() ≤ maxY) {
            returnVector.emplace_back(building);
        }
    }
    return returnVector;
}

std::vector<Building> BuildingsMonitor::get_selected() {
    std::vector<Building> selected_buildings;
    for (Building &building : buildings) {
        if (building.is_selected()) {
            selected_buildings.push_back(building);
        }
    }
    return selected_buildings;
}

void BuildingsMonitor::wipe_selected() {
    for (Building &b: buildings) {
        b.unselect();
    }
}

Building BuildingsMonitor::get_building(int id) {
    Lock l(m);
    for (Building& building: buildings) {
        if (building.get_ID() ≡ id) {
            return building;
        }
    }
```

```cpp
    return Building();
}

void
BuildingsMonitor::update_building(int id, int minutes, int seconds, int hp,
                                  TeamEnum team) {
    Lock l(m);
    for (auto b = buildings.begin(); b ≠ buildings.end(); ++b) {
        if (b→get_ID() ≡ id) {
            b→update_hp(hp);
            b→update_time_left(minutes, seconds);
            b→update_team(team);
        }
    }
}

void BuildingsMonitor::clear() {
    Lock l(m);
    buildings.clear();
}
```

```
#ifndef Z_TPGRUPAL_BUILDING_H
#define Z_TPGRUPAL_BUILDING_H

#include <gtkmm/drawingarea.h>
#include "enums/BuildingsEnum.h"
#include "enums/TeamEnum.h"
#include <utility>
#include <map>
#include <string>
#include <vector>

class Building {
private:
    BuildingsEnum buildingType;
    TeamEnum team;
    std::pair<unsigned int, unsigned int> position;

    /* bool selected: indicates wether the unit has been selected
     * with the mouse or not */
    bool selected;
    int id;
    std::string owner;
    int minutes;
    int seconds;
    unsigned int max_hp;
    unsigned int hp;
public:
    Building(BuildingsEnum type, int x, int y, int id, TeamEnum team,
             const std::string& owner, unsigned int hp);

    Building();
    /**
     * This methods checks all the player’s units to see if any of its units is
     * located within the area of selection. If so, the units’ attribute
     * selected is set to TRUE.
     */
    void markAsSelectedInRange(bool &buildingSelected,
                               gdouble xStartCoordinate,
                               gdouble yStartCoordinate,
                               gdouble xFinishCoordinate,
                               gdouble yFinishCoordinate);

    unsigned int getXCoordinate();

    unsigned int getYCoordinate();

    BuildingsEnum getBuildingType();

    TeamEnum getTeam();
    bool is_selected();
    void unselect();
    std::string get_owner();
    unsigned int get_hp();

    unsigned int get_max_hp();

    int get_ID();

    void update_time_left(int minutes, int seconds);

    void update_hp(unsigned int hp);

    void update_team(TeamEnum team);

    std::pair<int, int> get_time_left();

```

```
    void destroy();
};


#endif //Z_TPGRUPAL_BUILDING_H
```

```cpp
#include "Building.h"

#define LENIENCY_FORT 16
#define LENIENCY_FACTORY 8

void Building::markAsSelectedInRange(bool &buildingSelected,
                                     gdouble xStartCoordinate,
                                     gdouble yStartCoordinate,
                                     gdouble xFinishCoordinate,
                                     gdouble yFinishCoordinate) {
    int x = (int) xFinishCoordinate;
    int y = (int) yFinishCoordinate;
    int x_abs = abs(position.first − x);
    int y_abs = abs(position.second − y);

    unsigned int leniency;

    if (buildingType ≡ BuildingsEnum::FORT or
            buildingType ≡ BuildingsEnum::FORT_DESTROYED) {
        /* case building is fort... */
        leniency = LENIENCY_FORT;
    } else {
        /* case building is factory... */
        leniency = LENIENCY_FACTORY;
    }
    if (x_abs ≤ leniency ∧ y_abs ≤ leniency) {
        selected = true;
        buildingSelected = true;
    }
}

Building::Building(BuildingsEnum type, int x, int y, int id, TeamEnum team,
                   const std::string& owner, unsigned int hp) :
        buildingType(type),
        position(x, y),
        id(id),
        selected(false),
        team (team),
        owner(owner),
        hp(hp),
        max_hp(hp),
        minutes(0),
        seconds(0)

{
}

unsigned int Building::getXCoordinate() {
    return position.first;
}

unsigned int Building::getYCoordinate() {
    return position.second;
}

BuildingsEnum Building::getBuildingType() {
    return buildingType;
}

TeamEnum Building::getTeam() {
    return team;
}

bool Building::is_selected() {
    return selected;
}
```

```cpp

void Building::unselect() {
    selected = false;
}

Building::Building() {
    id = 0;
}

std::string Building::get_owner() {
    return owner;
}

unsigned int Building::get_hp() {
    return hp;
}

unsigned int Building::get_max_hp() {
    return max_hp;
}

int Building::get_ID() {
    return id;
}

void Building::update_time_left(int minutes, int seconds) {
    this→minutes = minutes;
    this→seconds = seconds;
}

void Building::update_hp(unsigned int hp) {
    this→hp = hp;
    if (hp ≤ 0) {
        destroy();
    }
}

std::pair<int, int> Building::get_time_left() {
    return {minutes, seconds};
}

void Building::destroy() {
    if (buildingType ≡ BuildingsEnum::ROBOT_FABRIC) {
        buildingType = BuildingsEnum::ROBOT_FABRI_DESTROYED;
    } else if (buildingType ≡ BuildingsEnum::VEHICLE_FABRIC) {
        buildingType = BuildingsEnum::VEHICLE_FABRIC_DESTROYED;
    } else if (buildingType ≡ BuildingsEnum::FORT) {
        buildingType = BuildingsEnum ::FORT_DESTROYED;
    }
}

void Building::update_team(TeamEnum team) {
    this→team = team;
}
```

```
#ifndef Z_TPGRUPAL_ARMAMENT_H
#define Z_TPGRUPAL_ARMAMENT_H


#include <vector>
#include <string>
#include <map>
#include <gtkmm/drawingarea.h>
#include <SDL2/SDL_mixer.h>

class Armament {
private:
    /* sound used when shot */
    Mix_Chunk *sound;

    /* animation is stored as a series of images needed
     * to draw the animation. */
    std::vector<Glib::RefPtr<Gdk::Pixbuf>> animation;
};


#endif //Z_TPGRUPAL_ARMAMENT_H
```

```
#include "Armament.h"
```

Table of Contents