File - main.cpp

```cpp
#include <iostream>
#include <algorithm>
#include "field/Field.hpp"
#include "display/FieldController.hpp"

bool isNumber(const std::string &s) {
   return !s.empty() && std::find_if(s.begin(), s.end(),
                                     [](unsigned char c) { return !std::isdigit(c); }) == s.end();
}

int main(int argc, char **argv) {
   if (argc != 5) {
      std::cerr << "Usage: " << argv[0] << " <width> <height> <nbVampire> <nbHuman>" << std::endl;
      return 1;
   }

   if (!isNumber(argv[1]) || !isNumber(argv[2]) || !isNumber(argv[3]) || !isNumber(argv[4])) {
      std::cerr << "Invalid parameters. All parameters must be numbers." << std::endl;
      return 1;
   }

   int width = std::atoi(argv[1]);
   int height = std::atoi(argv[2]);
   int nbVampire = std::atoi(argv[3]);
   int nbHuman = std::atoi(argv[4]);

   if (width <= 0 || height <= 0 || nbVampire < 0 || nbHuman < 0) {
      std::cerr << "Invalid parameters. All numbers must be non-negative,"
                   " and dimensions must be positive." << std::endl;
      return 1;
   }
   Field field{static_cast<size_t>(height),
               static_cast<size_t>(width),
               static_cast<size_t>(nbVampire),
               static_cast<size_t>(nbHuman)};
   FieldController controller{field};

   std::cout << "Simulation is running..." << std::endl;
   controller.start();
   std::cout << "Simulation is finished." << std::endl;
   return 0;
}
```

File - Field.cpp

```cpp
/**
 * @file Implementation of the Field class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#include <iostream>
#include "Field.hpp"
#include "../humanoids/Vampire.hpp"
#include "../humanoids/Human.hpp"
#include "../humanoids/Buffy.hpp"
#include "../utils/Random.hpp"

Field::Field(size_t height, size_t width, size_t nbVampire, size_t nbHuman) : height_(height), width_(width),
                                                                              nbVampire_(nbVampire), nbHuman_(nbHuman) {

   humanoids.emplace_back(std::make_shared<Buffy>(Vec2D::getRandomVector(width_ - 1, height_ - 1)));

   for (size_t i = 0; i < nbVampire; i++) {
      humanoids.emplace_back(std::make_shared<Vampire>(Vec2D::getRandomVector(width_ - 1, height_ - 1)));

   }
   for (size_t i = 0; i < nbHuman; i++) {
      humanoids.emplace_back(std::make_shared<Human>(Vec2D::getRandomVector(width_ - 1, height_ - 1)));

   }
   turn_ = 0;
}

size_t Field::getTurn() const {
   return turn_;
}

size_t Field::nextTurn() {
   // Déterminer les prochaines actions
   for (auto& humanoid: humanoids)
      humanoid->setAction(*this);
   // Executer les actions
   for (auto& humanoid: humanoids)
      humanoid->executeAction(*this);
   // Enlever les humanoides tués
   for (auto it = humanoids.begin(); it != humanoids.end();) {
      if (!(*it)->isAlive()) {
         it = humanoids.erase(it); // suppression de l'élément dans la liste
      } else
         ++it;
   }

   //std::cout << "Turn " << turn_ << " : " << nbVampire_ << " vampires, " << nbHuman_ << " humans" << std::endl;

   for (auto& it : humanoids) {
      if (it->getPosition().x < 0 || it->getPosition().x >= width_ || it->getPosition().y < 0 || it->getPosition().y >= height_) {
         std::cout << "Humanoid: " << it->getType() << " is out of bounds, pos: {" << it->getPosition().x << ", " <<
         it->getPosition().y << "}" << std::endl;
      }
   }

```

File - Field.cpp

```cpp
    return turn_++;
}

std::vector<std::vector<std::string>> Field::fieldToStringVector() const {
    std::vector<std::vector<std::string>> map(height_, std::vector<std::string>(width_, ""));
    for (const auto& humanoid: humanoids) {
        auto position = humanoid->getPosition();
        map[position.x][position.y] = humanoid->getType();
    }
    return map;
}

size_t Field::getHeight() const {
    return height_;
}

size_t Field::getWidth() const {
    return width_;
}

void Field::vampireKilled() {
    if (nbVampire_ > 0)
        nbVampire_--;
}

void Field::humanKilled() {
    if (nbHuman_ > 0)
        nbHuman_--;
}

void Field::vampireBorn() {
    nbVampire_++;
}

bool Field::aliveVampires() const {
    return nbVampire_ > 0;
}

bool Field::aliveHumans() const {
    return nbHuman_ > 0;
}

Status Field::isFinished() const {
    if (nbHuman_) {
        return nbVampire_ > 0 ? Status::RUNNING : Status::WIN;
    }
    return Status::LOSE;
}

void Field::addHumanoid(const std::shared_ptr<Humanoid>& humanoid) {
    humanoids.emplace_back(humanoid);
}

```

File - Field.hpp

```cpp
/**
 * @file Declaration of the Field class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include <list>
#include <memory>
#include <vector>
#include "../humanoids/Humanoid.hpp"
#include "../utils/Status.hpp"

/**
 * Field class to represent the game. It will have a list of all humanoids. It will also have a height and a width
 * to represent the limits of the game. It will have the number of vampire and humans as to not iterate over the lists
 * to know how many vampires and humans are left within the list.
 */
class Field {
private:
    std::list<std::shared_ptr<Humanoid>> humanoids;
    size_t height_;
    size_t width_;
    size_t nbVampire_;
    size_t nbHuman_;
    size_t turn_;
public:

    /**
     * Constructor of the class
     * @param height height of the game
     * @param width width of the game
     * @param nbVampire nbVampire at the beginning of the game
     * @param nbHuman nbHuman at the beginning of the game
     */
    Field(size_t height, size_t width, size_t nbVampire, size_t nbHuman);

    /**
     * Getter for the turn_ attribute
     * @return the turn number of the simulation
     */
    size_t getTurn() const;

    /**
     * Method to go to the next turn
     * @return the number of the turn
     */
    size_t nextTurn();

    /**
     * Generic method to search for the closest Humanoid (either Vampire or Human)
     * @tparam T the type of humanoid we want to look for
     * @param hunter the humanoid looking for another humanoid
     * @return a shared pointer on the target
     */
    template<typename T>
    std::shared_ptr<T> findClosestHumanoid(const std::shared_ptr<Humanoid>& hunter) const;

    /**
```

File - Field.hpp

```hpp
     * Method to convert the field into a vector of strings
     * @return the vector of strings representing the field
     */
    [[nodiscard]] std::vector<std::vector<std::string>> fieldToStringVector() const;

    /**
     * Getter for the height_ attribute
     * @return the height_ attribute
     */
    size_t getHeight() const;

    /**
     * Getter for the width_ attribute
     * @return the width_ attribute
     */
    size_t getWidth() const;

    /**
     * Method to decrement the number of vampire
     */
    void vampireKilled();

    /**
     * Method to decrement the number of human
     */
    void humanKilled();

    /**
     * Method to increment the number of vampire
     */
    void vampireBorn();

    /**
     * Method to give the information if there are alive vampires in the currentField
     */
    bool aliveVampires() const;

    /**
     * Method to give the information if there are alive humans in the currentField
     */
    bool aliveHumans() const;

    /**
     * Method to know if the game is finished or not
     * @return true if finished false if running
     */
    Status isFinished() const;

   /**
    * Getter for the number of humans
    * @return the number of humans
    */
    size_t getNbHuman() const {
       return nbHuman_;
    };

    /**
     * Getter for the number of vampires
     * @return the number of vampires
     */
    size_t getNbVampire() const {
       return nbVampire_;
    };

```

File - Field.hpp

```
127     /**
128      * Method to add a humanoid to the list within the field
129      * @param humanoid the humanoid to add to the list
130      */
131     void addHumanoid(const std::shared_ptr<Humanoid>& humanoid);
132 
133 };
134 
135 #include "Field_Generic_Impl.hpp"
136 
137 
```

File - Field_Generic_Impl.hpp

```cpp
/**
 * @file Implementation of the findClosestHumanoid function
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include <numeric>

/**
 * Method to find the closest humanoid of type T from the hunter.
 * @tparam T the type of the humanoid we want to find
 * @param hunter the humanoid looking for another humanoid
 * @return the closest humanoid of type T from the hunter
 */
template<typename T>
std::shared_ptr<T> Field::findClosestHumanoid(const std::shared_ptr<Humanoid>& hunter) const {
   std::shared_ptr<T> closest;
   size_t minDistance = width_ * height_;
   for (const auto& humanoid: humanoids) {
      if (dynamic_cast<T*>(humanoid.get())) {
         size_t distance = hunter->getPosition().calculateDistance(humanoid->getPosition());
         if (distance < minDistance) {
            minDistance = distance;
            closest = std::dynamic_pointer_cast<T>(humanoid);
         }
      }
   }
   return closest;
}

```

File - FieldStatsCalculator.cpp

```cpp
/**
 * @file Implementation of the FieldStatsCalculator struct
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#include "FieldStatsCalculator.hpp"
#include <iostream>

double FieldStatsCalculator::simulate(size_t height, size_t width, size_t nbVampire,
                                      size_t nbHuman, size_t nbSimulations) {
   size_t successCount = 0;
   std::cout << "Stats are being calculated..." << std::endl;
   for (size_t i = 0; i < nbSimulations; ++i) {
      Field simulatingField(height, width, nbVampire, nbHuman);
      while (simulatingField.isFinished() == Status::RUNNING) {
         simulatingField.nextTurn();
      }
      if (simulatingField.isFinished() == Status::WIN) {
         successCount++;
      }
   }
   return static_cast<double>(successCount) / static_cast<double>(nbSimulations) * 100;
}

```

File - FieldStatsCalculator.hpp

```hpp
/**
 * @file Declaration of the FieldStatsCalculator struct
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include <cstddef>
#include "../field/Field.hpp"

/**
 * Structure to calculate the statistics of n simulations.
 */
struct FieldStatsCalculator {
    static double simulate(size_t height, size_t width, size_t nbVampire, size_t nbHuman, size_t nbSimulations=10000);
};


```

File - Vec2D.cpp

```cpp
/**
 * @file Implementation of the Vec2D struct
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */


#include "Vec2D.hpp"
#include "Random.hpp"
#include <cmath>

Vec2D::Vec2D(int x_, int y_) : x(x_), y(y_) {}

Vec2D::Vec2D() : x(0), y(0) {}

size_t Vec2D::calculateDistance(const Vec2D& other) const {
   /*Vec2D difference = subtraction(other).absValue();
   size_t casesNotInDiagonal = std::abs(difference.x-difference.y);
   return std::min(difference.x, difference.y) + casesNotInDiagonal;*/
   double x_ = std::abs(x - other.x);
   double y_ = std::abs(y - other.y);
   return (size_t) std::round(std::hypot(x_, y_));
}


Vec2D Vec2D::subtraction(const Vec2D& other) const {
   return {this->x - other.x, this->y - other.y};
}

Vec2D Vec2D::addition(const Vec2D& other) const {
   return {this->x + other.x, this->y + other.y};
}

Vec2D Vec2D::getDirection(const Vec2D& other) const {
   int x_ = other.x - x;
   int y_ = other.y - y;

   return {
           x_ == 0 ? 0 : x_ / std::abs(x_),
           y_ == 0 ? 0 : y_ / std::abs(y_)
   };
}

Vec2D Vec2D::getRandomVector(size_t maxX, size_t maxY) {
   return {
      Random::generateFrom0(maxX - 1),
      Random::generateFrom0(maxY - 1)
   };
}

```

File - Vec2D.hpp

```hpp
/**
 * @file Declaration of the Vec2D struct
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include <array>
#include <cstddef>

/**
 * Structure representing a 2D vector with integer coordinates
 */
struct Vec2D {
public:
    int x, y;

    /**
     * Constructor of the Vec2D struct
     * @param x_ x-coordinate of the vector
     * @param y_ y-coordinate of the vector
     */
    Vec2D(int x_, int y_);

    /**
     * Empty constructor of the struct
     */
    Vec2D();

    /**
     * Method to calculate the distance between two vectors
     * @param other the vector with which to calculate the distance
     * @return the distance between the two vectors
     */
    size_t calculateDistance(const Vec2D& other) const;

    /**
     * Method to calculate the subtraction of two vectors
     * @param other the vector with which to calculate the subtraction
     * @return a new vector representing the subtraction of the two vectors
     */
    Vec2D subtraction(const Vec2D& other) const;

   /**
    * Method to calculate the addition of two vectors
    * @param other the vector with which to calculate the addition
    * @return a new vector representing the addition of the two vectors
    */
    Vec2D addition(const Vec2D& other) const;

    /**
     * Method to calculate the direction from one vector to another
     * @param other the vector with which to calculate the direction
     * @return a new vector representing the direction in which the other vector is
     */
    Vec2D getDirection(const Vec2D& other) const;

    /**
     * Method to get a random vector from 0 to given bounds
     * @param maxX the max x-coordinate value
     * @param maxY
```

File - Vec2D.hpp

```
65      * @return
66      */
67     static Vec2D getRandomVector(size_t maxX, size_t maxY);
68
69 };
70
71
72
73
```

File - Random.cpp

```cpp
/**
 * @file Implementation of the Random class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#include "Random.hpp"
#include <chrono>

std::mt19937 Random::rng(
        static_cast<unsigned>(std::chrono::system_clock::now().time_since_epoch().count())
);

int Random::generate(int min, int max) {
   if (min >= max) {
      throw std::invalid_argument("Min must be smaller than max");
   }

   std::uniform_int_distribution<int> int_dist(min, max); // range [min, max[

   return int_dist(rng);
}

int Random::generateFrom0(size_t max) {
   return Random::generate(0, static_cast<int>(max));
}

bool Random::randomBool() {
   int rand = generate(0, 1);
   return rand == 1;
}
```

File - Random.hpp

```hpp
/**
 * @file Declaration of the Random class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include <random>

/**
 * Random class with static methods
 */
struct Random {
private:

    static std::mt19937 rng;

public:

    /**
     * Static method to get a random number between [min, max]
     * @throw invalid_argument if min is greater or equal to max
     * @param min the minimal value
     * @param max the maximal value
     * @return random int between the range given
     */
    static int generate(int min, int max);

    /**
     * Static method to get a random number between [0, max]
     * @param max the maximal value
     * @return random int between 0 and max
     */
    static int generateFrom0(size_t max);

    /**
     * Static method to get a random bool
     * @return either 1 or 0 in a random fashion
     */
    static bool randomBool();

};

```

File - Status.hpp

```hpp
/**
 * @file Declaration of the Status enum
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

/**
 * Enum to represent the status of the game
 */
enum Status {
   RUNNING,
   LOSE,
   WIN
};

```

File - Bite.cpp

```cpp
/**
 * @file Implementation of the Bite class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */



#include "Bite.hpp"
#include "../humanoids/Humanoid.hpp"
#include "../field/Field.hpp"
#include "../utils/Random.hpp"
#include "../humanoids/Vampire.hpp"
#include "Kill.hpp"


Bite::Bite(const std::shared_ptr<Humanoid> &target) : Kill(target) {}

void Bite::execute(Field &field) {
   const auto &humanoid = getHumanoid();
   if (humanoid->isAlive()) {
      if (Random::randomBool()) {
         field.addHumanoid(std::make_shared<Vampire>(humanoid->getPosition()));
         field.vampireBorn();
      }
      Kill::execute(field);
   }
}


```

File - Bite.hpp

```cpp
/**
 * @file Declaration of the Bite class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include "Kill.hpp"

/**
 * Bite class to represent the action of biting a humanoid.
 * It inherits from the Kill class as the action executed
 * with a bite can be either a Kill or a transformation into a Vampire.
 */
class Bite : public Kill {
public:
    /**
     * Constructor of the class Bite representing the action of biting
     * @param hunter the humanoid that will bite
     */
   explicit Bite(const std::shared_ptr<Humanoid>& target);

   /**
    * Execute the bite action
    * @param field the current field
    */
   void execute(Field &field) override;
};

```

File - Kill.cpp

```cpp
/**
 * @file Implementation of the Kill class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#include "Kill.hpp"
#include "../humanoids/Humanoid.hpp"

Kill::Kill(const std::shared_ptr<Humanoid>& target) : Action(target) {}

void Kill::execute(Field& field) {
   if (getHumanoid()->isAlive()) {
      getHumanoid()->kill(field);
   }
}

```

File - Kill.hpp

```hpp
/**
 * @file Declaration of the Kill class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include "Action.hpp"

/**
 * Kill class to represent the action of killing a humanoid.
 * It inherits from the Action class. It is used to kill a humanoid.
 */
class Kill : public Action {

public:

    /**
     * Constructor of the class Kill representing the action of killing
     * @param target the humanoid that will be killed
     */
    explicit Kill(const std::shared_ptr<Humanoid>& target);

    /**
     * Execute the Kill action
     * @param field current field
     */
    void execute(Field& field) override;

};
```

File - Move.cpp

```cpp
/**
 * @file Implementation of the Move class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */


#include <iostream>
#include "Move.hpp"
#include "../humanoids/Humanoid.hpp"
#include "../field/Field.hpp"
#include "../utils/Random.hpp"

Move::Move(size_t range, const std::shared_ptr<Humanoid>& mover, std::optional<Vec2D> target)
        : Action(mover), range_(range), target_(target) {}

void Move::execute(Field& field) {

   std::shared_ptr<Humanoid> mover = getHumanoid();

   if (mover == nullptr) {
      return;
   }

   Vec2D newPosition = mover->getPosition();
   Vec2D direction;

   for (size_t i = 0; i < range_; i++) {
      if (auto target = target_) {
         direction = newPosition.getDirection(target.value());
      } else {
         std::vector<Vec2D> directions = Move::getPossiblePositions(newPosition, field);

         if (directions.empty()) {
            break;
         }

         direction = directions.at(Random::generateFrom0(static_cast<int>(directions.size() - 1)));
      }
      newPosition = newPosition.addition(direction);
   }

   mover->setPosition(newPosition);
}

std::vector<Vec2D> Move::getPossiblePositions(const Vec2D &position, const Field &field) {
   std::vector<Vec2D> nextPositions{};
   nextPositions.reserve(8);

   int maxX = static_cast<int>(field.getWidth());
   int maxY = static_cast<int>(field.getHeight());
   for (int i = -1; i <= 1; i++) {
      for (int j = -1; j <= 1; j++) {
         if (i == 0 && j == 0) {
            continue;
         }

         int x = position.x + i;
         int y = position.y + j;

         if (x < 0 || x >= maxX || y < 0 || y >= maxY) {
            continue;
```

File - Move.cpp

```
65          }
66
67          nextPositions.emplace_back(i, j);
68       }
69    }
70
71    return nextPositions;
72 }
73
```

File - Move.hpp

```hpp
/**
 * @file Declaration of the Move class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include "Action.hpp"
#include "../utils/Vec2D.hpp"
#include <vector>
#include <optional>

class Vec2D;

/**
 * Move class to represent the action of moving a humanoid.
 */
class Move : public Action {

public:

   /**
    * Method to execute the Move action
    * @param field the current field
    */
    void execute(Field& field) override;

    /**
     * Constructor of the class Move representing the action of moving
     * @param range the range of the movement
     * @param mover the humanoid that will move
     * @param target the humanoid that will be moved
     */
    Move(size_t range, const std::shared_ptr<Humanoid>& mover, std::optional<Vec2D> target = std::nullopt);


private:
    size_t range_;
    std::optional<Vec2D> target_;

    /**
     * Method to get the possible positions for the humanoid to move
     * @param position the present position of the humanoid
     * @param field the current field
     * @return vector of all possible positions
     */
   static std::vector<Vec2D> getPossiblePositions(const Vec2D &position, const Field &field);
};
```

File - Action.cpp

```cpp
/**
 * @file Implementation of the Action class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#include "Action.hpp"

Action::Action(const std::shared_ptr<Humanoid>& humanoid) : humanoid_(humanoid) {}

std::shared_ptr<Humanoid> Action::getHumanoid() const {
   return humanoid_.lock();
}

```

File - Action.hpp

```hpp
/**
 * @file Declaration of the Action class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include <memory>

class Field;

class Humanoid;

/**
 * Action class to represent the action made by a humanoid
 */
class Action {

public:
    /**
     * Constructor of the class
     * @param humanoid the humanoid that will execute the action
     */
    explicit Action(const std::shared_ptr<Humanoid>& humanoid);

    /**
     * Default destructor of the class
     */
    virtual ~Action() = default;

    /**
     * Preventing the copy of an action
     */
    Action(const Action&) = delete;

    /**
     * Preventing the copy of an action
     */
    Action& operator=(const Action&) = delete;

    /**
     * Purely virtual method to execute the action
     * @param field the current field
     */
    virtual void execute(Field& field) = 0;

    /**
     * Getter for the humanoid_ attribute
     * @return the humanoid_ attribute
     */
    [[nodiscard]] std::shared_ptr<Humanoid> getHumanoid() const;

private:
    std::weak_ptr<Humanoid> humanoid_;
};

```

File - FieldController.cpp

```cpp
/**
 * @file Implementation of the FieldController class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#include <iostream>
#include <functional>
#include "FieldController.hpp"

const std::string FieldController::CORNER = "+";
const std::string FieldController::HORIZONTAL_BORDER = "-";
const std::string FieldController::VERTICAL_BORDER = "|";

struct FieldController::Command {
    std::string description;
    std::function<void(FieldController&)> action;
};

const std::map<std::string, FieldController::Command> FieldController::COMMANDS = {
        {"q", {"quit",
                    [](FieldController& f) { f.stop(); }}},
        {"n", {"next",
                    [](FieldController& f) { f.nextTurn(); }}},
        {"s", {"statistics",
                    [](FieldController&  f) {
                        std::cout << FieldStatsCalculator::simulate(
                                f.currentField.getHeight(), f.currentField.getWidth(),
                                f.initialNbVampire, f.initialNbHuman) << std::endl;
        }}},
};

FieldController::FieldController(Field& field)
        : currentField(field),
          initialNbHuman(field.getNbHuman()),
          initialNbVampire(field.getNbVampire()) {}

void FieldController::printHorizontalBorder(const Field& field, std::string& output) {
   output += CORNER;
   for (size_t j = 0; j < field.getWidth(); j++) {
      output += HORIZONTAL_BORDER;
   }
   output += CORNER + "\n";
}

void FieldController::printVerticalBorder(const Field& field, std::string& output) {
   auto map = field.fieldToStringVector();
   for (const auto& line: map) {
      output += VERTICAL_BORDER;
      for (const auto& humanoid: line) {
         if (humanoid.empty()) {
            output += ' ';
         } else {
            output += humanoid;
         }
      }
      output += VERTICAL_BORDER + "\n";
   }
}

std::string FieldController::displayFieldToString(const Field& field) {
   std::string output;
```

File - FieldController.cpp

```
65     printHorizontalBorder(field, output);
66     printVerticalBorder(field, output);
67     printHorizontalBorder(field, output);
68     return output;
69 }
70
71 void FieldController::display(const Field& field) {
72     std::cout << displayFieldToString(field);
73     std::cout << std::flush;
74 }
75
76 void FieldController::start() {
77     running = true;
78     display(currentField);
79     while (running && currentField.isFinished() == Status::RUNNING) {
80         std::cout << "[" + std::to_string(currentField.getTurn()) + "] ";
81         for (const auto& [key, cmd]: COMMANDS) {
82             std::cout << key << "<" << cmd.description << " ";
83         }
84         std::cout << ": ";
85         std::string command;
86         std::getline(std::cin, command);
87
88         auto it = COMMANDS.find(command);
89         if (it == COMMANDS.end()) {
90             std::cout << "Invalid command << " << command << std::endl;
91             continue;
92         }
93         handleCommand(it->second);
94     }
95     running = false;
96 }
97
98 void FieldController::handleCommand(const Command& command) {
99     command.action(*this);
100 }
101
102 void FieldController::stop() {
103     running = false;
104 }
105
106 void FieldController::nextTurn() {
107     currentField.nextTurn();
108     display(currentField);
109 }
110
111
112
113
114
115
116
117
```

File - FieldController.hpp

```hpp
/**
 * @file Declaration of the FieldController class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include <string>
#include "../field/Field.hpp"
#include "../stats/FieldStatsCalculator.hpp"

#include <map>

/**
 * FieldController a class used to display the field and run it. It will
 * have a field and will display it on the console. It will be able to handle
 * different commands passed by the terminal. n for the next turn, q for quitting
 * and s to have the stats of many simulations.
 */
class FieldController {
public:
    explicit FieldController(Field& field);

    /**
     * Method to convert the currentField to a string
     * @param field the currentField to convert
     * @return the string representing the currentField
     */
    static std::string displayFieldToString(const Field& field);

    /**
     * Method to simulate the currentField and display it
     * @param field the currentField to simulate
     */
     void start();

     /**
      * Method to stop the simulation
      */
     void stop();

     /**
      * Method to go to the next turn of the simulation
      */
     void nextTurn();

    /**
     * Default constructor of the class
     */
    FieldController() = delete;

private:
    struct Command;
    static const std::map<std::string, Command> COMMANDS;
    static const std::string CORNER;
    static const std::string HORIZONTAL_BORDER;
    static const std::string VERTICAL_BORDER;
    Field& currentField;
    bool running = false;
    size_t initialNbHuman;
    size_t initialNbVampire;
```

File - FieldController.hpp

```
65
66      /**
67       * Method to handle the command given by the user
68       * @param command the command to handle
69       */
70       void handleCommand(const Command& command);
71
72      /**
73       * Method to print the horizontal border of the currentField
74       * @param field the currentField to print the border of
75       * @param output the string to print the border to
76       */
77      static void printHorizontalBorder(const Field& field, std::string& output);
78
79      /**
80       * Method to print the vertical border of the currentField
81       * @param field the currentField to print the border of
82       * @param output the string to print the border to
83       */
84      static void printVerticalBorder(const Field& field, std::string& output);
85
86      /**
87       * Method to display the currentField on the console
88       * @param field the currentField to display
89       */
90      static void display(const Field& field);
91
92  };
93
94
```

File - Buffy.cpp

```cpp
/**
 * @file Implementation of the Buffy class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#include "Buffy.hpp"
#include "Vampire.hpp"
#include "../action/Kill.hpp"
#include "../action/Move.hpp"
#include "../field/Field.hpp"

Buffy::Buffy(const Vec2D& position) : Humanoid(position) {}

std::unique_ptr<Action> Buffy::chooseAction(const Field& field) {
   if (!field.aliveVampires()) {
      // Move like a human
      return std::make_unique<Move>(Humanoid::getActionRange(), shared_from_this());
   }

   auto closest = field.findClosestHumanoid<Vampire>(shared_from_this());
   size_t distanceToClosestVampire = this->getPosition().calculateDistance(closest->getPosition());
   if (distanceToClosestVampire <= 1) {
      return std::make_unique<Kill>(closest);
   }
   return std::make_unique<Move>(getActionRange(), shared_from_this(), closest->getPosition());
}

size_t Buffy::getActionRange() const {
   return 2;
}

std::string Buffy::getType() {
   return "B";
}

void Buffy::kill(Field &field) {
   // Buffy cannot die
}
```

File - Buffy.hpp

```hpp
/**
 * @file Declaration of the Buffy class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include "Humanoid.hpp"
#include <string>

/**
 * Class to reprensent the hunteress Buffy. It inherits from Humanoid and will override its function.
 * Buffy is super speedy and can move 2 units at a time. She can only kill at a range of 1
 */
class Buffy : public Humanoid {
private:

public:

   /**
    * Constructor of the Buffy class
    * @param position starting position of the object
    */
    explicit Buffy(const Vec2D& position);

    /**
     * Method that will choose the next action of Buffy depending on its environment
     * @param field the current field
     * @return the next action the object will execute
     */
     std::unique_ptr<Action> chooseAction(const Field& field) override;

     /**
      * Method to get the range of the action Buffy
      * @return
      */
    size_t getActionRange() const override;

    /**
     * Getter for the displayer to know what's the type of the object
     * @return the string representing the object type
     */
    std::string getType() override;

    /**
     * Method overriding the kill method of the Humanoid class. Buffy can't be killed.
     * She's a superwoman
     * @param field the current field
     */
    void kill(Field& field) override;
};
```

File - Human.cpp

```cpp
/**
 * @file Implementation of the Human class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#include "Human.hpp"
#include "../action/Move.hpp"
#include "../field/Field.hpp"

Human::Human(const Vec2D& position) : Humanoid(position) {}

std::unique_ptr<Action> Human::chooseAction(const Field& field) {
   return std::make_unique<Move>(getActionRange(), shared_from_this());
}

std::string Human::getType() {
   return "h";
}

void Human::kill(Field& field) {
   Humanoid::kill(field);
   field.humanKilled();
}

```

File - Human.hpp

```hpp
/**
 * @file Declaration of the Human class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include <string>
#include <vector>
#include "Humanoid.hpp"

/**
 * Class representing a human in the game. It inherits from the Humanoid class.
 * Its behavior is simple, it moves 1 unit at a time in a random direction and can be killed.
 */
class Human : public Humanoid {
public:

   /**
    * Constructor of the Human class
    * @param position the starting position of the human
    */
   explicit Human(const Vec2D &position);

   /**
    * Getter for the displayer to know what's the type of the object
    * @return the string representing the object type
    */
   std::string getType() override;

   /**
    * Method that will choose the next action of the human depending on its environment
    * @param field the current field
    * @return the next action the human will execute
    */
   std::unique_ptr<Action> chooseAction(const Field &field) override;

   /**
    * Method to kill the humanoid and decrement the number of humans currently in the game
    * @param field the current field
    */
   void kill(Field &field) override;
};
```

File - Vampire.cpp

```cpp
/**
 * @file Implementation of the Vampire class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */


#include "Vampire.hpp"
#include "Human.hpp"
#include "../action/Kill.hpp"
#include "../action/Bite.hpp"
#include "../action/Move.hpp"
#include "../field/Field.hpp"

Vampire::Vampire(const Vec2D& position) : Humanoid(position) {}

std::string Vampire::getType() {
   return "v";
}

std::unique_ptr<Action> Vampire::chooseAction(const Field& field) {
   if (!field.aliveHumans()) {
      return nullptr;
   }
   auto closestHuman = field.findClosestHumanoid<Human>(shared_from_this());
   size_t distanceToClosestHuman = this->getPosition().calculateDistance(closestHuman->getPosition());
   if(distanceToClosestHuman > getActionRange()) {
      return std::make_unique<Move>(getActionRange(), shared_from_this(), closestHuman->getPosition());
   }
   return std::make_unique<Bite>(closestHuman);
}

void Vampire::kill(Field& field) {
   Humanoid::kill(field);
   field.vampireKilled();
}
```

File - Vampire.hpp

```cpp
/**
 * @file Declaration of the Vampire class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include "Humanoid.hpp"

/**
 * Class representing a vampire in the game. It inherits from the Humanoid class.
 * It will move 1 unit at a time towards the closest human.
 * It can kill humans at a range of 1.
 */
class Vampire : public Humanoid {

public:

    /**
     * Constructor of the Vampire class
     * @param position the starting position of the object
     */
    explicit Vampire(const Vec2D &position);

    /**
     * Getter for the displayer to know what's the type of the object
     * @return the string representing the object type
     */
    std::string getType() override;

    /**
     * Method that will choose the next action of the vampire depending on its environment
     * @param field the current field
     * @return the next action the object will execute
     */
    std::unique_ptr<Action> chooseAction(const Field &field) override;

    /**
     * Method to kill the vampire. It will also decrement the number of vampires currently in the game
     * @param field the current field
     */
    void kill(Field &field) override;
};

```

File - Humanoid.cpp

```cpp
/**
 * @file Implementation of the Random class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#include "Humanoid.hpp"

Humanoid::Humanoid(const Vec2D& position) : position_(position), imAlive(true) {}

bool Humanoid::isAlive() const {
   return imAlive;
}

Vec2D Humanoid::getPosition() const {
   return position_;
}

void Humanoid::setAction(const Field& field) {
   nextAction = chooseAction(field);
}

void Humanoid::executeAction(Field& field) {
   if (nextAction) {
      nextAction->execute(field);
      nextAction.reset();
   }
}

void Humanoid::kill(Field& field) {
   imAlive = false;
}

void Humanoid::setPosition(const Vec2D& newPos) {
   position_ = newPos;
}

size_t Humanoid::getActionRange() const {
   return 1;
}
```

File - Humanoid.hpp

```cpp
/**
 * @file Declaration of the Humanoid class
 *
 * @author Rachel Tranchida
 * @author Massimo Stefani
 * @author Quentin Surdez
 */

#pragma once

#include <string>
#include "../utils/Vec2D.hpp"
#include "../action/Action.hpp"


class Field;

/**
 * Humanoid abstract class representing a humanoid
 */
class Humanoid : public std::enable_shared_from_this <Humanoid> {
public:
    /**
     * Constructor of the class
     * @param position at which it will be created
     */
    explicit Humanoid(const Vec2D& position);

    /**
     * Default destructor
     */
    virtual ~Humanoid() = default;

    /**
     * Purely abstract method to get the next action of the humanoid
     * @param field the current currentField
     * @return a pointer to the action created
     */
    virtual std::unique_ptr<Action> chooseAction(const Field& field) = 0;

    /**
     * Setter for the nextAction attribute
     * @param field the current currentField
     */
    virtual void setAction(const Field& field);

    /**
     * Method to execute the action on the currentField
     * @param field the current currentField
     */
    virtual void executeAction(Field& field);

    /**
     * Getter for the imAlive attribute
     * @return the value of imAlive
     */
    bool isAlive() const;

    /**
     * Method to get the string corresponding to each humanoid
     * @return the string representation of the humanoid
     */
    virtual std::string getType() = 0;

```

File - Humanoid.hpp

```
65      /**
66       * Getter for the position attribute
67       * @return the position attribute
68       */
69      Vec2D getPosition() const;
70
71      /**
72       * Setter for the position attribute
73       * @param newPos the new value of the position attribute
74       */
75      void setPosition(const Vec2D& newPos);
76
77      /**
78       * Method to kill the humanoid
79       */
80      virtual void kill(Field& field);
81
82      /**
83       * Abstract method for the range of movement each humanoid has
84       * @return the range for movement
85       */
86      virtual size_t getActionRange() const;
87
88  private:
89      std::unique_ptr<Action> nextAction;
90      Vec2D position_;
91      bool imAlive;
92  };
93
94
```