```cpp
#include <iostream>
#include "Controller/Controller.hpp"

int main(int argc, char* argv[]) {
   const int NB_ARGS = 5;

   if (argc != NB_ARGS) {
      std::cerr << "Please enter a valid number of parameters : " << argv[0]
                << "<width> <height> <number of humans> <number of vampires
>"
                << std::endl;
      return 1;
   }

   size_t width, height, nbHumans, nbVampires;
   try {
      for (int i = 1; i < argc; ++i) {
         if (std::stoi(argv[i]) < 0) {
            std::cerr << "Values should be positives." << std::endl;
            return 1;
         }
      }
      width = static_cast<size_t>(std::stoul(argv[1]));
      height = static_cast<size_t>(std::stoul(argv[2]));
      nbHumans = static_cast<size_t>(std::stoul(argv[3]));
      nbVampires = static_cast<size_t>(std::stoul(argv[4]));
   } catch (const std::invalid_argument& e) {
      return 1;
   } catch (const std::out_of_range& e) {
      std::cerr << "An argument is out of range." << std::endl;
      return 1;
   }

   Controller controller(width, height, nbHumans, nbVampires);
   controller.run();

   return 0;
}
```

```cpp
/**
 * @file Field.cpp
 * @brief Declaration of the Field class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include "Field.hpp"

Field::Field(const size_t width, const size_t height, size_t nbHumans,
             size_t nbVampires): width(width), height(height), nbHumans(nbHumans),
             nbVampires(nbVampires), turn(0)
{
    for (size_t i = 0; i < nbHumans; i++) {
       humanoids.emplace_back(
          new Human(Position(Position::random(width, height))));
    }
    for (size_t i = 0; i < nbVampires; i++) {
       humanoids.emplace_back(
          new Vampire(Position(Position::random(width, height))));
    }
    humanoids.emplace_back(new Buffy(Position::random(width, height)));
}

int Field::nextTurn() {
   // Déterminer les prochaines actions
   for (std::list<Humanoid *>::iterator it = humanoids.begin();
        it != humanoids.end(); it++)
      (*it)->setAction(*this);
   // Executer les actions
   for (std::list<Humanoid *>::iterator it = humanoids.begin();
        it != humanoids.end(); it++)
      (*it)->executeAction(*this);
   // Enlever les humanoides tués
   for (std::list<Humanoid *>::iterator it = humanoids.begin();
        it != humanoids.end();) {
      if (!(*it)->isAlive()) {
         Humanoid *toDelete = *it;
         it = humanoids.erase(it); // suppression de l'élément dans la liste
         delete toDelete; // destruction de l'humanoide référencé
      } else
         ++it;
   }
   return turn++;
}

size_t Field::getNbHuman() const {
   return nbHumans;
}

size_t Field::getNbVampire() const {
   return nbVampires;
```

```
52 }
53
54 std::list<Humanoid*> Field::getHumanoids() const {
55    return humanoids;
56 }
57
58 size_t Field::getWidth() const {
59    return width;
60 }
61
62 size_t Field::getHeight() const {
63    return height;
64 }
65
66 int Field::getTurn() const {
67    return turn;
68 }
69
70 void Field::vampireDie() {
71    if(nbVampires) --nbVampires;
72 }
73
74 void Field::humanDie() {
75    if(nbHumans) --nbHumans;
76 }
77
78 void Field::vampireBorn() {
79    ++nbVampires;
80 }
81
82 void Field::addHumanoid(Humanoid* humanoid) {
83    for(auto it = humanoids.begin(); it != humanoids.end(); it++)
84       if(*it == humanoid) return;
85    humanoids.emplace_back(humanoid);
86 }
87
88
89
```

```cpp
/**
 * @file Field.hpp
 * @brief Definition of the Field class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef LAB4_FIELD_HPP
#define LAB4_FIELD_HPP

#include <list>
#include <limits>
#include "../Person/Humanoid.hpp"
#include "../Person/Human.hpp"
#include "../Person/Vampire.hpp"
#include "../Person/Buffy.hpp"

class Humanoid;

class Human;

class Vampire;

/**
 * @brief Class representing a field.
 * @details A field has a width, a height, a number of humans and a number
 * of vampires.
 */
class Field {
   std::list<Humanoid *> humanoids;
   int turn = 0;
   const size_t width, height;
   size_t nbHumans, nbVampires;

public:

   /**
    * @brief Constructor of the Field class.
    * @param width Width of the field.
    * @param height Height of the field.
    * @param nbHumans Number of humans on the field.
    * @param nbVampires Number of vampires on the field.
    */
   Field(const size_t width, const size_t height, size_t nbHumans,
         size_t nbVampires);


   /**
    * @brief We prepare and execute the action of each humanoid, removing
  those
    * that are dead and moving on to the next round.
    */
   int nextTurn();
```

```
 52
 53    /**
 54     * @brief Find the closest humanoid of a specific type to a given
    humanoid.
 55     * @tparam T The type of humanoid to find.
 56     * @param h The humanoid from which to find the closest humanoid.
 57     * @return The closest humanoid of the specified type.
 58     */
 59    template<typename T>
 60    T *findClosest(const Humanoid &h) const {
 61       T *closest = nullptr;
 62       int minDist = std::numeric_limits<int>::max();
 63       T *specificType;
 64
 65       for (Humanoid *humanoid: humanoids) {
 66          specificType = dynamic_cast<T *>(humanoid);
 67          if (specificType != nullptr) {
 68             int dist = humanoid->getPosition().distance(h.getPosition());
 69             if (dist < minDist) {
 70                minDist = dist;
 71                closest = specificType;
 72             }
 73          }
 74       }
 75       return closest;
 76    }
 77
 78    /**
 79     * @brief Get the number of humans on the field.
 80     * @return The number of humans on the field.
 81     */
 82    size_t getNbHuman() const;
 83
 84    /**
 85     * @brief Get the number of vampires on the field.
 86     * @return The number of vampires on the field.
 87     */
 88    size_t getNbVampire() const;
 89
 90    /**
 91     * @brief Get the list of humanoids on the field.
 92     * @return The list of humanoids on the field.
 93     */
 94    std::list<Humanoid *> getHumanoids() const;
 95
 96    /**
 97     * @brief Get the width of the field.
 98     * @return The width of the field.
 99     */
100    size_t getWidth() const;
101
102    /**
```

```
103     * @brief Get the height of the field.
104     * @return The height of the field.
105     */
106    size_t getHeight() const;
107
108    /**
109     * @brief Get the current turn.
110     * @return The current turn.
111     */
112    int getTurn() const;
113
114    /**
115     * @brief Decrease the number of vampires by one.
116     */
117    void vampireDie();
118
119    /**
120     * @brief Decrease the number of humans by one.
121     */
122    void humanDie();
123
124    /**
125     * @brief Increase the number of vampires by one.
126     */
127    void vampireBorn();
128
129    /**
130     * @brief Add a humanoid to the field.
131     * @param humanoid The humanoid to add.
132     */
133    void addHumanoid(Humanoid *humanoid);
134 };
135
136 #endif //LAB4_FIELD_HPP
137
```

```cpp
/**
 * @file Random.cpp
 * @brief Declaration of the Random class and related operators.
 * @authors Demont Kilian & Graf Calvin
*/

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

std::mt19937 Random::gen = std::mt19937(std::random_device()());

int Random::generateInt(int max) {
   return generateInt(0, max);
}

int Random::generateInt(int min, int max) {
   if(max <= min) throw std::invalid_argument("Max must be greater or equal than "
                                              "min");
   std::uniform_int_distribution<int> dist(min, max - 1);
   return dist(gen);
}

bool Random::generateBool() {
   // Generate a random integer between 0 and 1 (generateInt not including max)
   return generateInt(0, 2);
}

```

```cpp
/**
 * @file Random.hpp
 * @brief Definition of the Random class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef BUFFY_RANDOM_HPP
#define BUFFY_RANDOM_HPP

#include <random>

/**
 * @brief Class representing a random number generator.
 */
class Random {
   static std::mt19937 gen;
public:
   /**
    * Generate a random integer between min (including) and max (not including)
    * @param min the minimum value
    * @param max the maximum value
    * @return a random integer between min and max
    */
   static int generateInt(int min, int max);

   /**
    * Generate a random integer between 0 (including) and max (not including)
    * @param max the maximum value
    * @return a random integer between 0 and max
    */
   static int generateInt(int max);

   /**
    * Generate a random boolean
    * @return a random boolean
    */
   static bool generateBool();
};

#endif //BUFFY_RANDOM_HPP

```

```cpp
/**
 * @file Position.cpp
 * @brief Declaration of the Position class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

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

const Position Position::UP = Position(0, -1);
const Position Position::UP_RIGHT = Position(1, -1);
const Position Position::RIGHT = Position(1, 0);
const Position Position::DOWN_RIGHT = Position(1, 1);
const Position Position::DOWN = Position(0, 1);
const Position Position::DOWN_LEFT = Position(-1, 1);
const Position Position::LEFT = Position(-1, 0);
const Position Position::UP_LEFT = Position(-1, -1);

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

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

size_t Position::getX() const {
   return x;
}

size_t Position::getY() const {
   return y;
}

size_t Position::distance(const Position & pos) const {
   return (size_t)round(hypot(abs((x - pos.x)), abs((y - pos.y))));
}

Position Position::random(size_t width, size_t height) {
   return Position(Random::generateInt(width),
                   Random::generateInt(height));
}

Position Position::getDirection(const Position &pos) const {
    int dx = pos.getX() - x;
    int dy = pos.getY() - y;
    return Position(dx == 0 ? 0 : dx / abs(dx),
                    dy == 0 ? 0 : dy / abs(dy));
}

void Position::move(const Position &direction) {
   x += direction.getX();
   y += direction.getY();
}

```

```cpp
/**
 * @file Position.hpp
 * @brief Definition of the Position class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef LAB4_POSITION_HPP
#define LAB4_POSITION_HPP

#include <cstddef>

/**
 * @brief Class representing a position
 */
class Position {
   size_t x, y;

public:
   /**
    * @brief Static positions representing the 8 directions.
    */
   static const Position
      UP,
      UP_RIGHT,
      RIGHT,
      DOWN_RIGHT,
      DOWN,
      DOWN_LEFT,
      LEFT,
      UP_LEFT;

   /**
    * @brief Default constructor of the Position class.
    */
   Position();

   /**
    * @brief Constructor of the Position class.
    * @param x The x coordinate of the position.
    * @param y The y coordinate of the position.
    */
   Position(size_t x, size_t y);

   /**
    * @brief Getter for the x coordinate of the position.
    * @return The x coordinate of the position.
    */
   size_t getX() const;

   /**
    * @brief Getter for the y coordinate of the position.
    * @return The y coordinate of the position.
```

```
53      */
54     size_t getY() const;
55 
56     /**
57      * @brief Compute the distance between two positions.
58      * @param pos The position to which the distance will be computed.
59      * @return The distance between the two positions.
60      */
61     size_t distance(const Position & pos) const;
62 
63     /**
64      * @brief Get the direction to go to reach a position.
65      * @param pos The current position.
66      * @return The position to go to reach the given position.
67      */
68     Position getDirection(const Position& pos) const;
69 
70     /**
71      * @brief Move the position in a given direction.
72      * @param direction The direction in which the position will be moved.
73      */
74     void move(const Position& direction);
75 
76     /**
77      * @brief Generate a random position in a given width and height.
78      * @param width The width of the field.
79      * @param height The height of the field.
80      * @return A random position in the field.
81      */
82     static Position random(size_t width, size_t height);
83 };
84 
85 
86 
87 
88 #endif //LAB4_POSITION_HPP
89 
```

```cpp
/**
 * @file Kill.cpp
 * @brief Declaration of the Kill class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

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

Kill::Kill(Humanoid& humanoid) : Action(humanoid) {}

void Kill::execute(Field& f) {
   if(getHumanoid()->isAlive()) getHumanoid()->die(f);
}
```

```cpp
/**
 * @file Kill.hpp
 * @brief Definition of the Kill class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef BUFFY_KILL_HPP
#define BUFFY_KILL_HPP

#include "Action.hpp"

/**
 * @brief Class representing the action of killing a humanoid.
 */
class Kill : public Action {
public:
   /**
    * @brief Constructor of the Kill class.
    * @param humanoid The humanoid that will execute the action.
    */
   explicit Kill(Humanoid& humanoid);

   void execute(Field& f) override;
};

#endif //BUFFY_KILL_HPP

```

```cpp
/**
 * @file Move.cpp
 * @brief Declaration of the Move class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include "Move.hpp"
#include "../Person/Humanoid.hpp"
#include "../Field/Field.hpp"
#include "../Utils/Random.hpp"

Move::Move(const size_t range, Humanoid& humanoid, const Humanoid* target)
   : range(range), Action(humanoid), target(target) {}

void Move::execute(Field& field) {
   Position direction;
   Position newPosition = getHumanoid()->getPosition();

   for (size_t i = 0; i < range; ++i) {
      if (target) direction = newPosition.getDirection(target->getPosition
   ());
      else {
         std::vector<const Position *> directions =
            getDirectionsPossible(newPosition, field);
         if (directions.empty()) return;
         direction = *directions.at(Random::generateInt(directions.size()));
      }
      newPosition.move(direction);
   }
   getHumanoid()->setPosition(newPosition);
}

std::vector<const Position*> Move::getDirectionsPossible(const Position& pos
   ,
                                                            const Field& field
   ) const {
   std::vector<const Position*> directionsPossible;
   size_t x = pos.getX();
   size_t y = pos.getY();

   const Position* directions[] = {
      &Position::UP,
      &Position::UP_RIGHT,
      &Position::RIGHT,
      &Position::DOWN_RIGHT,
      &Position::DOWN,
      &Position::DOWN_LEFT,
      &Position::LEFT,
      &Position::UP_LEFT
   };

   for (const Position* direction : directions) {
```

```
50        size_t newX = x + direction->getX();
51        size_t newY = y + direction->getY();
52        size_t width = field.getWidth();
53        size_t height = field.getHeight();
54        if (newX >= 0 && newX < width && newY >= 0 && newY < height)
55           directionsPossible.emplace_back(direction);
56     }
57     return directionsPossible;
58 }
59
```

```hpp
/**
 * @file Move.hpp
 * @brief Definition of the Move class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef BUFFY_MOVE_HPP
#define BUFFY_MOVE_HPP

#include "Action.hpp"
#include <vector>
#include "../Utils/Position.hpp"

class Humanoid;

/**
 * @brief Class representing the action of moving a humanoid.
 */
class Move : public Action {
   size_t range;
   const Humanoid* target;
public:
   /**
    * @brief Constructor of the Move class.
    * @param range The range of the move.
    * @param humanoid The humanoid that will execute the move.
    * @param target The target of the move.
    */
   explicit Move(const size_t range, Humanoid& humanoid,
                 const Humanoid* target = nullptr);

   /**
    * @brief Execute the move of the humanoid on the field.
    * @param field The field on which the move will be executed.
    */
   std::vector<const Position*> getDirectionsPossible(const Position& pos,
                                                      const Field& field) const;

   void execute(Field& field) override;
};

#endif //BUFFY_MOVE_HPP

```

```cpp
/**
 * @file Action.cpp
 * @brief Declaration of the Action class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include "Action.hpp"

Action::Action(Humanoid &humanoid) : humanoid(&humanoid) {}

Humanoid* Action::getHumanoid() const {
   return humanoid;
}
```

```cpp
/**
 * @file Action.hpp
 * @brief Definition of the Action class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef LAB4_ACTION_HPP
#define LAB4_ACTION_HPP

class Field;
class Humanoid;

/**
 * @brief Class representing an action that a humanoid can execute on the field.
 */
class Action {
   Humanoid* humanoid;
public:
   /**
    * @brief Constructor of the Action class.
    * @param humanoid The humanoid that will execute the action.
    */
   Action(Humanoid &humanoid);

   /**
    * @brief Destructor of the Action class.
    */
   virtual ~Action() = default;

   // Delete copy constructor and assignment operator of the action class
   Action(const Action&) = delete;
   Action& operator=(const Action&) = delete;

   /**
    * @brief Getter for the humanoid of the action.
    * @return The humanoid of the action.
    */
   Humanoid* getHumanoid() const;

   /**
    * @brief Execute the action of the humanoid on the field.
    * @param f The field on which the action will be executed.
    */
   virtual void execute(Field& f) = 0;
};


#endif //LAB4_ACTION_HPP


```

```cpp
/**
 * @file Transform.cpp
 * @brief Declaration of the Transform class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include "Transform.hpp"
#include "../Person/Vampire.hpp"

Transform::Transform(Humanoid& humanoid) : Action(humanoid) {}

void Transform::execute(Field& f) {
   if (getHumanoid()->isAlive()) {
      getHumanoid()->die(f);
      f.addHumanoid(new Vampire(getHumanoid()->getPosition()));
      f.vampireBorn();
   }
}
```

```cpp
/**
 * @file Transform.hpp
 * @brief Definition of the Transform class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef BUFFY_TRANSFORM_HPP
#define BUFFY_TRANSFORM_HPP

#include "Action.hpp"

/**
 * @brief Class representing the transformation of a human into a vampire.
 */
class Transform : public Action {
public:
   /**
    * @brief Constructor of the Transform class.
    * @param humanoid Humanoid to transform.
    */
   explicit Transform(Humanoid& humanoid);

   void execute(Field& f) override;
};

#endif //BUFFY_TRANSFORM_HPP

```

```cpp
/**
 * @file Buffy.cpp
 * @brief Declaration of the Buffy class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include <iostream>
#include "Buffy.hpp"
#include "Human.hpp"

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

Action* Buffy::getNextAction(const Field& field) {
   if(field.getNbVampire() == 0) return new Move(Human::getMoveRange(),
                                                 *this);

   Humanoid* target = field.findClosest<Vampire>(*this);
   if(target->getPosition().distance(this->getPosition()) <= KILL_RANGE)
      return new Kill(*target);
   return new Move(MOVE_RANGE, *this, target);
}

char Buffy::display() const {
   return 'B';
}

```

```cpp
/**
 * @file Buffy.hpp
 * @brief Definition of the Buffy class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef LAB4_BUFFY_HPP
#define LAB4_BUFFY_HPP

#include "Humanoid.hpp"
#include "Vampire.hpp"
#include "../Action/Kill.hpp"
#include "../Action/Move.hpp"

/**
 * @brief Class representing a Buffy.
 * @details A Buffy is a humanoid that can kill vampires.
 */
class Buffy : public Humanoid {
   static constexpr size_t KILL_RANGE = 1;
   static constexpr size_t MOVE_RANGE = 2;

public:
   /**
    * @brief Constructor of the Buffy class.
    * @param position The position of the Buffy.
    */
   Buffy(const Position& position);

   char display() const override;

   Action* getNextAction(const Field& field) override;
};

#endif //LAB4_BUFFY_HPP

```

```cpp
/**
 * @file Human.cpp
 * @brief Declaration of the Human class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include <iostream>
#include "Human.hpp"
#include "../Action/Move.hpp"
#include "../Field/Field.hpp"

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

Action* Human::getNextAction(const Field& field) {
    return new Move(MOVE_RANGE, *this);
}

char Human::display() const {
    return 'h';
}

size_t Human::getMoveRange() {
    return MOVE_RANGE;
}

void Human::die(Field& field) {
    Humanoid::die(field);
    field.humanDie();
}

```

```hpp
/**
 * @file Human.hpp
 * @brief Definition of the Human class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef LAB4_HUMAIN_HPP
#define LAB4_HUMAIN_HPP

#include "Humanoid.hpp"

/**
 * @brief Class representing a human.
 * @details A human can only move
 */
class Human : public Humanoid {
   static constexpr size_t MOVE_RANGE = 1;
public:
   /**
    * @brief Constructor of the Human class.
    * @param position The position of the human.
    */
   Human(const Position& position);

   /**
    * @brief Get the move range of the human.
    * @return The move range of the human.
    */
   static size_t getMoveRange();

   Action* getNextAction(const Field& field) override;

   char display() const override;

   void die(Field& field) override;
};

#endif //LAB4_HUMAIN_HPP

```

```cpp
/**
 * @file Vampire.cpp
 * @brief Declaration of the Vampire class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include "Vampire.hpp"
#include "../Utils/Random.hpp"
#include "../Action/Transform.hpp"

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

Action* Vampire::getNextAction(const Field& field) {
    if(field.getNbHuman() == 0) return nullptr;

    Human* target = field.findClosest<Human>(*this);

    if(target->getPosition().distance(this->getPosition()) <= KILL_RANGE) {
      if(Random::generateBool()) return new Kill(*target);
      else return new Transform(*target);
   }
    return new Move(MOVE_RANGE, *this, target);
}

char Vampire::display() const {
   return 'V';
}

void Vampire::die(Field& field) {
   Humanoid::die(field);
   field.vampireDie();
}

```

```cpp
/**
 * @file Vampire.hpp
 * @brief Definition of the Vampire class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef LAB4_VAMPIRE_HPP
#define LAB4_VAMPIRE_HPP

#include "Humanoid.hpp"
#include "../Field/Field.hpp"

/**
 * @brief Class representing a vampire on the field.
 * @details A vampire can kill or transform a human if he is close enough,
 *       otherwise he will move towards the closest human.
 */
class Vampire : public Humanoid {
   static constexpr size_t KILL_RANGE = 1;
   static constexpr size_t MOVE_RANGE = 1;
public:
   /**
    * @brief Constructor of the Vampire class.
    * @param position The position of the vampire.
    */
   Vampire(const Position& position);

   Action* getNextAction(const Field& field) override;

   char display() const override;

   void die(Field& field) override;
};

#endif //LAB4_VAMPIRE_HPP

```

```cpp
/**
 * @file Humanoid.cpp
 * @brief Declaration of the Humanoid class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include "Humanoid.hpp"

Humanoid::Humanoid(const Position& position) : position(position), alive(true),
action(nullptr) {}

Humanoid::~Humanoid() {
   delete action;
}

const Position& Humanoid::getPosition() const {
   return position;
}

void Humanoid::setPosition(Position pos) {
   this->position = pos;
}

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

void Humanoid::die(Field& field) {
   if(action != nullptr) {
      delete action;
      action = nullptr;
   }
   alive = false;
}

void Humanoid::setAction(Field& field) {
   this->action = getNextAction(field);
}

void Humanoid::executeAction(Field& field) {
      if (action != nullptr) {
         action->execute(field);
         delete action;
         action = nullptr;
      }
}
```

```cpp
/**
 * @file Humanoid.hpp
 * @brief Definition of the Humanoid class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef LAB4_HUMANOIDE_HPP
#define LAB4_HUMANOIDE_HPP

#include "../Utils/Position.hpp"
#include "../Action/Action.hpp"

class Field;

/**
 * @brief Class representing a humanoid on the field.
 */
class Humanoid {
   Position position;
   bool alive;
   Action* action;

   public:
   /**
    * @brief Constructor of the Humanoid class.
    * @param pos The position of the humanoid.
    */
   explicit Humanoid(const Position& pos);

   // Delete copy constructor and assignment operator of the humanoid class
   Humanoid(const Humanoid&) = delete;
   Humanoid& operator=(const Humanoid&) = delete;

   /**
    * @brief Destructor of the Humanoid class.
    */
   virtual ~Humanoid();

   /**
    * @brief Getter for the position of the humanoid.
    * @return The position of the humanoid.
    */
   const Position& getPosition() const;

   /**
    * @brief Setter for the position of the humanoid.
    * @param pos The new position of the humanoid.
    */
   void setPosition(Position pos);

   /**
    * @brief Check if the humanoid is alive.
```

```
53      * @return True if the humanoid is alive, false otherwise.
54      */
55     bool isAlive() const;
56 
57     /**
58      * @brief Kill the humanoid.
59      * @param field The field on which the humanoid is.
60      */
61     virtual void die(Field& field);
62 
63     /**
64      * @brief Set the action of the humanoid.
65      * @param field The field on which the humanoid is.
66      */
67     void setAction(Field& field);
68 
69     /**
70      * @brief Execute the action of the humanoid on the field.
71      * @param field The field on which the action will be executed.
72      */
73     void executeAction(Field& field);
74 
75     /**
76      * @brief Get the next action of the humanoid.
77      * @param field The field on which the humanoid is.
78      * @return The next action of the humanoid.
79      */
80     virtual Action* getNextAction(const Field& field) = 0;
81 
82     /**
83      * @brief Display the humanoid on the field.
84      * @return The character representing the humanoid on the field.
85      */
86     virtual char display() const = 0;
87 };
88 
89 #endif //LAB4_HUMANOIDE_HPP
90 
```

```cpp
/**
 * @file Display.cpp
 * @brief Declaration of the Display class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include <iostream>
#include <iomanip>
#include "Display.hpp"
#include "../Controller/Controller.hpp"

Display::Display(const Field& field) : field(field), gameBoard(field.getHeight(),
             std::vector<const Humanoid*>(field.getWidth(),nullptr))
{}

void Display::print() {
   const Humanoid** toDisplay = nullptr;
   for(Humanoid* h : field.getHumanoids()) {
      gameBoard.at(h->getPosition().getY())
               .at(h->getPosition().getX()) = h;
   }
   printHorizontalBorder(field);

   for(size_t y = 0; y < field.getHeight(); y++) {
      std::cout << VERTICAL_BORDER;
      for(size_t x = 0; x < field.getWidth(); x++) {
         toDisplay = &gameBoard.at(y).at(x);
         if(*toDisplay) std::cout << (*(*toDisplay)).display();
         else std::cout << EMPTY;
      }
      std::cout << VERTICAL_BORDER << std::endl;
   }
   printHorizontalBorder(field);
   clearGameBoard();
}
/*
void Display::clear() const {
#ifdef _WIN32
   system("cls");
#elif __unix__
   system("clear");
#endif
}
*/

void Display::clearGameBoard() {
   for(auto& line : gameBoard) {
      for(auto& cell : line) cell = nullptr;
   }
}

```

```
52 void Display::printHorizontalBorder(const Field& field) const {
53    std::cout << CORNER << std::setfill(HORIZONTAL_BORDER)
54          << std::setw((size_t)field.getWidth() + 1) << CORNER << std::endl;
55 }
56
57 void Display::printPrompt() const {
58    std::cout << "[" << field.getTurn() << "] "
59       << Controller::QUIT << ">uit "
60       << Controller::STATISTICS << ">tatistics "
61       << Controller::NEXT << ">ext: ";
62 }
```

```cpp
/**
 * @file Display.hpp
 * @brief Definition of the Display class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef BUFFY_DISPLAY_HPP
#define BUFFY_DISPLAY_HPP

#include "../Field/Field.hpp"
#include <vector>

/**
 * @brief Class representing the display of the game.
 */
class Display {
   std::vector<std::vector<const Humanoid*>> gameBoard;
   const Field& field;
   static constexpr char CORNER = '+',
                         HORIZONTAL_BORDER = '-',
                         VERTICAL_BORDER = '|',
                         EMPTY = ' ';

   /**
    * @brief Print the horizontal border of the game board.
    * @param field The field to print the border of.
    */
   void printHorizontalBorder(const Field& field) const;
public:
   /**
    * @brief Constructor of the Display class.
    * @param field The field to display.
    */
   Display(const Field& field);

   /**
    * @brief Print the game board.
    */
   void print();

   /**
    * @brief Print the prompt with commands available.
    */
   void printPrompt() const;

   /*
   /**
    * @brief Clear the console.
    */
   /*
   void clear() const;
   */
```

```
53
54    /**
55     * @brief Clear the vector representing the game board
56     */
57    void clearGameBoard();
58 };
59
60 #endif //BUFFY_DISPLAY_HPP
61
```

```cpp
/**
 * @file Controller.cpp
 * @brief Declaration of the Controller class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include <iostream>
#include "Controller.hpp"
#include "../Statistics/statisticsCalculator.hpp"

Controller::Controller(const size_t width, const size_t height, size_t nbHumans,
                       size_t nbVampires) : field(width, height, nbHumans,
                       nbVampires), display(field), finished(false)
{}

void Controller::run() {
   display.print();
   display.printPrompt();
   while(!finished) handleCommand();
}

void Controller::handleCommand() {
   char command;
   std::cin >> command;
   switch(command) {
      case Controller::QUIT:
         quit();
         break;
      case Controller::STATISTICS:
         statistics();
         break;
      case Controller::NEXT:
         nextTurn();
         break;
      default:
         std::cout << "Invalid command" << std::endl;
         break;
   }
   if(!finished) display.printPrompt();
}

void Controller::nextTurn() {
   field.nextTurn();
   display.print();
   //display.clear();
}

void Controller::quit() {
   finished = true;
}

```

```
52 void Controller::statistics() {
53    std::cout << NB_SIMULATIONS << " simulations in progress..." << std::
   endl;
54    double res = statisticsCalculator::simulate(field.getWidth(),
55                  field.getHeight(),field.getNbHuman(),
56                  field.getNbVampire(), NB_SIMULATIONS);
57    std::cout << "Buffy's win rate: " << res << "%" << std::endl;
58 }
59
```

```cpp
/**
 * @file Controller.hpp
 * @brief Definition of the Controller class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef BUFFY_CONTROLLER_HPP
#define BUFFY_CONTROLLER_HPP

#include "../Field/Field.hpp"
#include "../Displayer/Display.hpp"

/**
 * @brief Class representing the controller of the game.
 */
class Controller {
   Field field;
   Display display;

   bool finished;
   static constexpr size_t NB_SIMULATIONS = 10000;

   /**
    * @brief Quit the game.
    */
   void quit();

   /**
    * @brief Display Buffy's average win rate with current field status
    */
   void statistics();

   /**
    * @brief Handle the command entered by the user.
    */
   void handleCommand();

   /**
    * @brief Execute the next turn of the game.
    */
   void nextTurn();

   public:
   /**
    * @brief Constants representing the commands that can be entered by the user.
    */
   static constexpr char QUIT = 'q',
      STATISTICS = 's',
      NEXT = 'n';

   /**
```

```
52     * @brief Constructor of the Controller class.
53     * @param width The width of the field.
54     * @param height The height of the field.
55     * @param nbHumans The number of humans in the field.
56     * @param nbVampires The number of vampires in the field.
57     */
58    Controller(const size_t width, const size_t height, size_t nbHumans,
   size_t
59    nbVampires);
60
61    /**
62     * @brief Run the game.
63     */
64    void run();
65 };
66
67 #endif //BUFFY_CONTROLLER_HPP
68
```

```cpp
/**
 * @file statisticsCalculator.cpp
 * @brief Declaration of the statisticsCalculator class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#include "statisticsCalculator.hpp"

double statisticsCalculator::simulate(size_t width, size_t height, size_t nbHuman,
                                      size_t nbVampire, size_t nbSimulations) {
   if(nbSimulations == 0 || nbHuman == 0) return 0;
   if(nbVampire == 0) return 100;

   size_t success = 0;
   for (size_t i = 0; i < nbSimulations; ++i) {
      Field simulatingField(width, height, nbHuman, nbVampire);
      while (simulatingField.getNbVampire() > 0) simulatingField.nextTurn();
      if (simulatingField.getNbHuman() > 0) success++;
   }
   return static_cast<double>(success) / static_cast<double>(nbSimulations) * 100;
}
```

```cpp
/**
 * @file statisticsCalculator.hpp
 * @brief Definition of the statisticsCalculator class and related operators.
 * @authors Demont Kilian & Graf Calvin
 */

#ifndef BUFFY_STATISTICSCALCULATOR_HPP
#define BUFFY_STATISTICSCALCULATOR_HPP

#include "../Field/Field.hpp"

/**
 * @brief Struct representing a statistics calculator.
 */
struct statisticsCalculator {

   /**
    * @brief Simulate the game with the given parameters.
    * @param width The width of the field.
    * @param height The height of the field.
    * @param nbHuman The number of humans on the field.
    * @param nbVampire The number of vampires on the field.
    * @param nbSimulations The number of simulations to run.
    * @return Buffy's average win rate percentage in simulations.
    */
   static double simulate(size_t width, size_t height, size_t nbHuman,
                          size_t nbVampire, size_t nbSimulations);
};

#endif //BUFFY_STATISTICSCALCULATOR_HPP

```